The New EDA: Explanatory Data Analysis

Brad Luen

2020-02-01

Contents

1 What's EDA? 5
1.1 What's not EDA? . . . 5
1.2 Goals of explanatory data analysis . . . 5
1.3 How do you learn to do EDA? . . . 6

2 Describing a quantitative variable 7
2.1 Individuals and distributions . . . 7
2.2 Learning ggplot2 . . . 8
2.3 Fits and residuals . . . 32
2.4 Transformations . . . 48
2.5 Robust fits . . . 60

3 Bivariate data 69
3.1 Linear modeling: Fly eyes . . . 69
3.2 Nonlinear modeling . . . 77
3.3 Robust modeling . . . 87
3.4 Comparing bivariate distributions . . . 105
3.5 Truly bivariate data . . . 112

4 Time series data 121
4.1 Time series objects . . . 121
4.2 Carbon dioxide: Seasonal time series . . . 122
4.3 Distributions in time . . . 128
4.4 Comparing time series: Presidential elections . . . 133

5 Trivariate data 139
5.1 Trivariate data: Ethanol . . . 139
5.2 Burning rubber . . . 156
5.3 Using weights: Immigrants and income . . . 178
5.4 Contour plots . . . 190
5.5 Truly 3D plots: Wireframes . . . 206

6 Hypervariate data 225
6.1 Four variables: Modeling ozone . . . 225

6.2 Four or more variables 241

7 Binary responses **251**
7.1 Logistic regression, plus dealing with messy data 251
7.2 Logistic regression with multiple predictors 264

8 Multiway data **291**
8.1 Two-way data, and also maps 291
8.2 Multiway data 307
8.3 The barley controversy 315

9 Count data **325**
9.1 Stop and frisk 325

10 Categorical responses **343**
10.1 Ordered categorical responses: polr() 343
10.2 Unordered categorical responses 354

11 High-dimensional data **373**
11.1 Regression with lots of predictors 373
11.2 All subsets 377
11.3 If prediction is a major goal 382
11.4 PCA 393
11.5 PCA vs. factor analysis 401
11.6 Chernoff faces 404
11.7 Example: The 88th Congress 404

12 Random effects **415**
12.1 Radon 415
12.2 Random effects 423
12.3 Uranium, too 427

13 Appendix: Miscellaneous examples and data tricks **433**
13.1 Tuberculosis 433
13.2 Baseball 435
13.3 Catterplots 441
13.4 Things I need to improve in this book 442

Chapter 1

What's EDA?

"EDA" traditionally stands for **exploratory data analysis**, but I've decided I like **explanatory data analysis** better.

In S350/520 we spent most of our time estimating and testing hypotheses and models. In S470/670 we'll spend most of our time coming up with hypotheses and models for data. In practice this means we'll draw a lot of graphs and we won't have to calculate any P-values.

1.1 What's not EDA?

There's a somewhat useful distinction between **exploratory** and **confirmatory** data analysis. In confirmatory data analysis, you (ideally) start out with a well-posed question (e.g. "Does X cause Y, and if so, how much of it?") and a probability model. The probability model is most rigorously justified through a randomized experiment or a random survey. The end product of the confirmatory analysis is usually a P-value and/or some kind of estimate (e.g. a confidence interval.)

Off to the side of the exploratory/confirmatory divide is **prediction**. The best predictions are often made by complex machine learning methods that puny humans can't understand. In EDA, prediction is not the main focus. However, as in all modeling, having some predictive ability is a positive sign that your model is not complete rubbish.

1.2 Goals of explanatory data analysis

- Come up with a descriptive model. The model might be probabilistic, but accurate probability models are very hard to fit unless you have either reams of data or an explicit random structure (as in randomization.)

- Instead of answering "Does X cause Y?", EDA is more suited to suggesting answers to the reversed question "What might explain variation in Y? How about X1? Does X1 still explain anything after controlling for X2 and X3?" EDA doesn't give us definitive causal answers, but it can suggest designs for follow-up confirmatory studies.

1.3 How do you learn to do EDA?

It's like learning a language. Simple stuff you can learn by yourself; harder stuff you're better off practising with a partner; even harder stuff you really need to practise with someone who's much better than you at the task in question so that they can tell you when you're messing up. To make the best use of everyone's one time, you should do the following:

- In your own time, **type** out the R code in this book (don't just copy-paste.) Every time you run a line, think about the purpose of that line.
- Work on small problems with a fellow student around to help you when you get stuck.
- When working on larger projects, keep in regular contact with an experienced analyst who will hopefully help you see how all the little pieces fit into a bigger picture.
- When working on a very big project, get regular feedback from an experienced analyst and/or someone with subject matter knowledge to tell you if something is a good idea or not.

Chapter 2

Describing a quantitative variable

2.1 Individuals and distributions

With one quantitative variable, the first question to ask is: Do you want to show all the data, or just summarize it? If you're interest in the specific individuals in your data set, showing all the data is a good idea.

On the other hand, if you have too many observations to deal with individually, or you only care about the distribution, you may want to find a **model** that describes the data well: it summarizes what's going on in the data reasonably accurately without excessive complexity. In this course, we use "model" loosely: here, we may mean a formal model like a normal distribution, or an implicit model (whenever you draw something like a density plot, you're implicitly fitting a model.) A dot plot or bar graph would be a good option.

If we want to study the distribution of data (and we don't think there's strong dependence we need to explicitly model, as in sequential data), then the basic steps are:

1. Plot the data. Does it look reasonably symmetric? Right-skewed? Left-skewed? If it's positive and right-skewed, try a log transformation. Otherwise we should be hesitant to do transformations without a good reason, since in EDA we above all want to interpret our outputs.
2. Model the **center** or **level** of each group in the distribution. If the data (after any transformation) is approximately symmetric, give means; if the data is still skewed (or if there are notable outliers), give medians.
3. Look at the **residuals**. In particular, is there **homoskedasticity**: are the spreads about the same for the different groups? If so, that simplifies the model considerably. If not, then quanitfy the spreads (give SDs or

quartiles.)

4. *If* the previous steps have gone well, check: do the residuals approximately follow a known distribution, such as the normal? If so, that's nice. If not, it makes it much harder to make probabilistic statements, but that's rarely the primary goal of EDA. (If the data isn't even homoskedastic, you can't do probabilistic stuff in any case, so there's no point in checking normality.)

2.2 Learning ggplot2

READ: Cleveland pp. 16–31.

2.2.1 Univariate plots: Choral singers again

In S320/520 we learned the basic plots of one quantitative variable:

- ECDF plots
- Histograms/density
- Boxplots
- Normal QQ plots

We'll learn how to redraw all these plots using the `ggplot2` package because it's a more complete "grammar" of graphics, but mostly because it produces much more attractive graphs than base R.

If you've taken my S320/520 class you'll remember we use a data set containing the heights (in inches) of members of the New York Choral Society. That data is in the `lattice` package in R, so load that:

```
library(lattice)
```

We'll need the `ggplot2` library, of course. If you have't yet installed it, then do so: `install.packages("ggplot2")`. Then load it:

```
library(ggplot2)
```

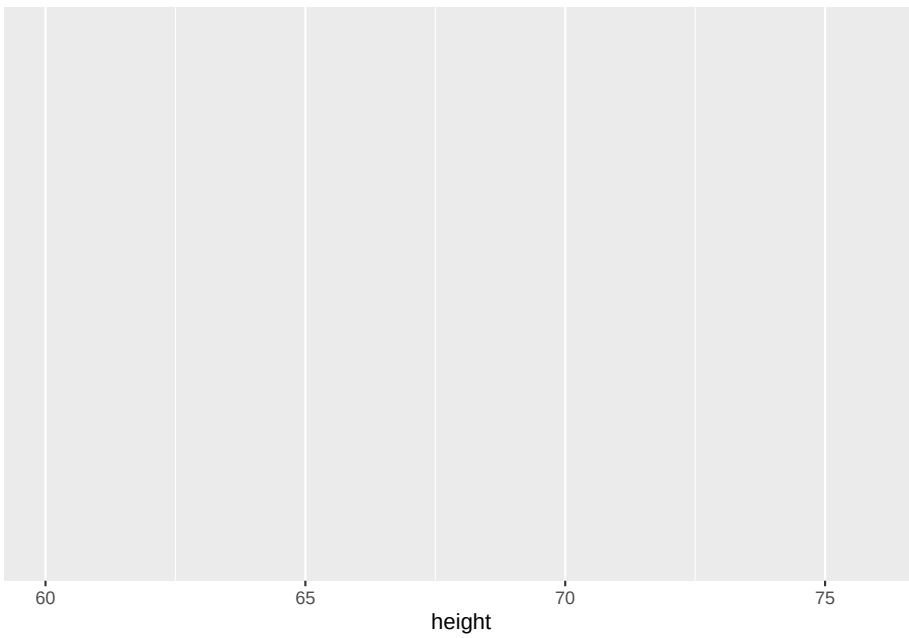
2.2.2 ggplot syntax

The two major inputs to the `ggplot()` function are:

- A data frame (or a "tibble", see below)
- A "mapping" using the `aes()` function (short for "aesthetic.")

We use `aes()` to tell the function how to use the varaibles in the data frame, e.g. to specify x- and y-variables. For example, suppose we want to draw a graph using `height` from the `singer` data frame as our x-variable. We use the following syntax:

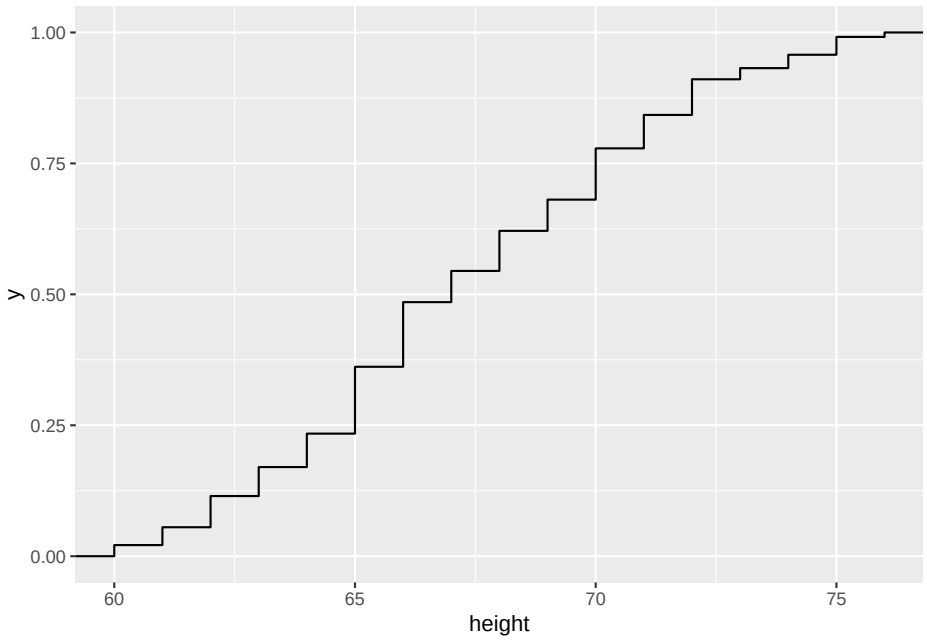
```
ggplot(singer, aes(x=height))
```



Now this doesn't actually draw a graph because we haven't specified what kind of graph we want to draw. To do this, we add stuff to our previous line, where stuff is most commonly a call to a function beginning with `stat_` or `geom_`.

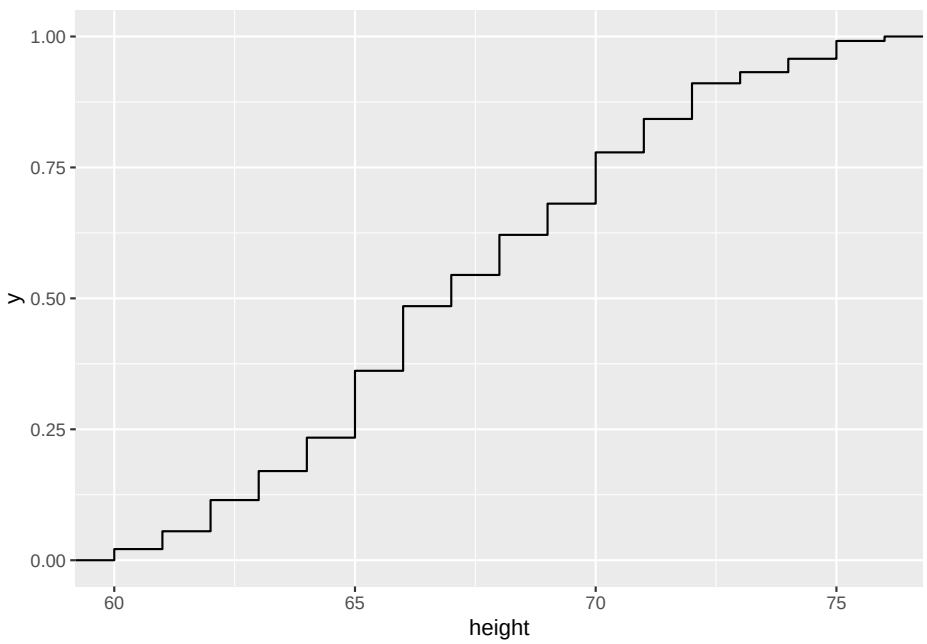
Let's draw an *ECDF plot*. Recall that the ECDF at x is the proportion of the data less than or equal to x. All we need to do is add `+ stat_ecdf()` to our previous command:

```
ggplot(singer, aes(x=height)) + stat_ecdf()
```



Note that the `ggplot()` command creates an object to which new stuff can be added. That is, we could created the same plot by doing:

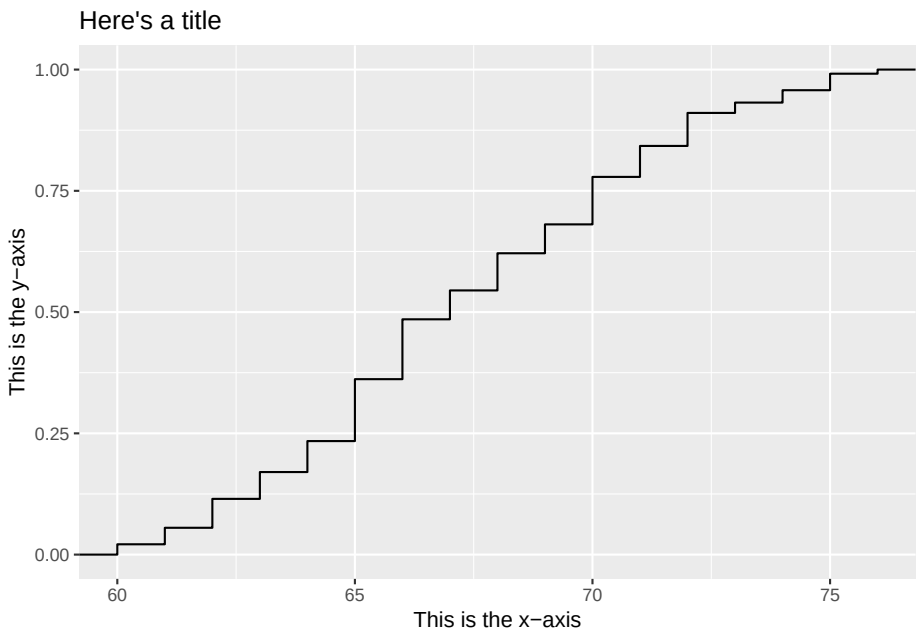
```
singer.gg = ggplot(singer, aes(x=height))
singer.gg + stat_ecdf()
```



Writing the code this way is arguably easier to read, but then you have to remember the names of the objects you create.

If you create plots to present to other people, you should of course label them:

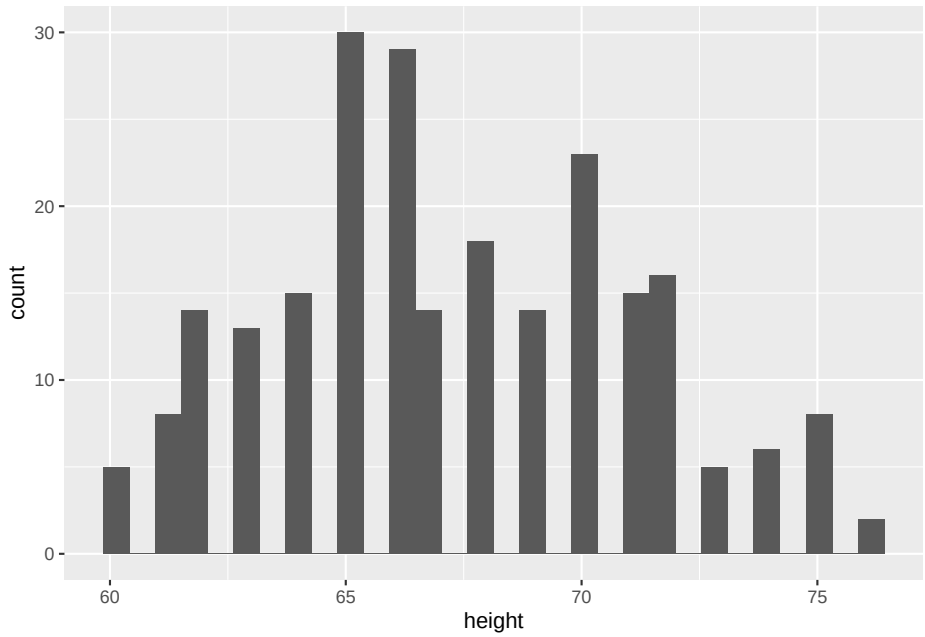
```
gg = singer.gg + stat_ecdf() + xlab("This is the x-axis") + ylab("This is the y-axis")
gg + ggtitle("Here's a title")
```



On the other hand, if you're exploring the data yourself then you don't need to spend too much time making sure things are neatly labeled (particularly if you have descriptive variable names.)

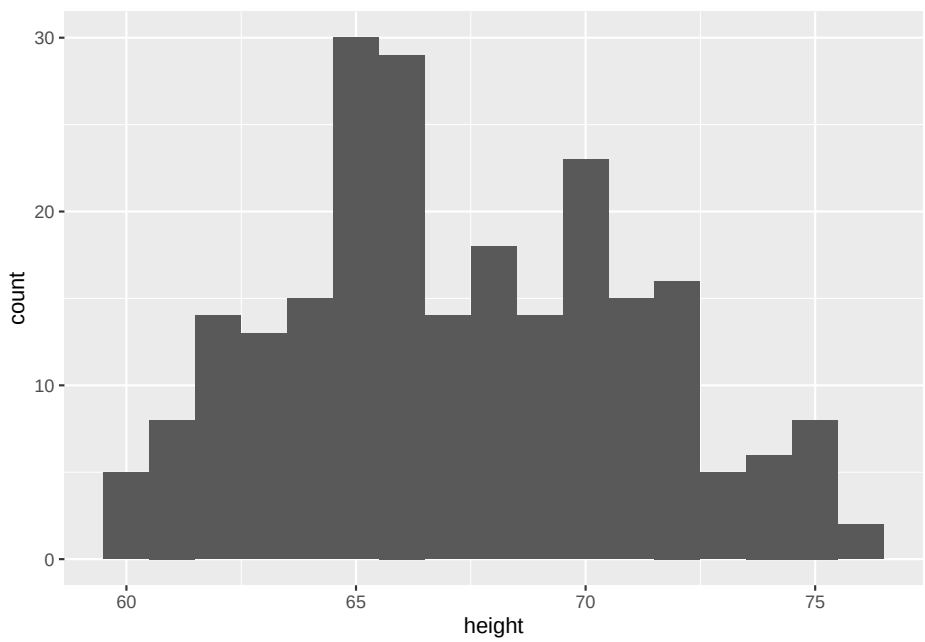
The CDF is not an object for which many people have intuition, so it's generally more useful to do a *histogram* or a *density plot*. Usually the density plot is a better estimate of the PDF, while the histogram gives you a clearer idea of the variability in the data. For a histogram:

```
ggplot(singer, aes(x=height)) + geom_histogram()
```



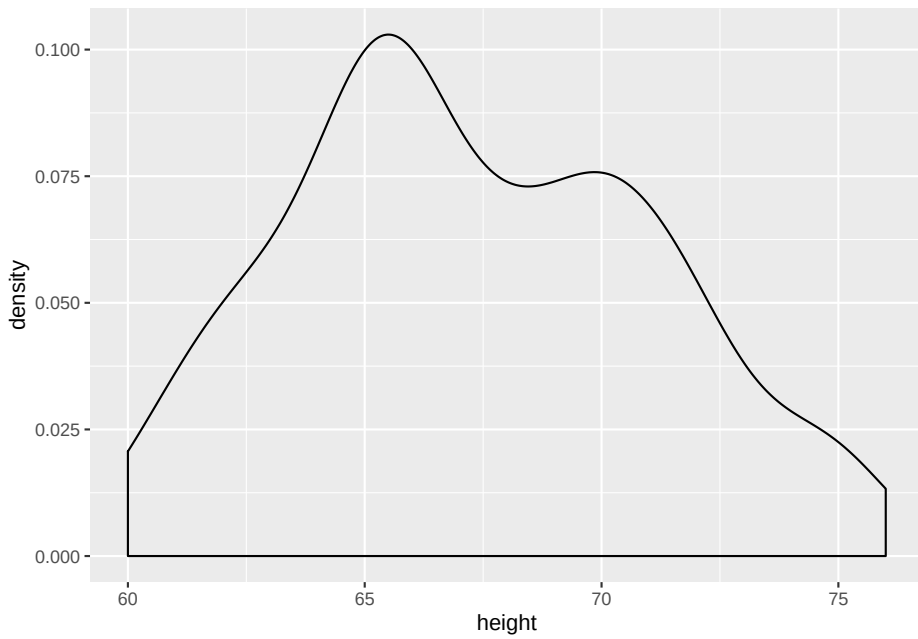
This is ugly. Since we know the heights are measured to the nearest inch, let's specify bins of width 1 inch:

```
ggplot(singer, aes(x=height)) + geom_histogram(binwidth=1)
```



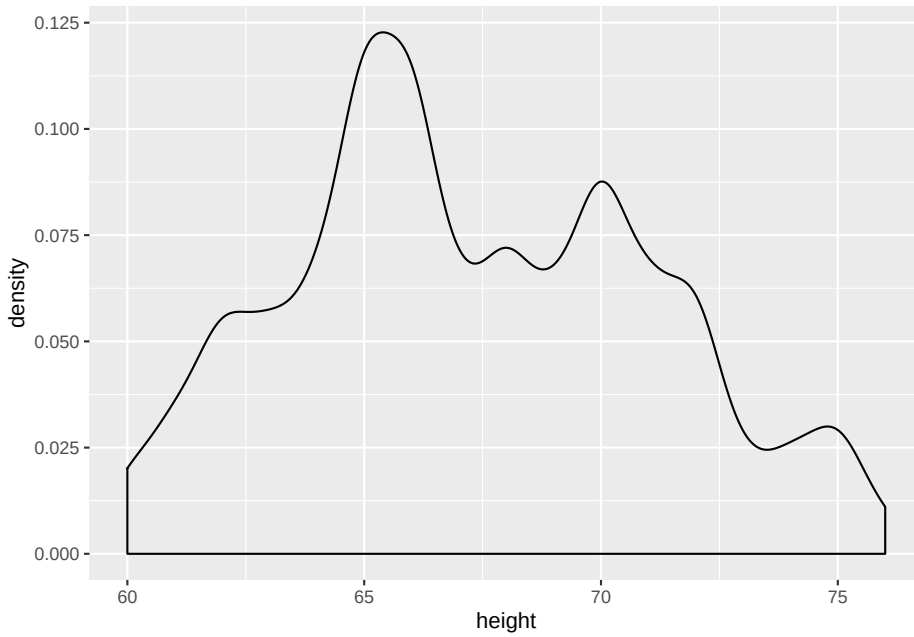
If you prefer using a density plot to estimate the PDF:

```
ggplot(singer, aes(x=height)) + geom_density()
```



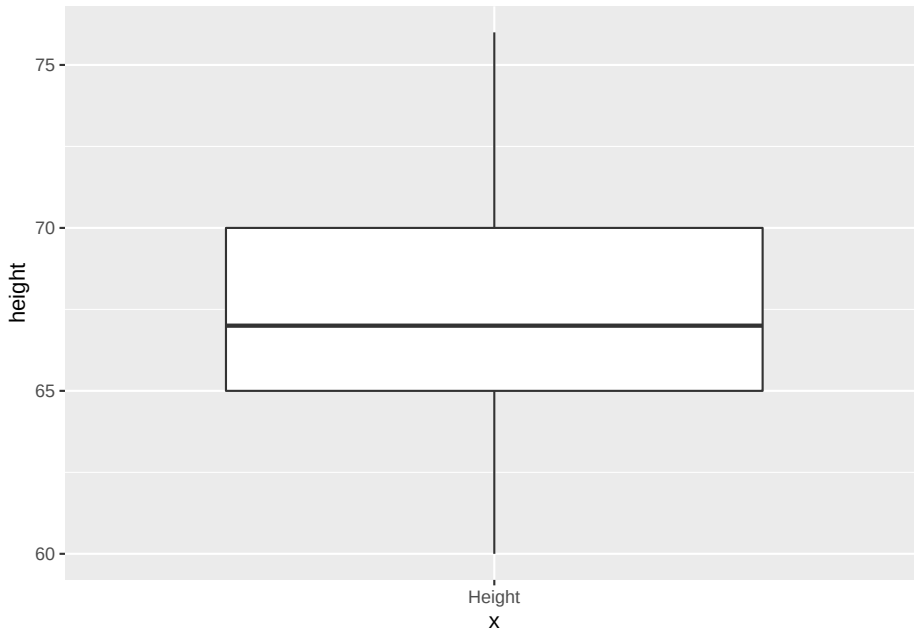
If you don't like your density plot, use `adjust` as an argument within `geom_density()` to change the bandwidth. An adjust value of less than 1 makes the plot less smooth, while a value of 1 makes it smoother:

```
ggplot(singer, aes(x=height)) + geom_density(adjust = 0.5)
```



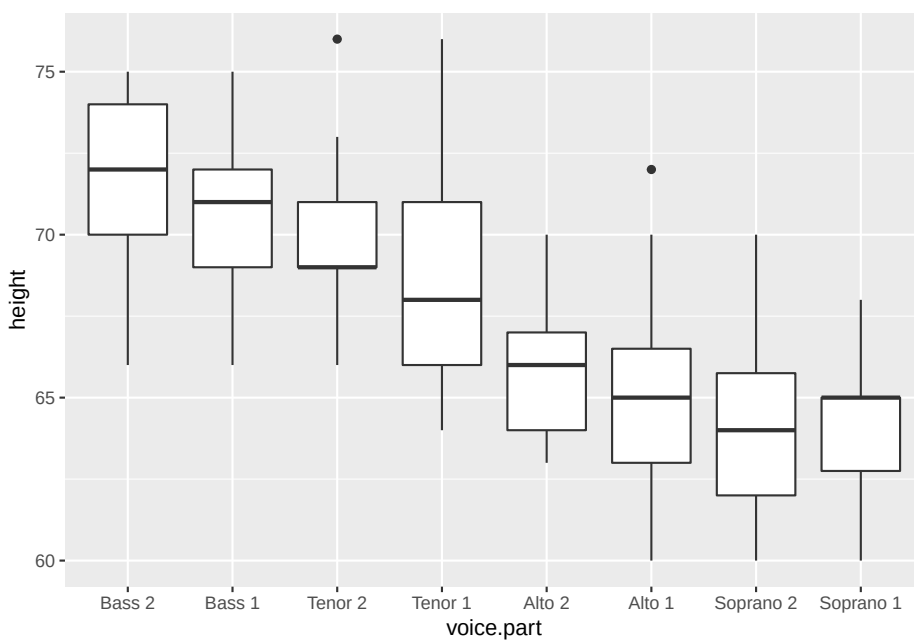
Recall that a *boxplot* gives the five-number summary for a set of data: minimum excluding outliers, first quartile, median, third quartile, and maximum excluding outliers. Drawing a single boxplot is a bit trickier because it requires both an x- and a y-variable. Here's a workaround:

```
ggplot(singer, aes(x="Height", y=height)) + geom_boxplot()
```



Note that drawing one boxplot by itself is usually not very useful. It's much better to draw a number of boxplots to compare them. We can draw a boxplot of heights for each voice part:

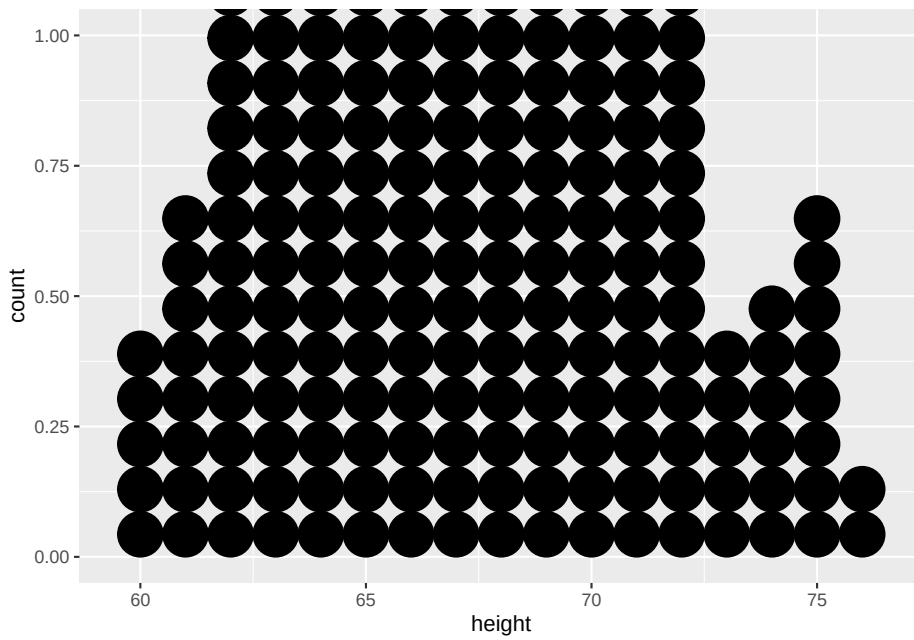
```
ggplot(singer, aes(x=voice.part, y=height)) + geom_boxplot()
```



2.2.3 Dotplots

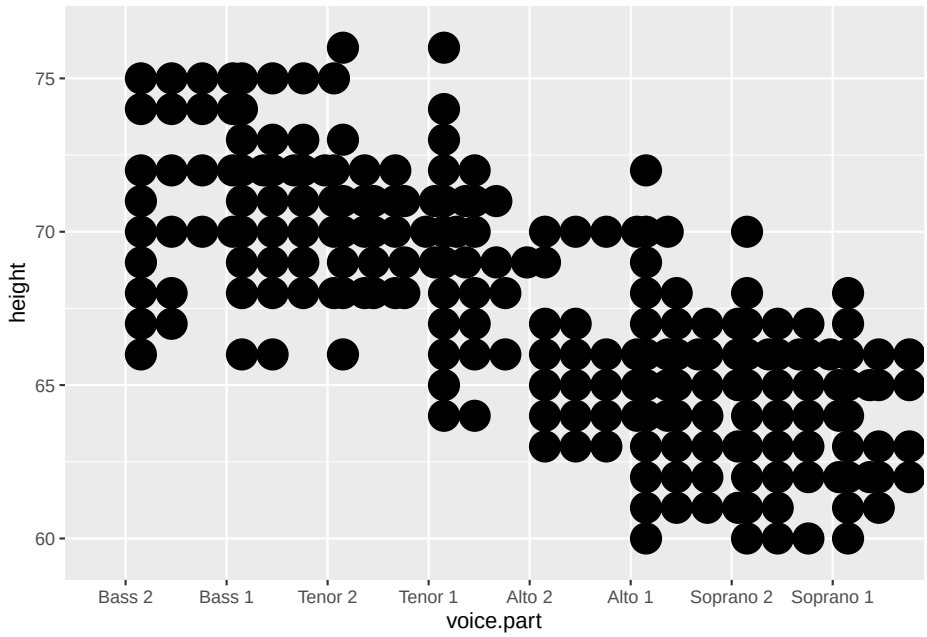
To see all the data, draw a **dotplot**:

```
ggplot(singer, aes(x = height)) + geom_dotplot(binwidth = 1)
```



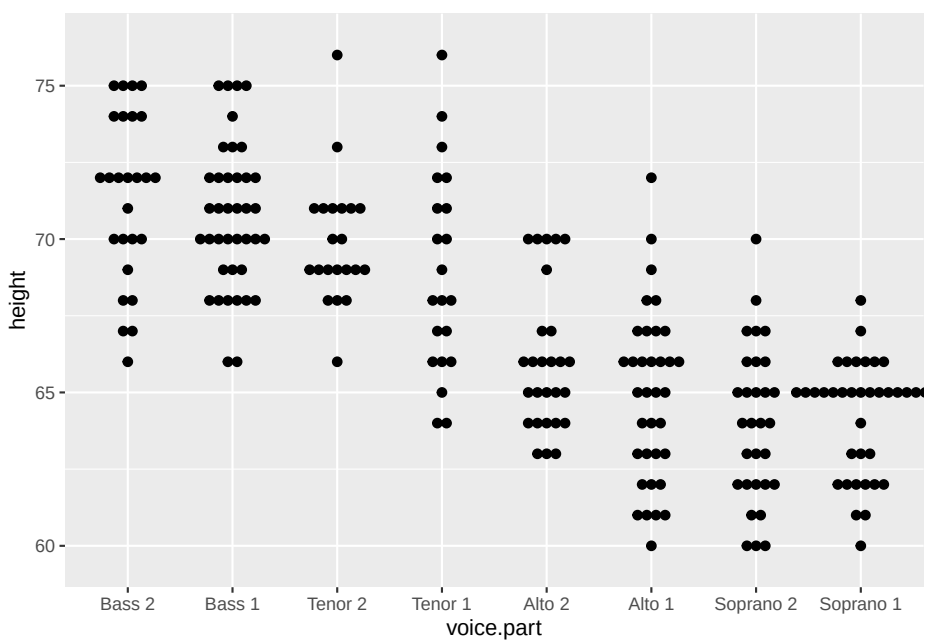
With one variable, all we have is an unreadable version of a histogram. But now we can include a second variable:

```
ggplot(singer, aes(x = voice.part, y = height)) + geom_dotplot(binaxis = "y", binwidth
```



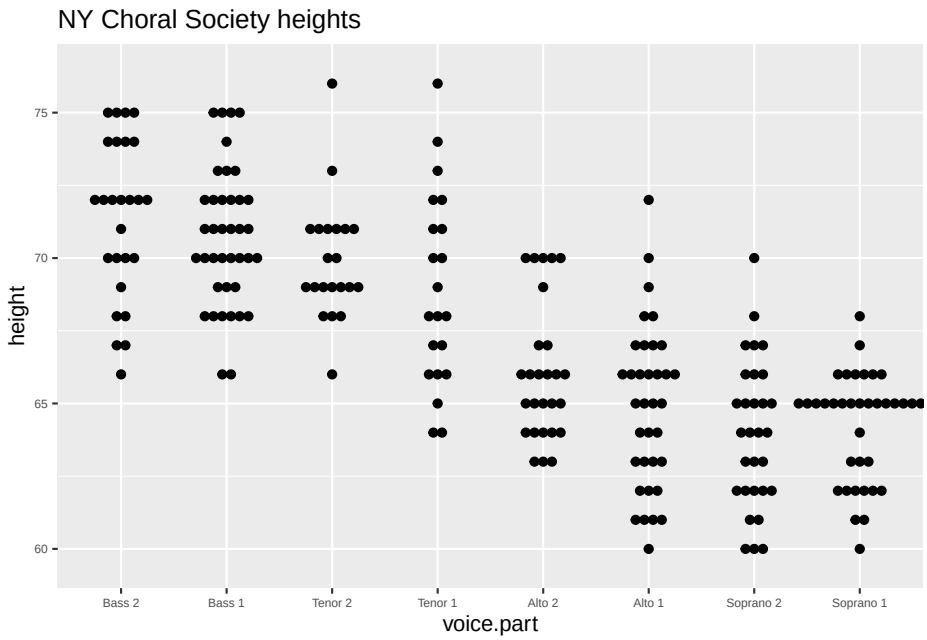
Center the dots and make them smaller:

```
gg = ggplot(singer, aes(x = voice.part, y = height)) + geom_dotplot(binaxis = "y", binwidth = 1,
gg
```



Add a title and make the labels look nicer:

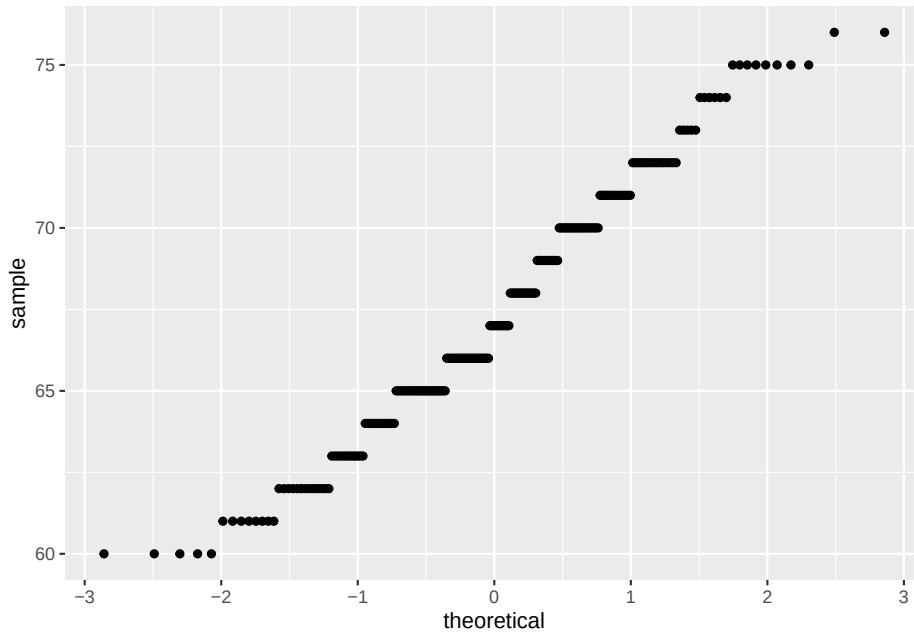
```
gg + labs(title = "NY Choral Society heights") + theme(axis.text = element_text(size =
```



2.2.4 Normal and uniform QQ plots

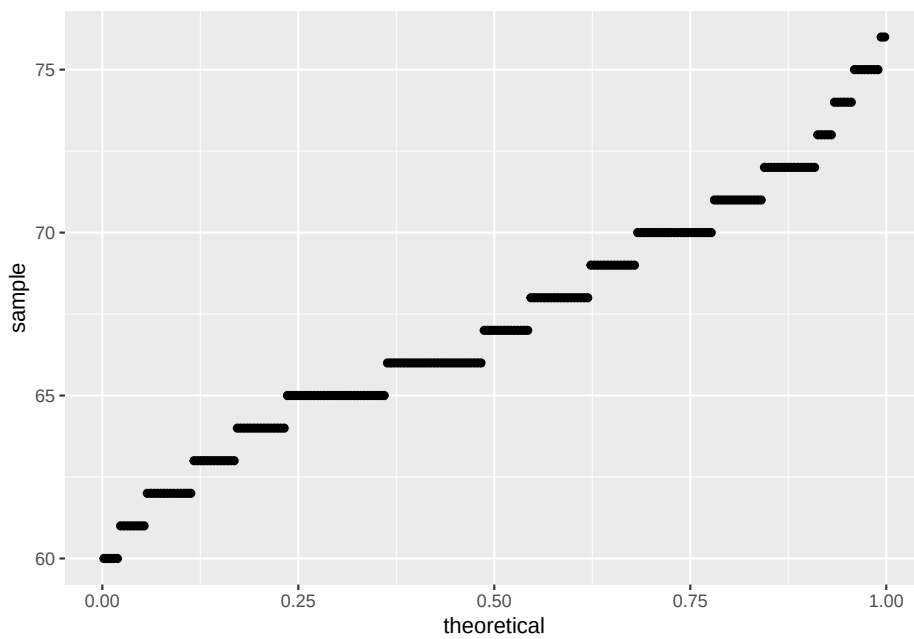
A normal QQ plot is a graph of the quantiles of a data set against the quantiles of a standard normal distribution. Here it's not as simple as setting `height` to be x or y: we need to use the heights as a *sample* from which we estimate quantiles.

```
ggplot(singer, aes(sample=height)) + stat_qq()
```



But there's no reason why the reference distribution has to be normal. In fact, it may be preferable to use a uniform as the reference.

```
ggplot(singer, aes(sample=height)) + stat_qq(distribution=qunif)
```



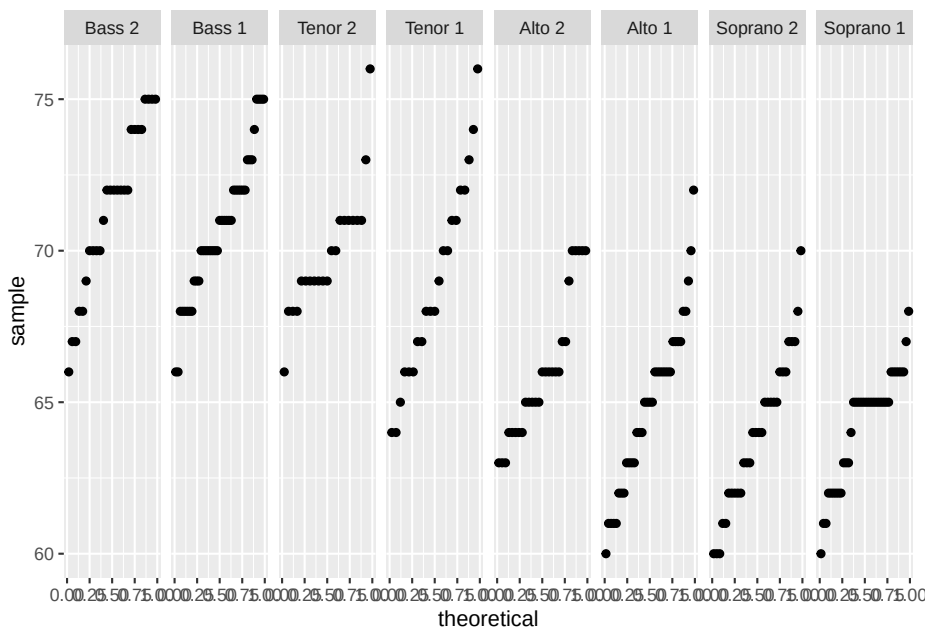
Note that this uniform QQ plot is *almost* the same as the ECDF graph with its

axes flipped.

2.2.5 Faceting

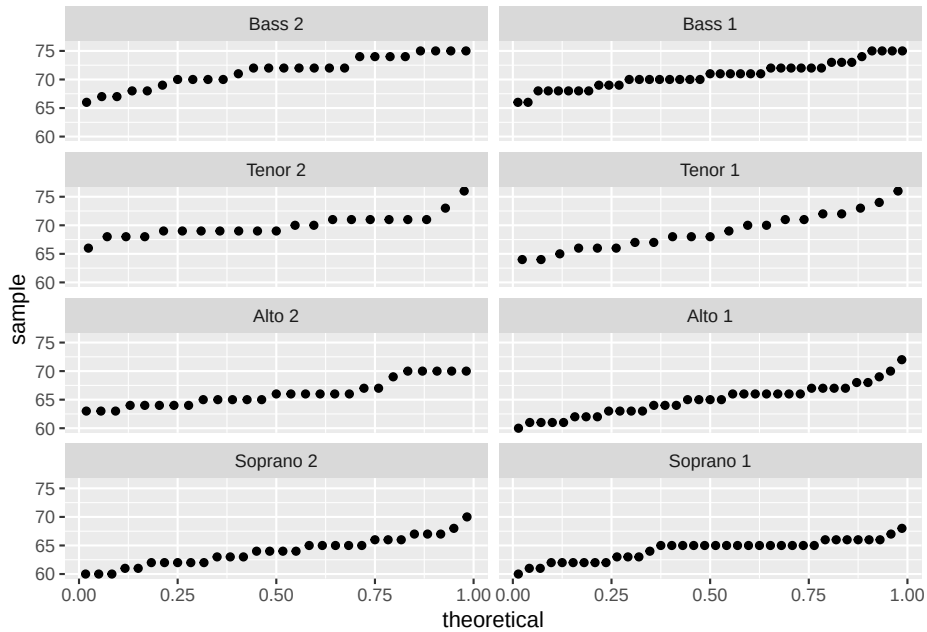
Faceting essentially means breaking up your data into subsets, then plotting those subsets separately. Suppose we want to draw a uniform QQ plot of singer heights for each voice part. We can use `facet_grid`:

```
ggplot(singer, aes(sample = height)) +
  stat_qq(distribution = qunif) +
  facet_grid(~voice.part)
```



Because we have eight different voice parts, the graph is cramped. It'll look better if we draw a grid (say 4-by-2) of graphs. `facet_wrap` lets us draw this:

```
ggplot(singer, aes(sample = height)) +
  stat_qq(distribution = qunif) +
  facet_wrap(~voice.part, ncol=2)
```



2.2.6 Creating a new variable

Suppose that instead of faceting by each voice part, we want to separate predominantly female voice parts (sopranos and altos) from predominantly male voice parts (tenors and basses.) We create a new vector for (assumed) sex using `recode()` in `dplyr` (also part of the `tidyverse`):

```
library(dplyr)
sex = recode(singer$voice.part, "Soprano 1" = "Female", "Soprano 2" = "Female",
  "Alto 1" = "Female", "Alto 2" = "Female", "Tenor 1" = "Male",
  "Tenor 2" = "Male", "Bass 1" = "Male", "Bass 2" = "Male")
```

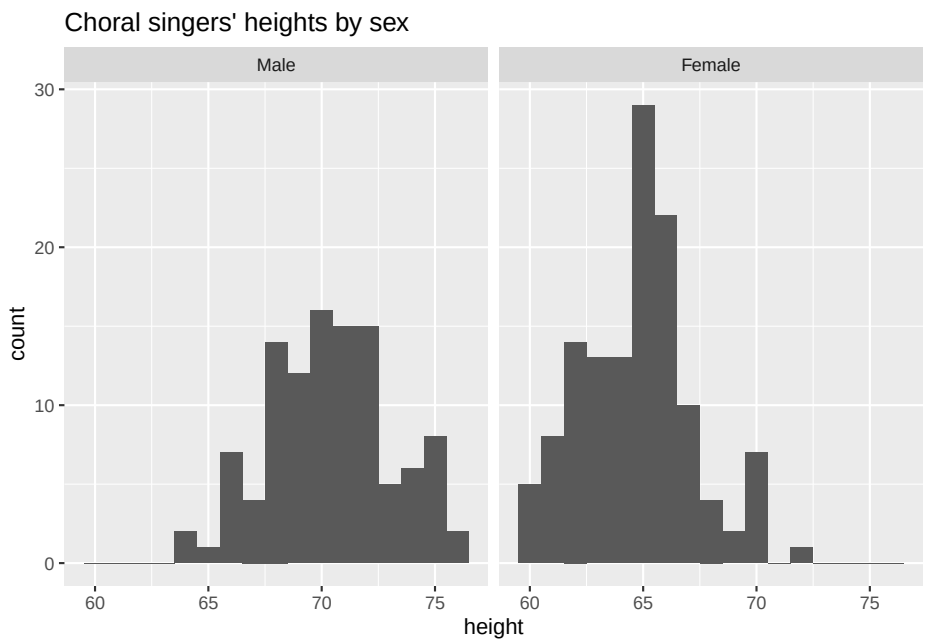
Put the vector into the data frame:

```
singer$sex = sex
summary(singer)
```

```
##      height         voice.part     sex
##  Min.   :60.0   Bass 1   :39   Male  :107
##  1st Qu.:65.0   Soprano 1:36   Female:128
##  Median :67.0   Alto 1   :35
##  Mean   :67.3   Soprano 2:30
##  3rd Qu.:70.0   Alto 2   :27
##  Max.   :76.0   Bass 2   :26
##                 (Other)  :42
```

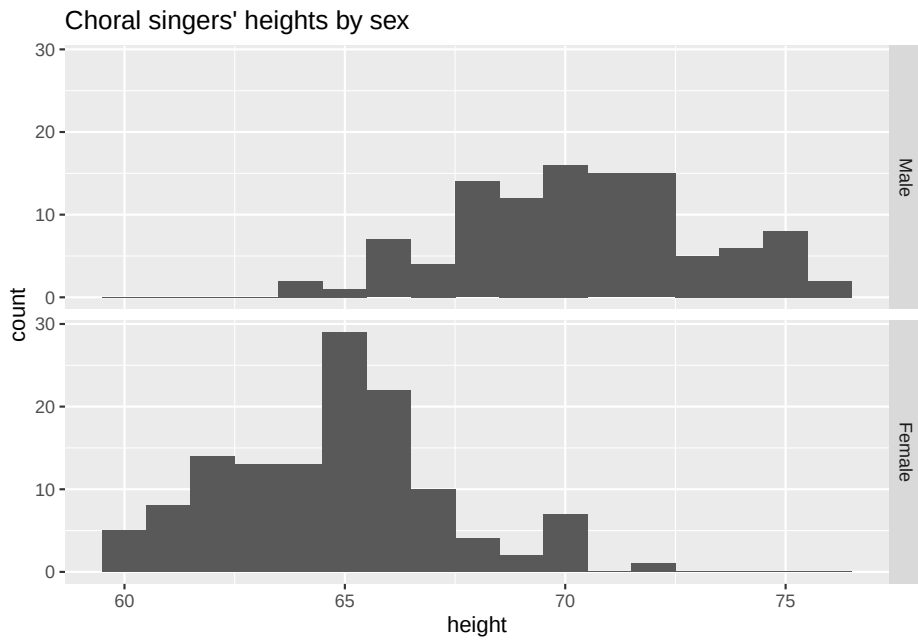
Now draw histograms faceted by the `sex` variable:

```
ggplot(singer, aes(x = height)) + geom_histogram(binwidth = 1) + facet_grid(~ sex) +
  labs(title = "Choral singers' heights by sex")
```



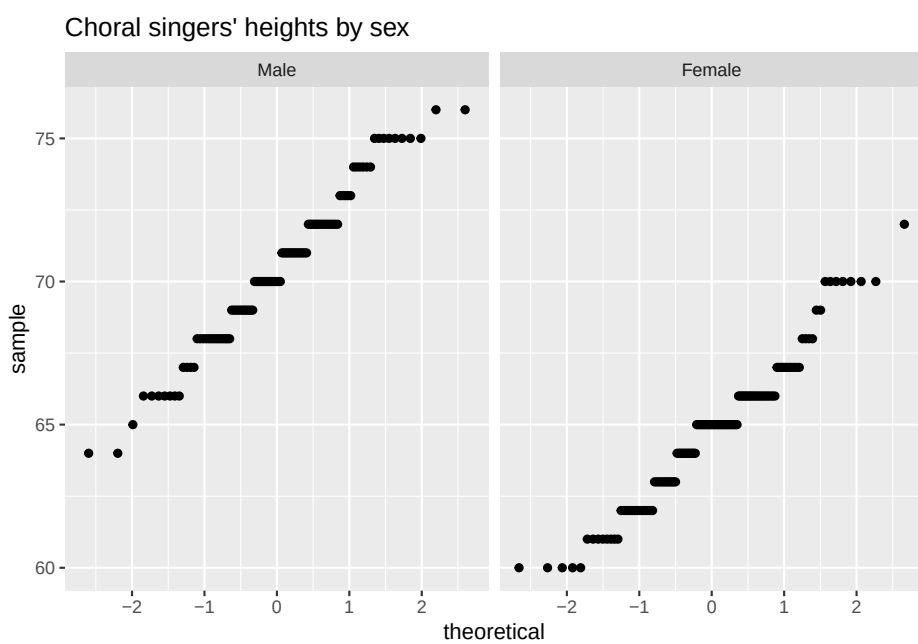
It might be better to draw one histogram above the other:

```
ggplot(singer, aes(x = height)) + geom_histogram(binwidth = 1) + facet_grid(sex ~ .) +
  labs(title = "Choral singers' heights by sex")
```



Are the distributions normal?

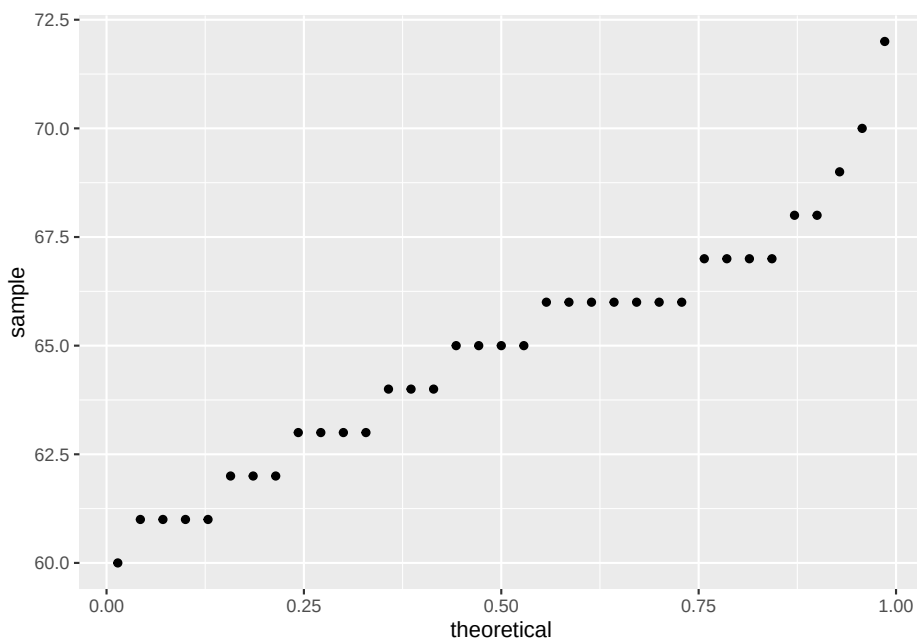
```
gg = ggplot(singer, aes(sample = height)) + stat_qq() + facet_grid(~ sex)
gg + labs(title = "Choral singers' heights by sex")
```



2.2.7 Normality revisited

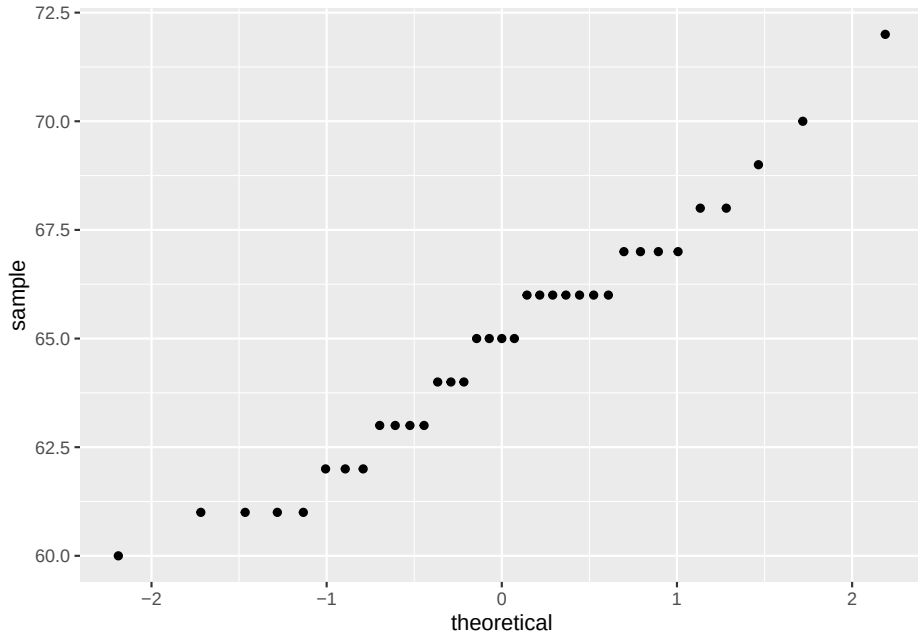
The normal QQ plot for the data as a whole doesn't look quite normal. What if we just looked at Alto 1s? First use the uniform as a reference:

```
ggplot(singer[singer$voice.part=="Alto 1",], aes(sample=height)) + stat_qq(distributio
```



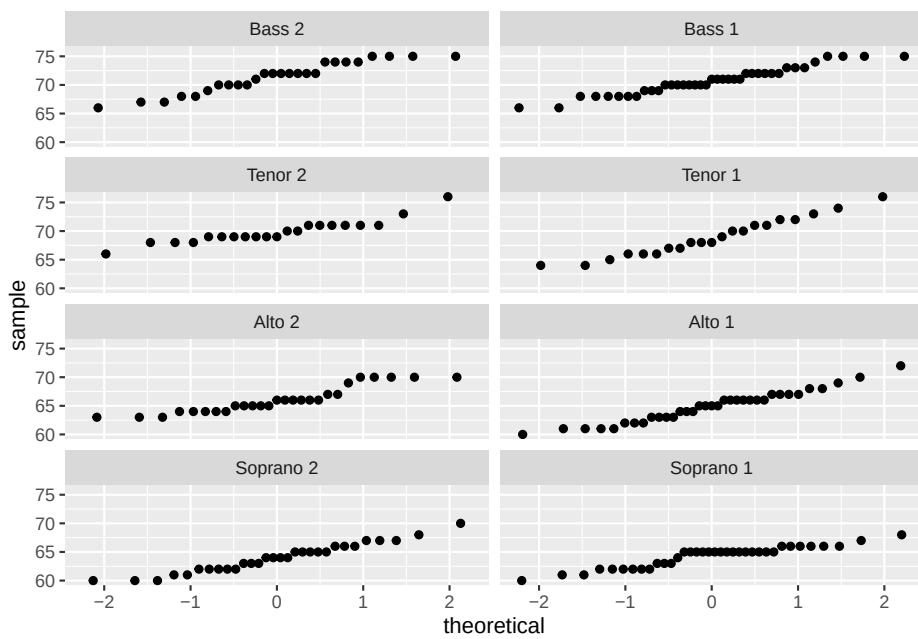
It looks like the line curves up at the end. Now let's compare to the normal:

```
ggplot(singer[singer$voice.part=="Alto 1",], aes(sample=height)) + stat_qq()
```



As before, we can facet the graphs to draw one for each voice part:

```
ggplot(singer, aes(sample = height)) +
  stat_qq() + facet_wrap(~voice.part, ncol=2)
```



The graphs generally look straightish. It seems fair to say that as far as we

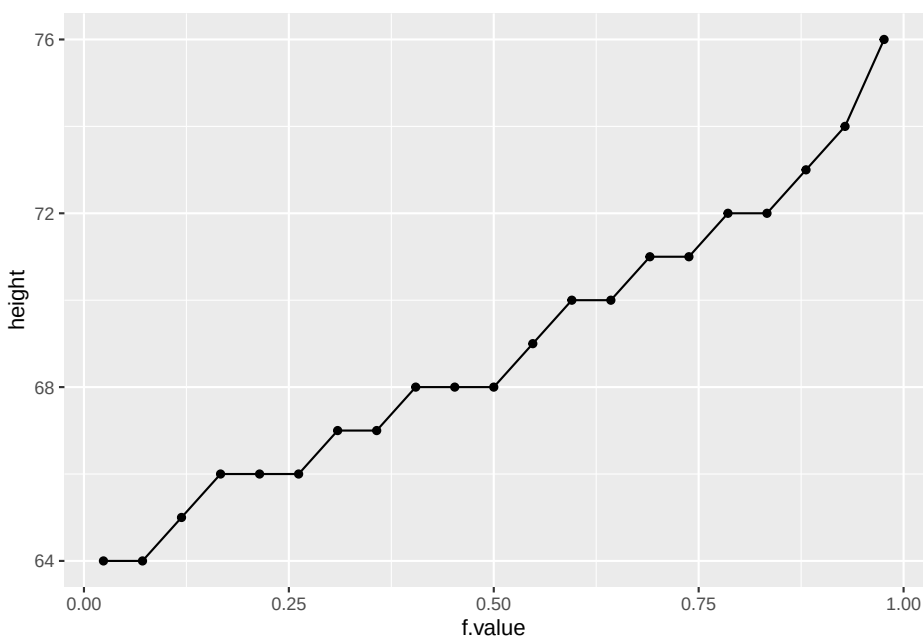
can tell, each voice part's heights are reasonably well approximated by a normal distribution. (As with most normals, this is an approximation that's most accurate in the center of the distribution: there's no evidence that the tails are normal and no theoretical reason to think they should be.)

Follow-up questions to think about: What kinds of normal distributions? What are the means? Do they all have the same standard deviation, or do the spreads differ?

2.2.8 Understanding the uniform QQ plot

Here we manually reproduce Cleveland figure 2.2. This requires first sorting the Tenor 1 heights, then finding their corresponding quantiles ("f-values".)

```
Tenor1 = sort(singer$height[singer$voice.part=="Tenor 1"])
nTenor1 = length(Tenor1)
f.value = (0.5:(nTenor1 - 0.5)) / nTenor1
Tenor1.df = data.frame(f.value, height=Tenor1)
ggplot(Tenor1.df, aes(x=f.value, y=height)) +
  geom_line() + geom_point()
```

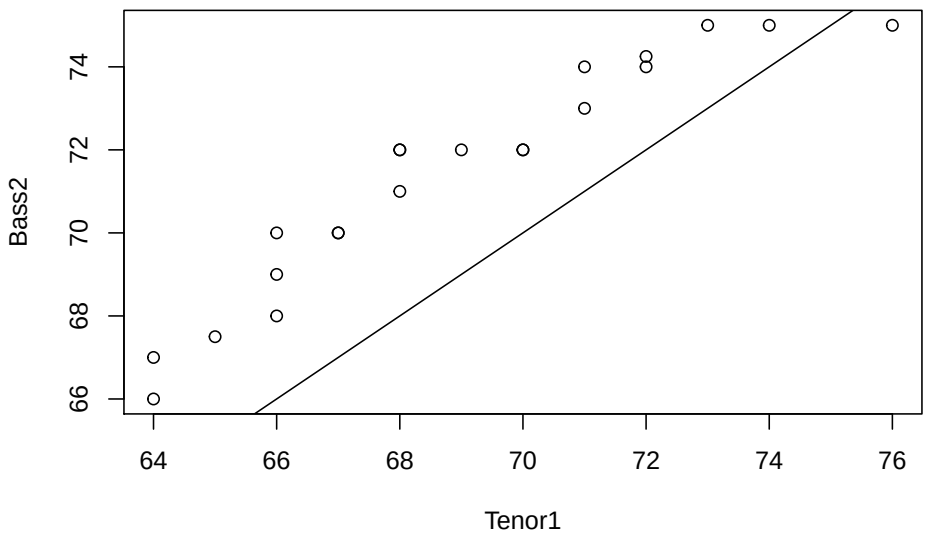


2.2.9 Two-sample QQ plots

Our QQ plots so far have plotted quantiles of one sample (at a time) against quantiles of a reference distribution. The `qqplot()` function in base R plots quantiles of one variable against quantiles of another. (Note that it does not

require the data sets to be sorted.)

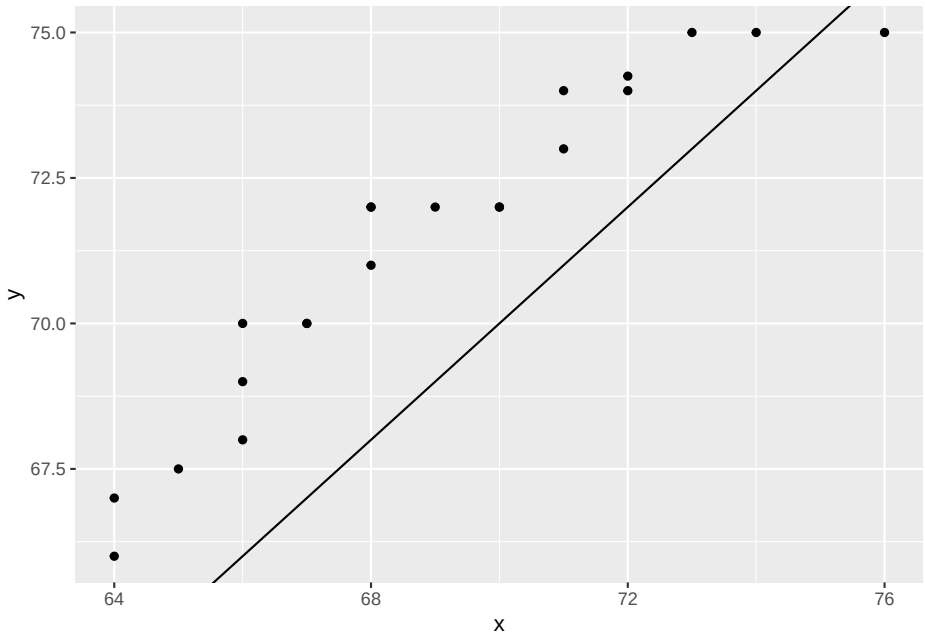
```
Tenor1 = singer$height[singer$voice.part=="Tenor 1"]
Bass2 = singer$height[singer$voice.part=="Bass 2"]
qqplot(Tenor1, Bass2)
abline(0, 1)
```



The points are well-approximated by a straight line *parallel* to the line $y = x$. This suggests the Tenor 1 and Bass 2 distributions might differ by an *additive shift**: they have similar shapes and spreads, but different centers. As the points are almost all above the line by 2–3 units, Bass 2 singers are on average 2–3 inches taller than Tenor 1 singers.

We can feed the quantiles calculated by `qqplot()` into a data frame, which we can then feed into ggplot. We'll represent the quantiles as points (`geom_point()`) then add a diagonal line (`geom_abline()`):

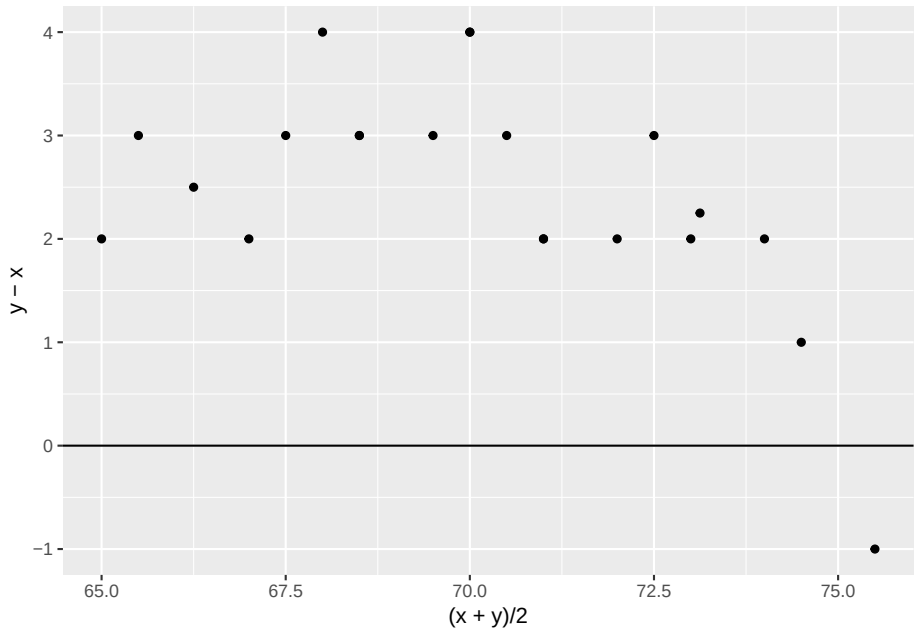
```
Tenor1 = singer$height[singer$voice.part=="Tenor 1"]
Bass2 = singer$height[singer$voice.part=="Bass 2"]
qq.df = as.data.frame(qqplot(Tenor1, Bass2, plot.it=FALSE))
ggplot(qq.df, aes(x=x, y=y)) + geom_point() +
  geom_abline()
```



2.2.10 Extra: Tukey mean-difference plots

Humans can more accurately compare to a horizontal line than a diagonal one. The *Tukey mean-difference plot* is designed to take advantage of this. On the x-axis goes the *mean* of the two quantiles (averaging Tenor 1 and Bass 2), while on the y-axis goes the difference between them (Bass 2 minus Tenor 1.)

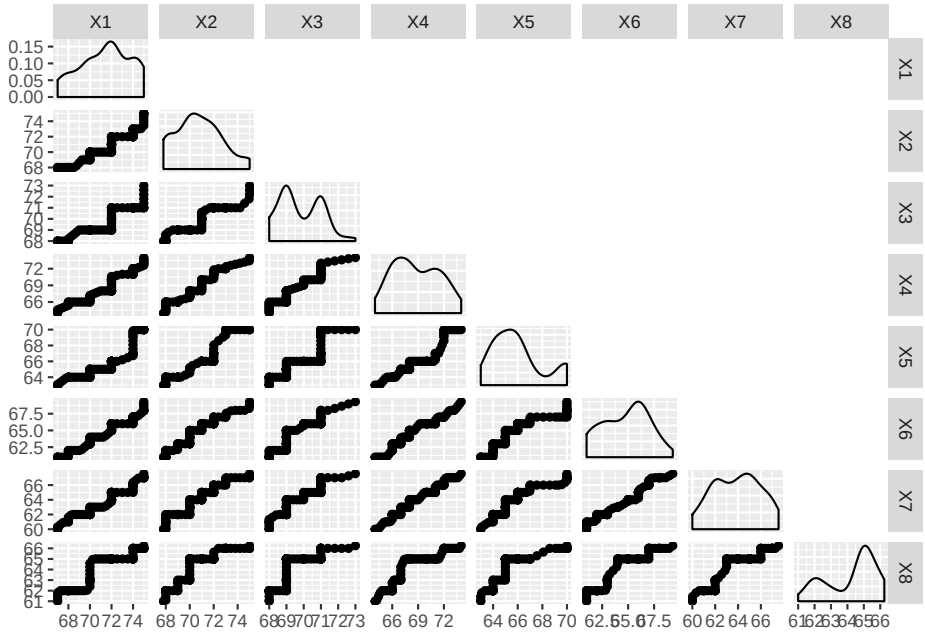
```
ggplot(qq.df, aes(x=(x+y)/2, y=y-x)) + geom_point() +
  geom_abline(slope=0)
```



2.2.11 Extra: Pairwise QQ plots

If you want to do these, a good choice is to use the `ggpairs()` function in the `GGally` library. The plots on the diagonal are density plots. The graphs are a bit too busy to be very useful in my opinion.

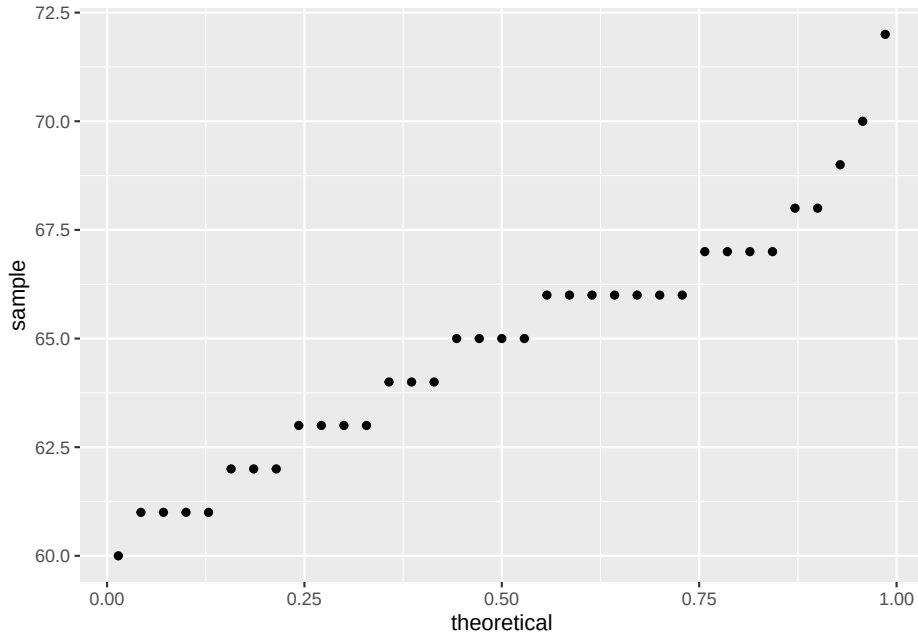
```
singer.q.rows = aggregate(height ~ voice.part, quantile, probs=seq(.05, .95, .01), data=singer)
singer.q = t(singer.q.rows[-1])
names(singer.q) = singer.q.rows[,1]
singer.q = data.frame(singer.q)
singer.panel = function(x, y){
  lines(x, y, xlim=range(singer$height), ylim=range(singer$height))
  abline(0, 1)
}
library(GGally)
ggpairs(singer.q, upper=c())
```



2.2.12 Normality revisited

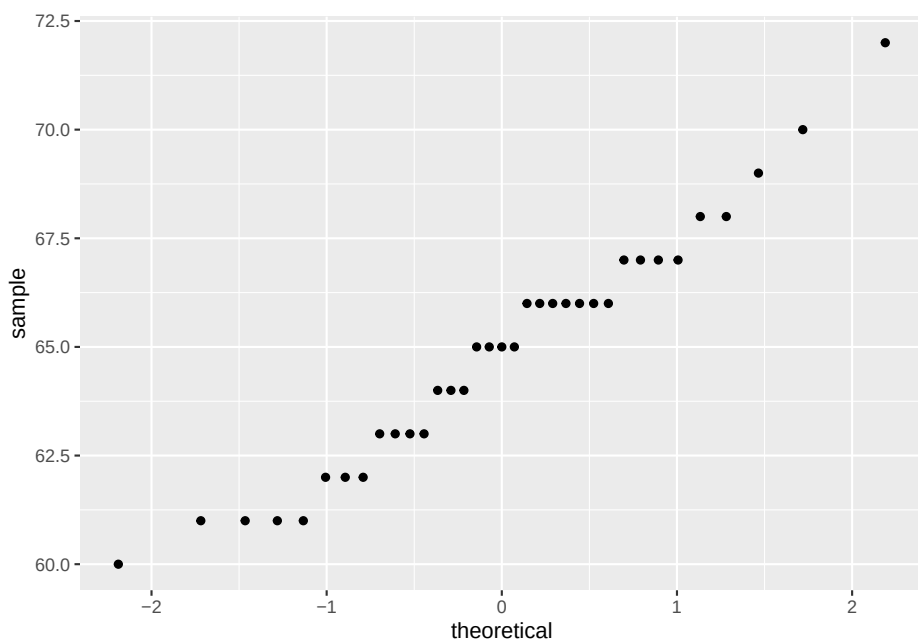
The normal QQ plot for the data as a whole doesn't look quite normal. What if we just looked at Alto 1s? First use the uniform as a reference:

```
ggplot(singer[singer$voice.part=="Alto 1",], aes(sample=height)) + stat_qq(distributio
```



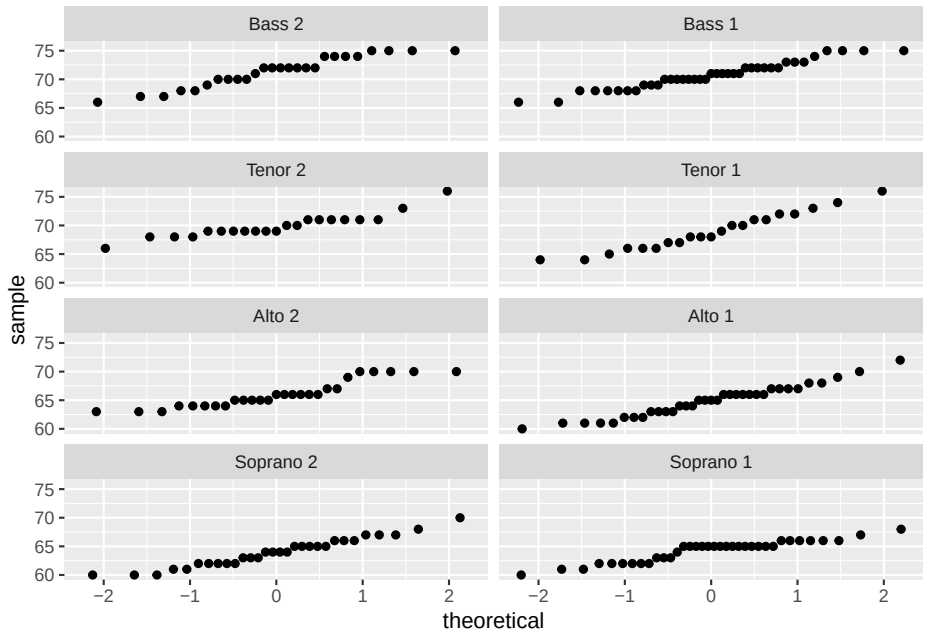
It looks like the line curves up at the end. Now let's compare to the normal:

```
ggplot(singer[singer$voice.part=="Alto 1",], aes(sample=height)) + stat_qq()
```



As before, we can facet the graphs to draw one for each voice part:

```
ggplot(singer, aes(sample = height)) +
  stat_qq() + facet_wrap(~voice.part, ncol=2)
```



The graphs generally look straightish. It seems fair to say that as far as we can tell, each voice part's heights are reasonably well approximated by a normal distribution. (As with most normals, this is an approximation that's most accurate in the center of the distribution: there's no evidence that the tails are normal and no theoretical reason to think they should be.) The next questions to answer: What kinds of normal distributions? What are the means? Do they all have the same standard deviation, or do the spreads differ?

2.3 Fits and residuals

READ: Cleveland pp. 34–41.

Load our standard libraries:

```
library(lattice)
library(ggplot2)
```

2.3.1 Aggregation and dot plots

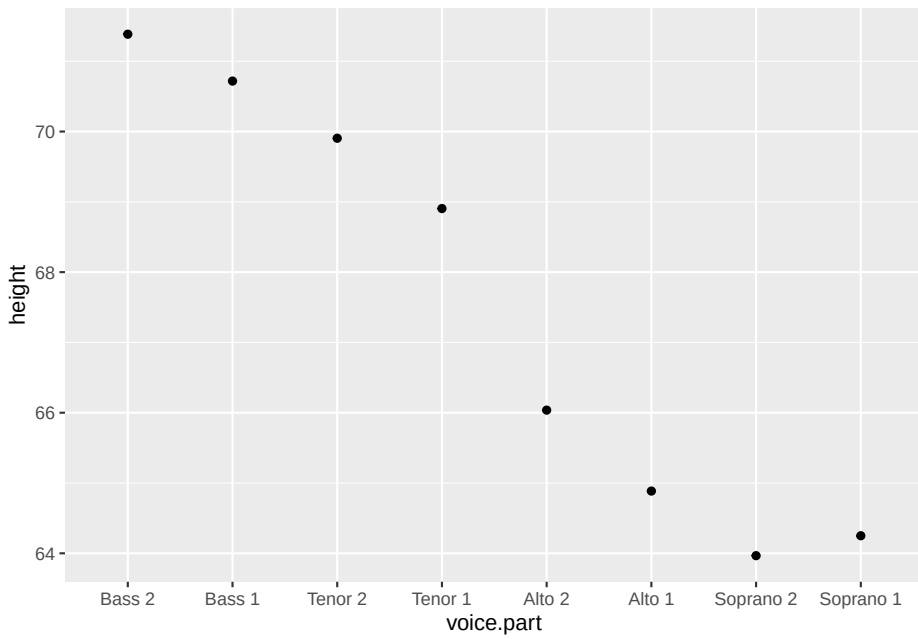
Last time we decided that the singer heights for each vocal part were well-approximated by normal distributions. Now we ask: What are the parameters of those distributions? Firstly, what are the mean heights for Soprano 1s, Soprano

2s, etc.? As you learned a long time ago, the "best" estimate of the population mean is the sample mean. We can use the `aggregate()` function to easily find the sample mean for each voice part.

```
singer.means = aggregate(height~voice.part, FUN=mean, data=singer)
```

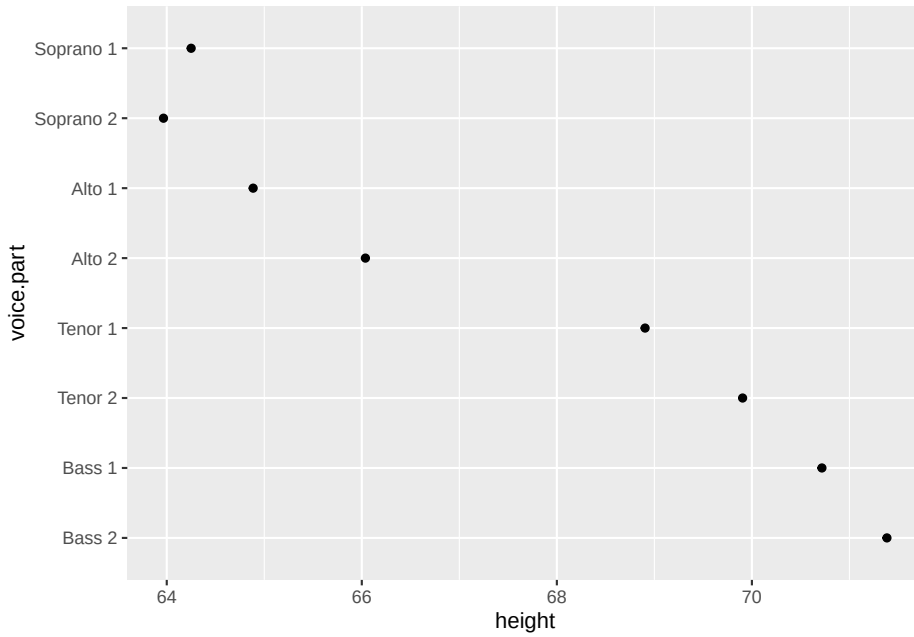
We now have one number for each of a set of categories. While this kind of data is traditionally displayed using a bar graph, a **dot plot** is arguably preferable, because they use position and not area to represent numbers. This is good because (i) humans are better at judging position than area, and (ii) for positions you don't have to start your axis at zero, while for areas you do. The `ggplot` function is `geom_point()`:

```
ggplot(singer.means, aes(x=voice.part, y=height)) +
  geom_point()
```



Note that you can also draw box plots the other way around (the choice of axes is personal preference):

```
ggplot(singer.means, aes(x=height, y=voice.part)) +
  geom_point()
```



2.3.2 Models and residuals

We now have a simple model for singer heights.

Singer height = Average height for their voice part + some error

If you've taken S431/631 or a similar regression course, you might recognize this as a special case of a linear model. If you haven't, well, it doesn't really matter much except we can use the `lm()` function to fit the model. The advantage of this is that `lm()` easily splits the data into **fitted values** and **residuals**:

Observed value = Fitted value + residual

Here the fitted values are just the sample averages for each voice part.

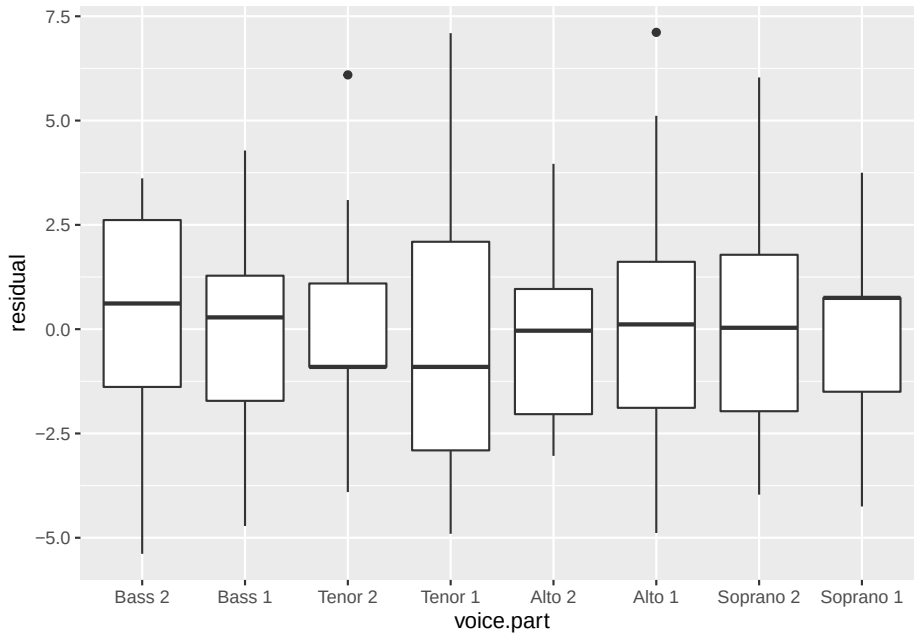
```
singer.lm = lm(height ~ voice.part, data=singer)
```

We can extract the fitted values using `fitted.values(singer.lm)` and the residuals with `residuals(singer.lm)` or `singer.lm$residuals`. For convenience, we create a data frame with two columns: the voice parts and the residuals.

```
singer.res = data.frame(voice.part=singer$voice.part, residual=residuals(singer.lm))
```

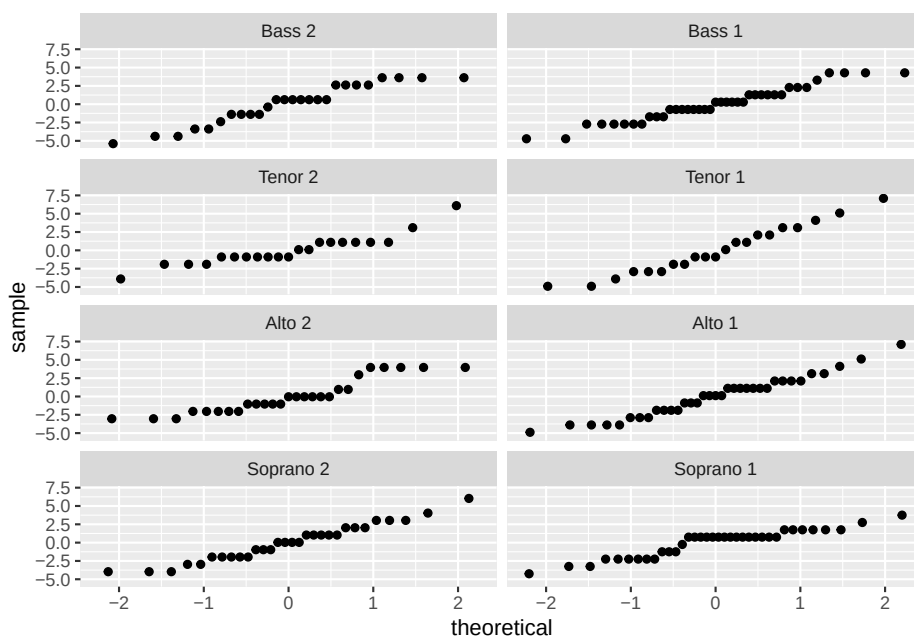
There are a few ways we can look at the residuals. Side-by-side boxplots give a broad overview:

```
ggplot(singer.res, aes(x=voice.part, y=residual)) + geom_boxplot()
```



We also want to examine normality of the residuals, broken up by voice part. We do this by faceting:

```
ggplot(singer.res, aes(sample = residual)) +
  stat_qq() + facet_wrap(~voice.part, ncol=2)
```

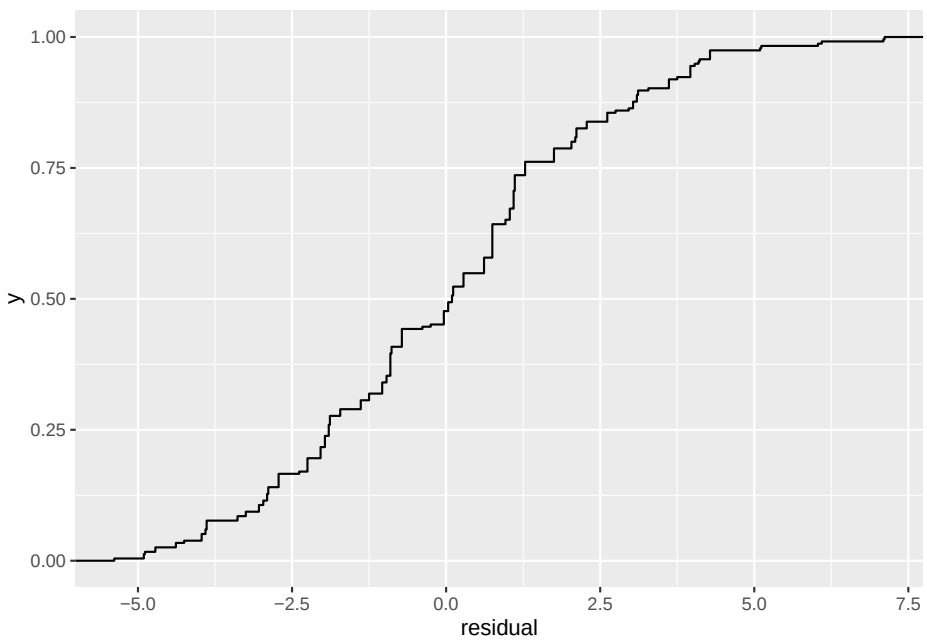


Not only do the lines look reasonably straight, the scales look similar for all eight voice parts. This suggests a model where all of the errors are normal with the *same* standard deviation. We propose this model:

Singer height = Average height for their voice part + Normal$(0, \sigma^2)$ error.

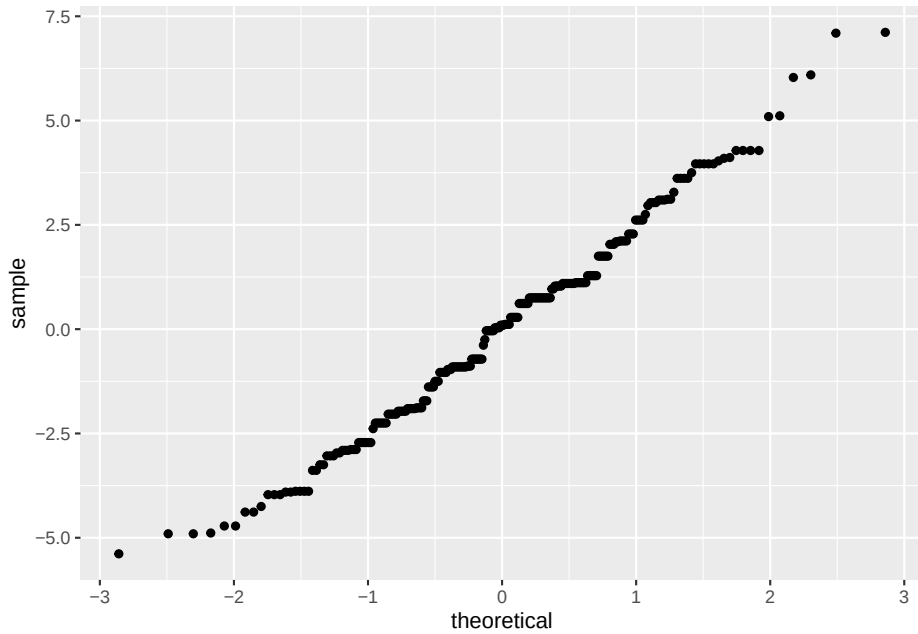
To see if this is a good fit, we **pool** the residuals and plot them. First let's try an ECDF plot:

```
ggplot(singer.res, aes(x = residual)) +
  stat_ecdf()
```



A normal distribution would give this S-shape, but so would many other distributions. To check normality directly, as usual we draw a normal QQ plot.

```
ggplot(singer.res, aes(sample = residual)) +
  stat_qq()
```



We'll also add a line with the mean of the residuals (which should be zero) as the intercept, and the SD of the residuals as the slope.

```
round(mean(singer.res$residual), 3)
```

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

```
round(sd(singer.res$residual), 3)
```

```
## [1] 2.465
```

Pedantic note: We should use an $n-8$ denominator instead of $n-1$ in the SD calculation for degrees of freedom reasons. We can get this directly from the linear model:

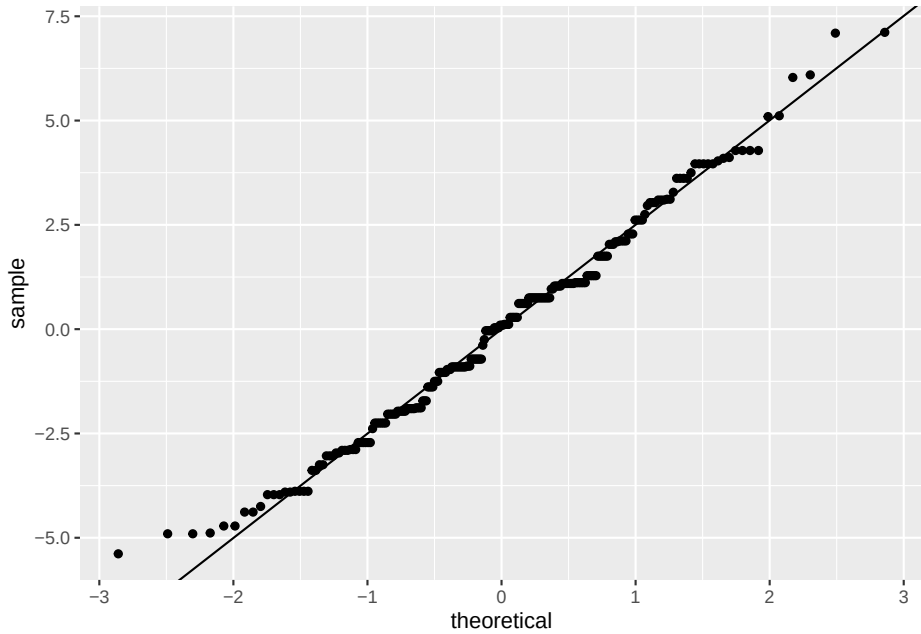
```
round(summary(singer.lm)$sigma, 3)
```

```
## [1] 2.503
```

However, the difference between this and the SD above is negligible.

Add the line:

```
ggplot(singer.res, aes(sample = residual)) +
  stat_qq() + geom_abline(intercept=0, slope=summary(singer.lm)$sigma)
```



The straight line isn't absolutely perfect, but it's doing a pretty good job. Our model is thus

Singer height = Average height for their voice part + Normal$(0, 2.5^2)$ error.

Note: While normality (or lack thereof) can be important for probabilistic prediction or (sometimes) for inferential data analysis, it's relatively unimportant for EDA. If your residuals are about normal that's nice, but as long as they're not horribly skewed they're probably not a problem.

2.3.3 Residual-fit spread

According to Cleveland, a useful thing to do (moreso when the predictor is continuous, but still useful here) is to visually compare the spread of the fitted values with the spread of the residuals. This gives an intuitive idea of how much variation is captured by the model fit and how much remains in the residuals. To do this, we want to draw two panels next to each other on the same scale: a uniform QQ plot of the fitted values (after subtracting the overall mean) and a uniform QQ plot of the residuals. We'll do this fairly manually. The first goal is to get the fitted values and the residuals, and separately sort them:

```
singer.fitted = sort(fitted.values(singer.lm)) - mean(fitted.values(singer.lm))
singer.residuals = sort(residuals(singer.lm))
```

Now we calculate the f-values (there's only one set, since there's the same number of fitted values and residuals) and stick everything into a data frame.

```
n = length(singer.residuals)
f.value = (0.5:(n - 0.5)) / n
singer.fit = data.frame(f.value, Fitted=singer.fitted, Residuals=singer.residuals)
```

We now have a problem: We have a data frame with 235 observations in three columns:

f.value, Fitted, Residuals

But to use faceting in `ggplot`, we'd like a data frame with (2×235) observations in these three columns:

f.value, type, value

where "type" is a categorical variable (either "Fitted" or "Residual") and "value" is the numerical value (either the fitted value or the residual.)

We could easily manually hammer the data into the right form, but instead we'll learn a new tool to do this.

2.3.4 gather(): A lifesaver

We'll use `gather()` in Hadley Wickham's `tidyr` library, which has saved the lives of countless frustrated R users in recent years. (You may need to install `tidyr` if you don't already have it.) `gather()`, as the names perhaps suggests, will take several variables and gathers them together, so you have more observations but fewer columns.

```
library(tidyr)
singer.fit.long = singer.fit %>% gather(type, value, Fitted:Residuals)
```

What did we just do? Let's illustrate with a simple example. Suppose you're studying the votes won by Clinton and Trump in 2016 in a few Indiana counties. You enter the data:

```
Counties = c("Hamilton","Marion","Monroe","Tippecanoe","Vermillion")
Clinton = c(2819, 212676, 34183, 27207, 2081)
Trump = c(8530, 130228, 20527, 30711, 4511)
Indiana = data.frame(Counties, Clinton, Trump)
```

Let's see what this looks like:

```
print(Indiana)
```

```
##     Counties Clinton  Trump
## 1   Hamilton    2819   8530
## 2     Marion  212676 130228
## 3     Monroe   34183  20527
## 4 Tippecanoe   27207  30711
## 5 Vermillion    2081   4511
```

Then you decide you want to do faceting, so you want to change the data to "long" form. That is, instead of three columns of length 5, you want 3 columns of length 10. Use `gather()`:

```
Indiana.long = Indiana %>% gather(Candidate, Votes, Clinton:Trump)
```

Let's see what we got:

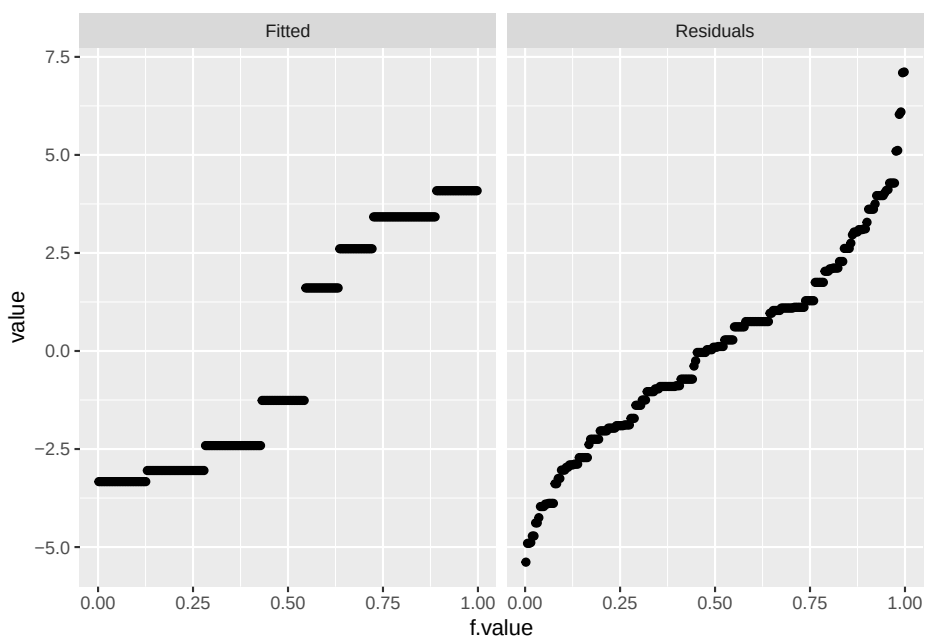
```
print(Indiana.long)
```

```
##      Counties Candidate  Votes
## 1    Hamilton   Clinton   2819
## 2      Marion   Clinton 212676
## 3      Monroe   Clinton  34183
## 4  Tippecanoe   Clinton  27207
## 5  Vermillion   Clinton   2081
## 6    Hamilton     Trump   8530
## 7      Marion     Trump 130228
## 8      Monroe     Trump  20527
## 9  Tippecanoe     Trump  30711
## 10 Vermillion     Trump   4511
```

Now you can do faceting and all kinds of crazy stuff.

Let's go back to the singer height residuals and draw our residual-fit plot.

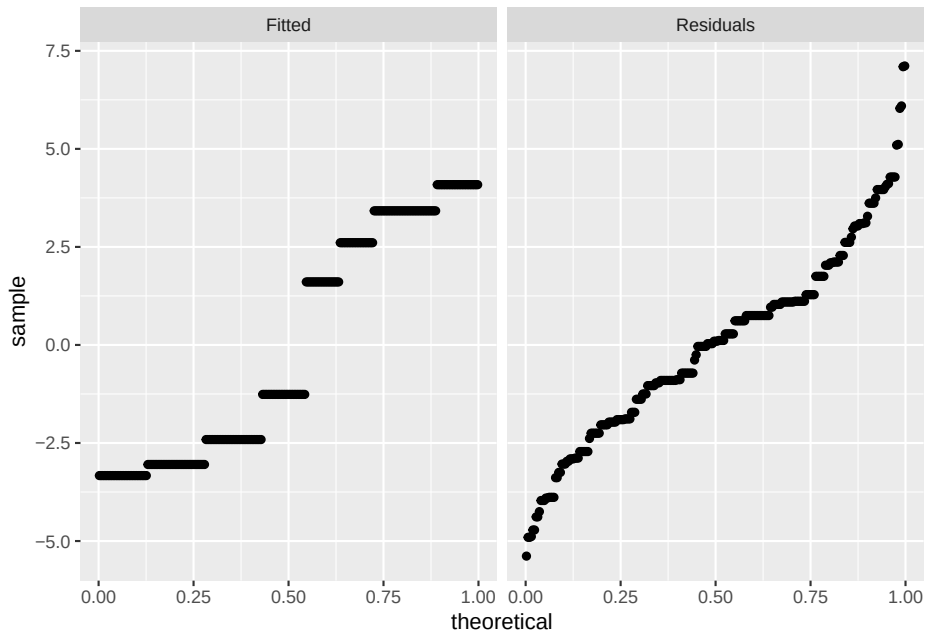
```
ggplot(singer.fit.long, aes(x=f.value, y=value)) +
  geom_point() + facet_wrap(~type)
```



The spread of the fitted values is broadly similar to the spread of the residuals. The model fit accounts for a decent chunk of the variation, but a decent chunk remains in the residuals.

Now, we didn't actually have to calculate the f-values ourselves. We can use the `stat_qq()` function, but tell R we want to use the quantiles of a uniform distribution instead of a normal:

```
ggplot(singer.fit.long, aes(sample = value)) + stat_qq(distribution = "qunif") + facet_grid(~type
```



We went to a lot of effort to draw the above graph. Personally, I'd much rather quantify the model fit by using R^2:

```
var(singer.fitted) / var(singer$height)
```

```
## [1] 0.5846243
```

Rather than use vauge terms like "variance explained" (or "coefficient of determination," whatever that means,) I'd say "the model captures 59% of the variance of the data." The remaining 41% is in the residuals.

2.3.5 Interlude: Tibbles

For large or complex data sets, *tibbles* are a minor improvement on data frames:

- They print more elegantly;
- They subset more predictably. e.g. When you use square brackets to subset a tibble, you always get another tibble. (With data frames you might get

a vector or a data frame.)

Generally though you can use them in all the same ways as you use data frames.

Let's read in a .csv file on lengths of Billboard Hot 100 song s in 2000:

```
billboard.raw = read.csv("https://github.com/hadley/tidy-data/raw/master/data/billboar
```

To turn this into a tibble, we use the `as_tibble()` function in the `dplyr` library.

```
library(dplyr)
billboard = as_tibble(billboard.raw)
```

The object `billboard` displays reasonably elegantly:

```
billboard
```

```
## # A tibble: 317 x 83
## year artist.inverted track time genre date.entered
date.peaked x1st.week
## <int> <chr> <chr> <chr> <chr> <chr> <chr> <int>
## 1 2000 Destiny's Child Inde~ 3:38 Rock 2000-09-23 2000-11-18
78
## 2 2000 Santana Mari~ 4:18 Rock 2000-02-12 2000-04-08 15
## 3 2000 Savage Garden I Kn~ 4:07 Rock 1999-10-23 2000-01-29 71
## 4 2000 Madonna Music 3:45 Rock 2000-08-12 2000-09-16 41
## 5 2000 Aguilera, Chri~ Come~ 3:38 Rock 2000-08-05 2000-10-14
57
## 6 2000 Janet Does~ 4:17 Rock 2000-06-17 2000-08-26 59
## 7 2000 Destiny's Child Say ~ 4:31 Rock 1999-12-25 2000-03-18
83
## 8 2000 Iglesias, Enri~ Be W~ 3:36 Latin 2000-04-01 2000-06-24
63
## 9 2000 Sisqo Inco~ 3:52 Rock 2000-06-24 2000-08-12 77
## 10 2000 Lonestar Amaz~ 4:25 Coun~ 1999-06-05 2000-03-04 81
## # ... with 307 more rows, and 75 more variables: x2nd.week
<int>,
## # x3rd.week <int>, x4th.week <int>, x5th.week <int>,
x6th.week <int>,
## # x7th.week <int>, x8th.week <int>, x9th.week <int>,
x10th.week <int>,
## # x11th.week <int>, x12th.week <int>, x13th.week <int>,
x14th.week <int>,
## # x15th.week <int>, x16th.week <int>, x17th.week <int>,
x18th.week <int>,
## # x19th.week <int>, x20th.week <int>, x21st.week <int>,
x22nd.week <int>,
## # x23rd.week <int>, x24th.week <int>, x25th.week <int>,
x26th.week <int>,
```

```
## # x27th.week <int>, x28th.week <int>, x29th.week <int>,
x30th.week <int>,
## # x31st.week <int>, x32nd.week <int>, x33rd.week <int>,
x34th.week <int>,
## # x35th.week <int>, x36th.week <int>, x37th.week <int>,
x38th.week <int>,
## # x39th.week <int>, x40th.week <int>, x41st.week <int>,
x42nd.week <int>,
## # x43rd.week <int>, x44th.week <int>, x45th.week <int>,
x46th.week <int>,
## # x47th.week <int>, x48th.week <int>, x49th.week <int>,
x50th.week <int>,
## # x51st.week <int>, x52nd.week <int>, x53rd.week <int>,
x54th.week <int>,
## # x55th.week <int>, x56th.week <int>, x57th.week <int>,
x58th.week <int>,
## # x59th.week <int>, x60th.week <int>, x61st.week <int>,
x62nd.week <int>,
## # x63rd.week <int>, x64th.week <int>, x65th.week <int>,
x66th.week <lgl>,
## # x67th.week <lgl>, x68th.week <lgl>, x69th.week <lgl>,
x70th.week <lgl>,
## # x71st.week <lgl>, x72nd.week <lgl>, x73rd.week <lgl>,
x74th.week <lgl>,
## # x75th.week <lgl>, x76th.week <lgl>
```

2.3.6 How long are songs?

How long were the songs on the 2000 Hot 100? We see from the tibble display that the `time` column of our tibble is a character variable instead of numeric. It'll take some work to fix this.

The immediate problem is the colon. All we want are the numbers before (the minutes) and after (the seconds) the colon for each entry. The function `strsplit` splits a character vector into strings:

```
billboard.time = strsplit(billboard$time, ":")
```

`billboard.time` is now a list. Now we need to:

- "unlist" `billboard.time` by making it into a matrix;
- extract the two columns of the matrix (minutes and seconds) and make them numeric;
- calculate the length of each song in seconds;
- change this to minutes (because minutes are more intuitive than seconds);
- replace the `time` column of `billboard.time` with this new numeric vector.

```
billboard.time = matrix(unlist(billboard.time),
  byrow=T, ncol=2)
billboard.mins = as.numeric(billboard.time[,1])
billboard.secs = as.numeric(billboard.time[,2])
billboard.time = billboard.mins * 60 + billboard.secs
billboard$time = billboard.time / 60
```

Depending on what we want to do, it may be useful to have the data in long form as well as in wide form. We use `gather()`:

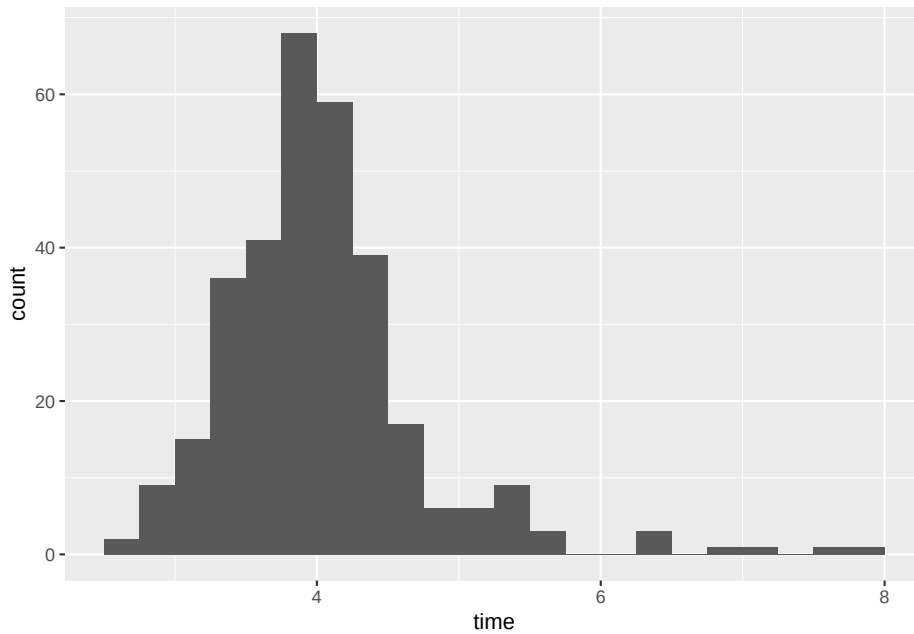
```
billboard.long <- billboard %>% gather(week, rank, x1st.week:x76th.week, na.rm = TRUE)
billboard.long
```

```
## # A tibble: 5,307 x 9
## year artist.inverted track time genre date.entered
date.peaked week rank
## <int> <chr> <chr> <dbl> <chr> <chr> <chr> <chr> <int>
## 1 2000 Destiny's Child Inde~ 3.63 Rock 2000-09-23 2000-11-18
x1st~ 78
## 2 2000 Santana Mari~ 4.3 Rock 2000-02-12 2000-04-08 x1st~ 15
## 3 2000 Savage Garden I Kn~ 4.12 Rock 1999-10-23 2000-01-29
x1st~ 71
## 4 2000 Madonna Music 3.75 Rock 2000-08-12 2000-09-16 x1st~ 41
## 5 2000 Aguilera, Chris~ Come~ 3.63 Rock 2000-08-05 2000-10-14
x1st~ 57
## 6 2000 Janet Does~ 4.28 Rock 2000-06-17 2000-08-26 x1st~ 59
## 7 2000 Destiny's Child Say ~ 4.52 Rock 1999-12-25 2000-03-18
x1st~ 83
## 8 2000 Iglesias, Enriq~ Be W~ 3.6 Latin 2000-04-01 2000-06-24
x1st~ 63
## 9 2000 Sisqo Inco~ 3.87 Rock 2000-06-24 2000-08-12 x1st~ 77
## 10 2000 Lonestar Amaz~ 4.42 Coun~ 1999-06-05 2000-03-04 x1st~
81
## # ... with 5,297 more rows
```

Here `x1st.week:x76th.week` gathers all the columns from `x1st.week` to `x76th.week` and turns them into separate observations. Thus "Independent Women Part I" accounts for one observation in `billboard` but 28 observations in `billboard.long`, because it was on the charts for 28 weeks.

Let's first go back to the wide form data. Using songs as the unit, here's a histogram of song lengths:

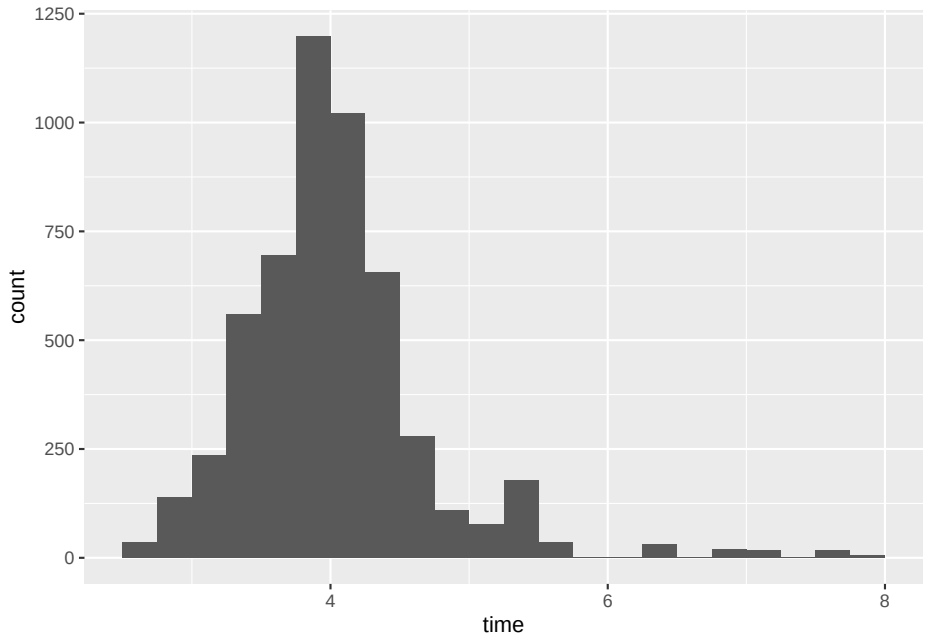
```
ggplot(billboard, aes(x=time)) + geom_histogram(breaks=seq(2.5, 8, 0.25))
```



The majority of Hot 100 songs (maybe two-thirds) are between 3:15 and 4:30.

Compare this to a histogram using song-weeks as the unit:

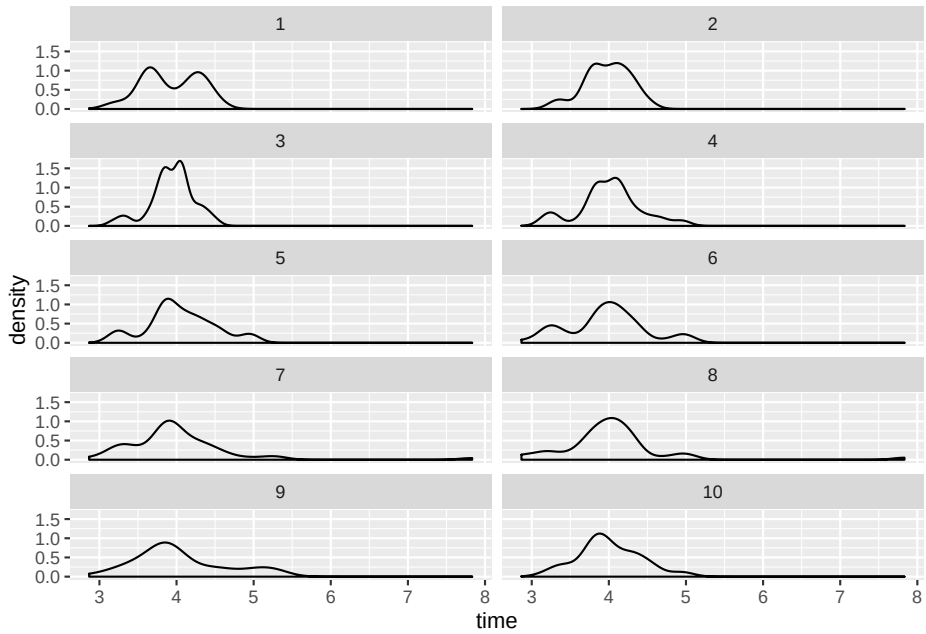
```
ggplot(billboard.long, aes(x=time)) + geom_histogram(breaks=seq(2.5, 8, 0.25))
```



The y-axis scale has changed (since there are more observations.) However, the shape of the histogram is pretty much the same, suggesting that weighting by number of weeks on the chart doesn't make much difference.

Let's facet by chart position. To avoid drawing 100 graphs, we use `subset()` to only consider the top 10 chart positions:

```
ggplot(subset(billboard.long, rank <= 10), aes(x=time)) + geom_density() + facet_wrap(
```



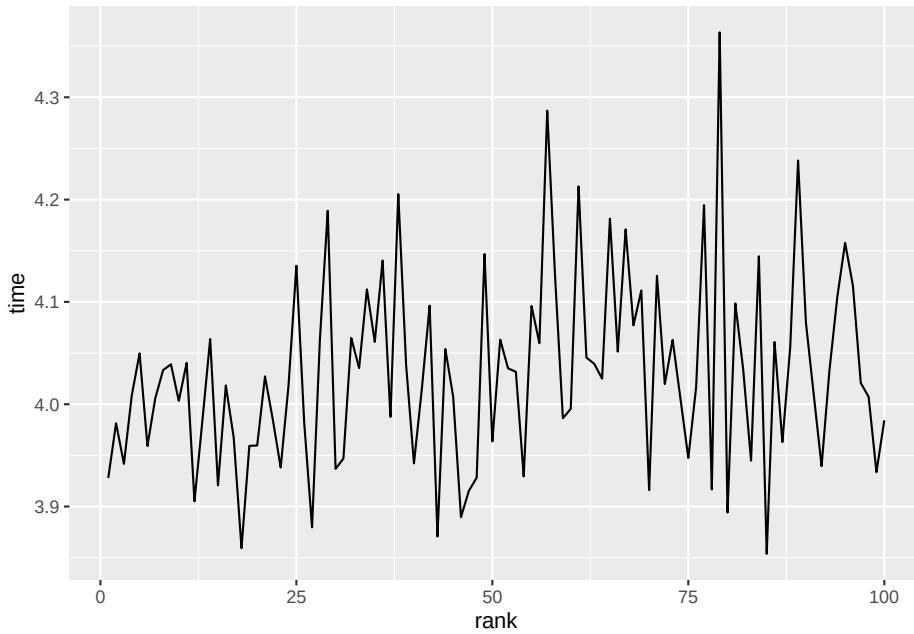
I drew density plots instead of histograms because of the small samples. It's a bit hard to tell here whether the differences are systematic or noise.

Finally, let's find mean song length by chart position.

```
time.means = aggregate(time~rank, FUN=mean, data=billboard.long)
```

Rank is numeric, so it makes sense to draw a **line graph** using `geom_line()`:

```
ggplot(time.means, aes(x=rank, y=time)) +
  geom_line()
```



There's still a lot of noise. (Later on we'll learn a bit about *smoothing*, which would help a lot here.) The one thing that seems apparent is the high ranks are noisier than low ranks. But that's just because the low ranks often have the same song appearing repeatedly (songs often stay in the top ten for ages, whilst they rarely stay in the nineties for more than a week.)

2.4 Transformations

READ: Cleveland pp. 42–67.

Load `ggplot2`:

```
library(ggplot2)
```

We'll also use an R workspace prepared by Cleveland to use in conjunction with the book. Among other things, this contains the data sets used in the book.

```
load("lattice.RData")
```

2.4.1 Log transformation

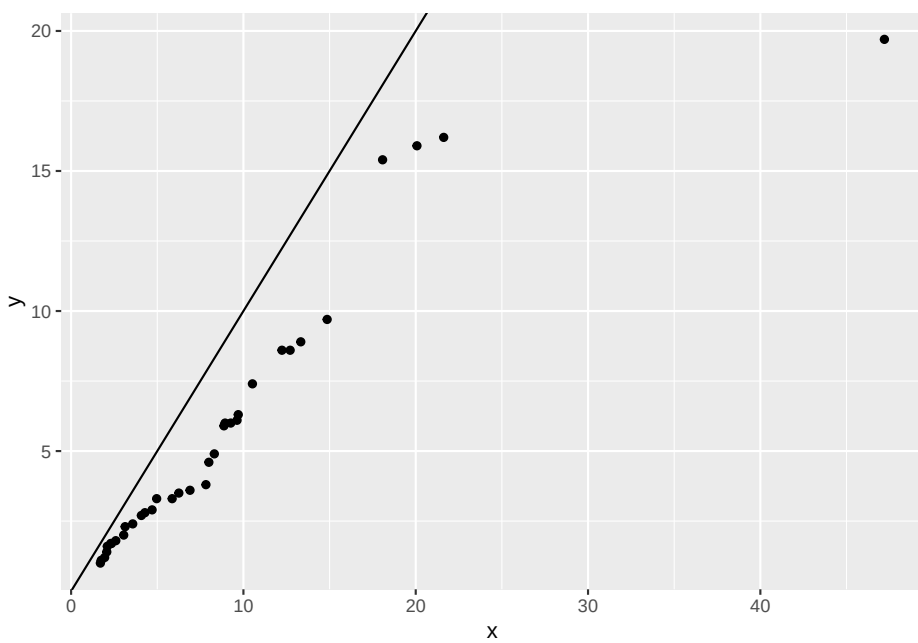
In an intro course like S320/520, you learned to use the most common transformation, the log. The main reason we gave was that it often made positive data more normal. But there's another and perhaps more fundamental reason: It often leads to differences between samples that we can interpret as a *multiplicative shift*. Some statisticians will go as far as to recommend log transforming

positive data by default, though by the end of Cleveland's chapter 2, we'll see an example where that backfires.

We'll use the stereogram fusion time data in `fusion.time`, which contains the group variable `nv.vv` (where "NV" means no visual information and "VV" means visual information) and the quantitative variable `time` (a positive number.)

We first draw a two-sample QQ plot of the data.

```
time = fusion.time$time
nv.vv = fusion.time$nv.vv
NV.times = sort(time[nv.vv=="NV"])
VV.times = sort(time[nv.vv=="VV"])
NV.VV.qq = as.data.frame(qqplot(NV.times, VV.times, plot.it=FALSE))
ggplot(NV.VV.qq, aes(x, y)) + geom_point() +
  geom_abline()
```

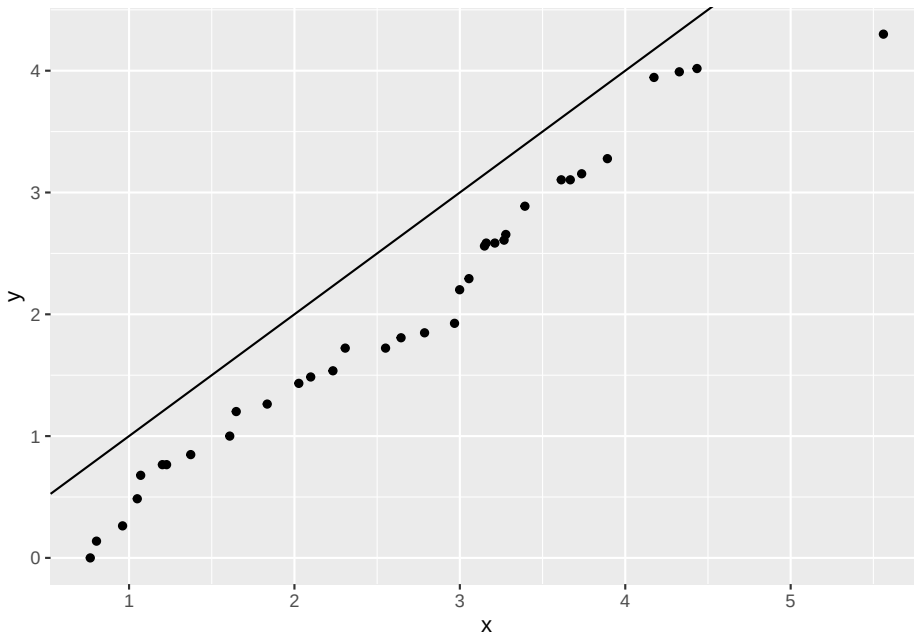


The line $y = x$ doesn't fit the plot. It seems like a line through the origin with a smaller slope – which would be equivalent to a multiplicative effect – would be a good fit, but it's hard to visually work out where the line should go.

Let's try a log transformation. In contrast to confirmatory data analysis, where natural logs are standard, in EDA it's usually much easier to understand what's going on if we use base 2 or base 10 logs.

```
NV.times.log = sort(log2(time[nv.vv=="NV"]))
VV.times.log = sort(log2(time[nv.vv=="VV"]))
```

```
NV.VV.qq.log = as.data.frame(qqplot(NV.times.log, VV.times.log, plot.it=FALSE))
ggplot(NV.VV.qq.log, aes(x, y)) + geom_point() +
  geom_abline()
```

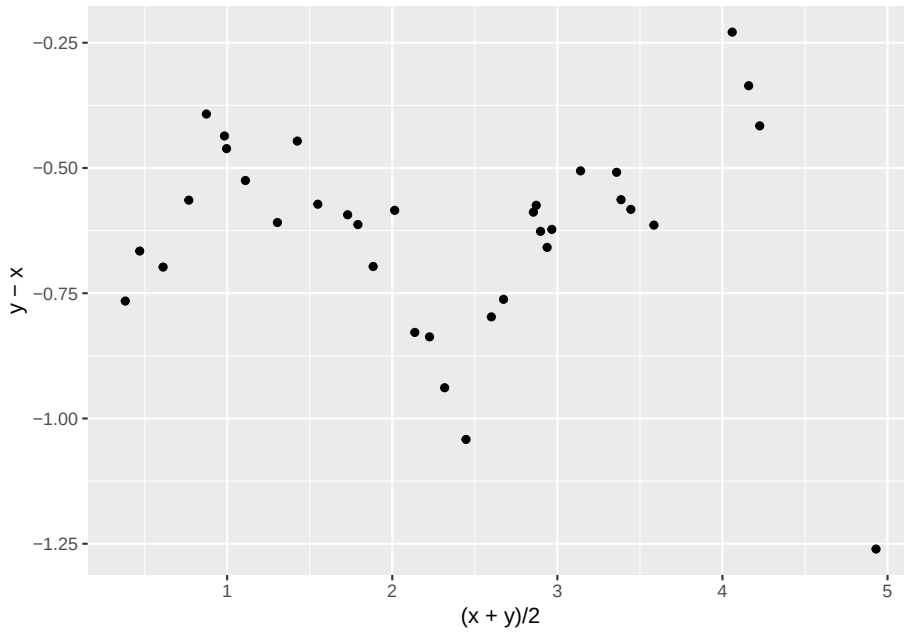


We can see the data is well-described by a straight line below $y = x$, indicating a multiplicative shift – roughly speaking, the distribution of the VV times is the NV distributions multiplied by some constant (less than 1.)

2.4.2 Tukey mean-difference plot

While the shift is pretty clear here, in messier cases it may be difficult to compare to the diagonal line. A *Tukey mean-difference* plot has the advantage that we can compare to the x-axis instead. It simply plots the difference $y - x$ against the mean $(x + y)/2$:

```
ggplot(NV.VV.qq.log, aes((x+y)/2, y-x)) + geom_point()
```



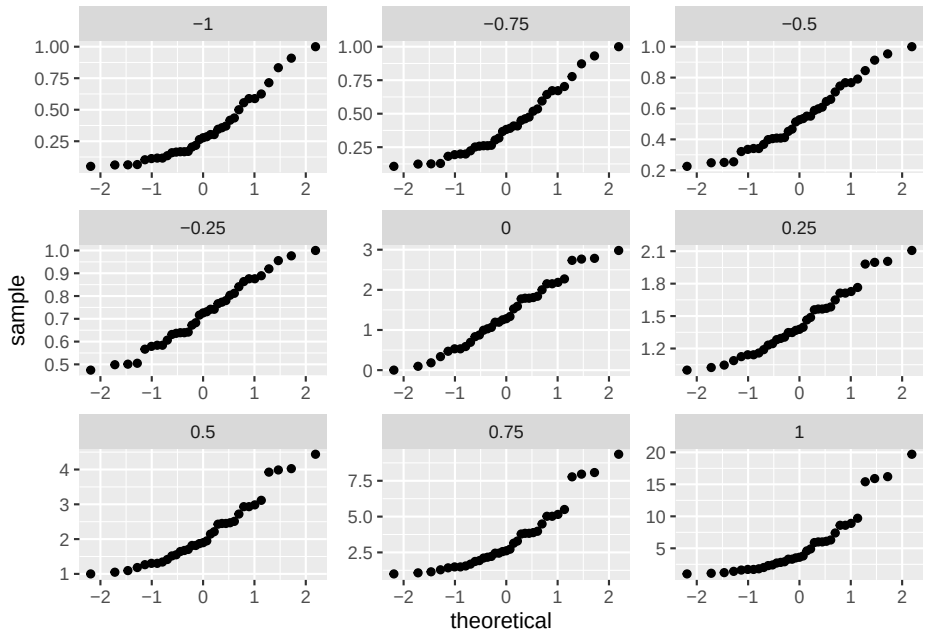
The points are scattered somewher around -0.75 to -0.5 on the y-axis. That means the multiplicative constant is around $2^{-0.75}$ to $2^{-0.5}$, or 0.6 to 0.7. Our best guess is that the visual information decreases fusion time by 30–40%.

2.4.3 Power transformation

The log transformation doesn't always work – for a start, you can't log zero or a negative number. **Power transformations** allow a wider range of options. Define a power transformation with parameter τ to be x^τ, and let the special case where $\tau = 0$ be the log transformation. Where practical, we'll hugely prefer $\tau = 1$ (no transformation), $\tau = 0$, or possibly $\tau = -1$ (inverse transformation) because they're much easier to interpret.

We can try out a variety of values of τ on the fusion time data, and see what gives us a distribution close to normal.

```
n.VV = length(VV.times)
power = rep(seq(-1,1,0.25), each=n.VV)
VV.time = c(VV.times^-1, VV.times^-.75, VV.times^-.5, VV.times^-.25, log(VV.times), VV.times^.25,
ggplot(data.frame(power, VV.time), aes(sample=VV.time)) + stat_qq() + facet_wrap(~power, scales='
```

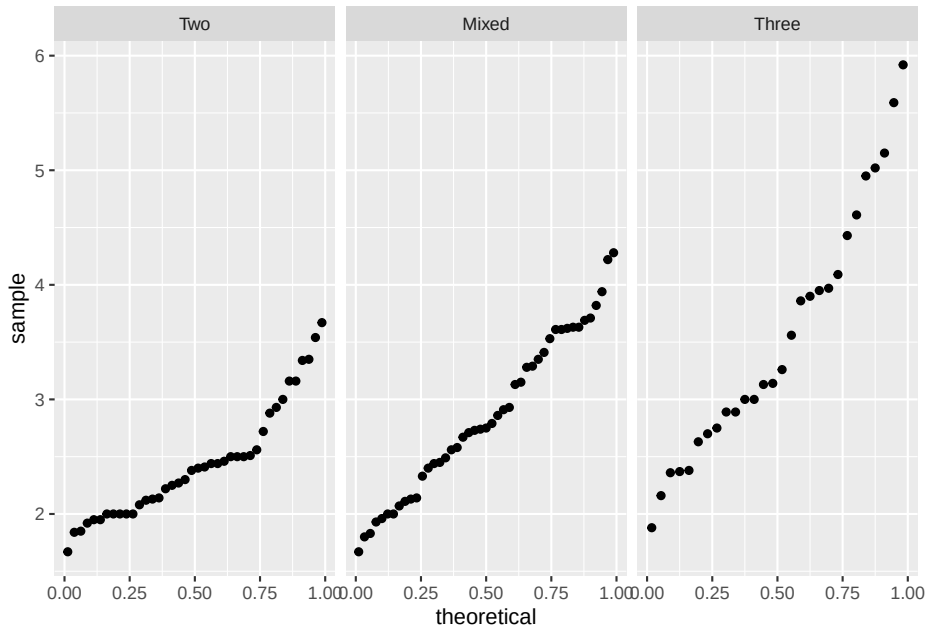


Here τ-values of 0 (the log transformation) and -0.25 give the straightest normal QQ plots. Since it's much, much easier to interpret $\log(\tau)$ than $\tau^{-0.25}$, we strongly prefer the log transformation.

2.4.4 Example: Food webs

The data set `food.web` contains the quantitative variable `mean.length` (the average number of links in the food chains in an ecosystem) and the categorical variable `dimension` (whether the ecosystem is two-dimensional like a plain, three-dimensional like a forest, or mixed.) We want to compare the average food chain length across these three type of ecosystem. Start with uniform QQ plots:

```
ggplot(food.web, aes(sample=mean.length)) +
  stat_qq(distribution = qunif) +
  facet_grid(~dimension)
```



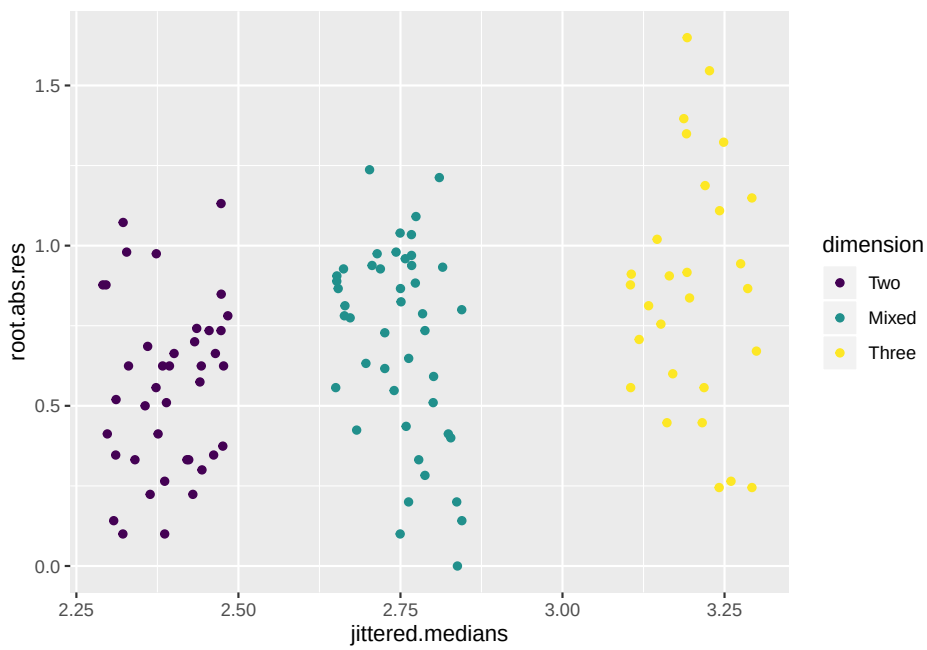
There's a clear difference in scale: Some of the three-dimensional webs approach a mean length of 6, while none of the two-dimensional webs even makes it to 4.

Is this a shift, or is there a difference in spread as well? This will be easier to see from the residuals. We'll display these in a **spread-location plot**:

- The x-axis is the model estimate. Here I'll use the median since it's more resistant to outliers than the mean. Also, since the model is categorical, I'll add some random uniform noise (the size of the noise term can be determined by trial and error.)
- The y-axis has some transformation of the residuals. Cleveland uses the square root of the absolute residuals. I'm not sure there's any theoretical justification for the square root absolute transformation, but it tends to work well: the absolute value transforms the equal spread problem into an equal location problem, while the square root transformation intends to reduce skewness.

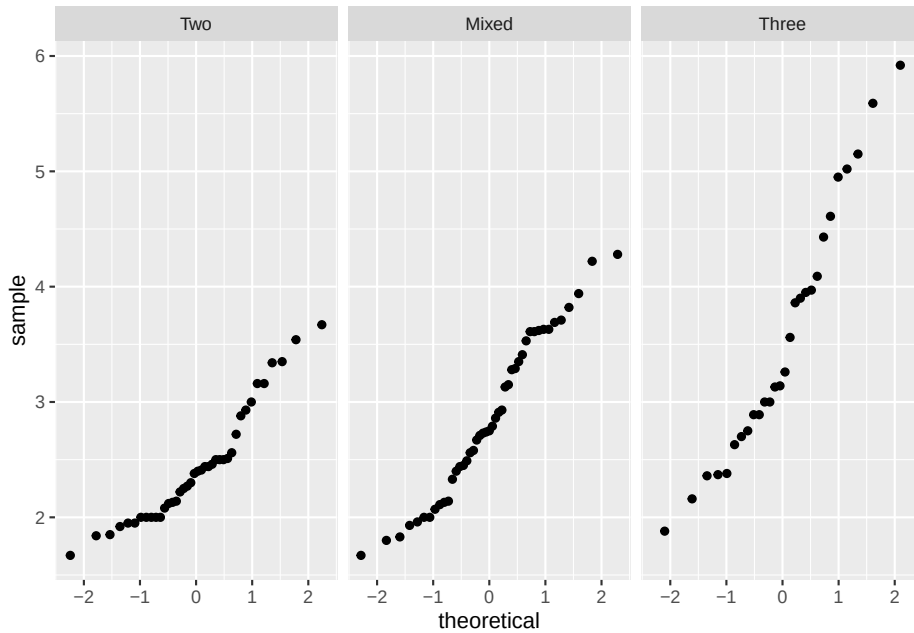
```
web.length = food.web$mean.length
dimension = food.web$dimension
n = nrow(food.web)
median.3 = median(web.length[dimension=="Three"])
median.2 = median(web.length[dimension=="Two"])
median.mixed = median(web.length[dimension=="Mixed"])
group.median = rep(NA, n)
group.median[dimension == "Three"] = median.3
group.median[dimension == "Two"] = median.2
group.median[dimension == "Mixed"] = median.mixed
```

```
jittered.medians = group.median + runif(n, -0.1, 0.1)
root.abs.res = sqrt(abs(web.length - group.median))
food.web.sl = data.frame(jittered.medians, root.abs.res, dimension)
ggplot(food.web.sl, aes(jittered.medians, root.abs.res, col=dimension)) +
  geom_point()
```



Pretty clearly the three sets of residuals are not all at the same average height. We also check for normality:

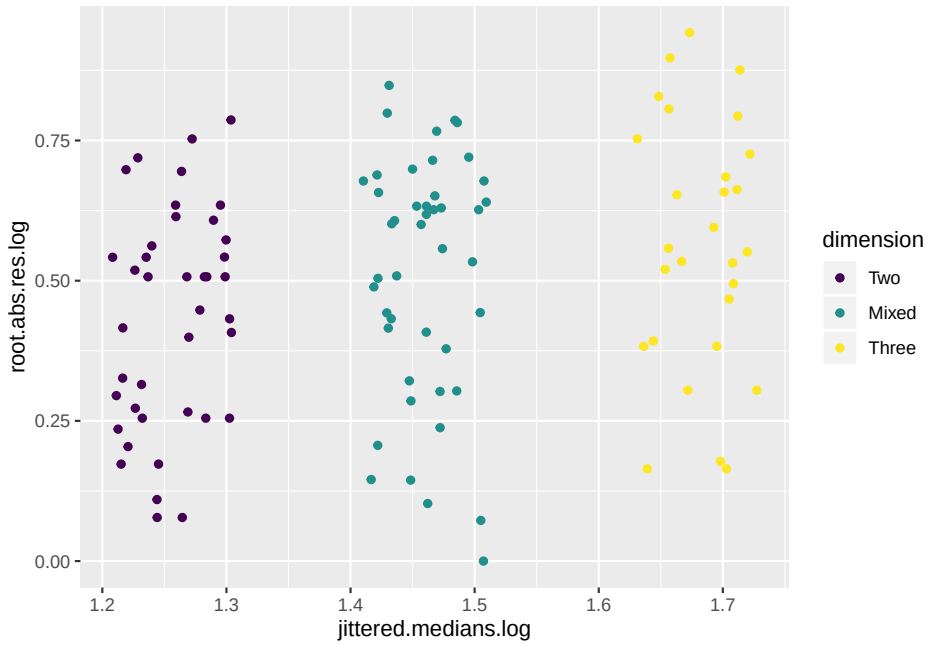
```
ggplot(food.web, aes(sample = mean.length)) +
  stat_qq() + facet_wrap(~dimension)
```



All the normal QQ plots look like they curve at least somewhat upward, indicating right skew. It looks like a transformation will help.

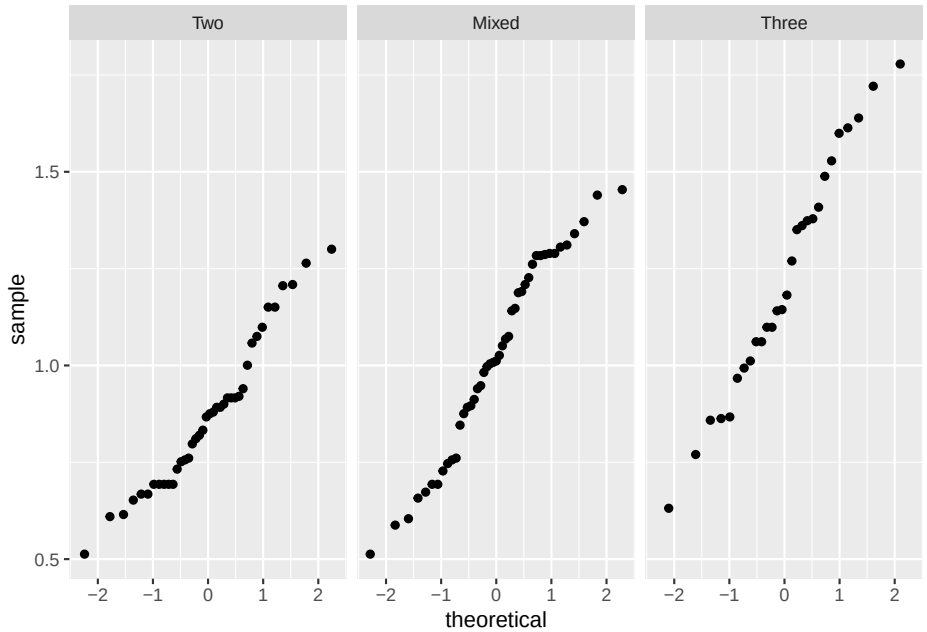
The food web mean lengths are positive, so we'll first try a log transformation. We repeat the spread-location plot on the transformed data:

```
log.web.length = log2(food.web$mean.length)
median.3.log = median(log.web.length[dimension=="Three"])
median.2.log = median(log.web.length[dimension=="Two"])
median.mixed.log = median(log.web.length[dimension=="Mixed"])
group.median.log = rep(NA, n)
group.median.log[dimension == "Three"] = median.3.log
group.median.log[dimension == "Two"] = median.2.log
group.median.log[dimension == "Mixed"] = median.mixed.log
jittered.medians.log = group.median.log + runif(n, -0.05, 0.05)
root.abs.res.log = sqrt(abs(log.web.length - group.median.log))
food.web.log.sl = data.frame(jittered.medians.log, root.abs.res.log, dimension)
ggplot(food.web.log.sl, aes(jittered.medians.log, root.abs.res.log, col=dimension)) +
  geom_point()
```



It's better but it still seems like there are differences between the three groups of residuals. We also check normality:

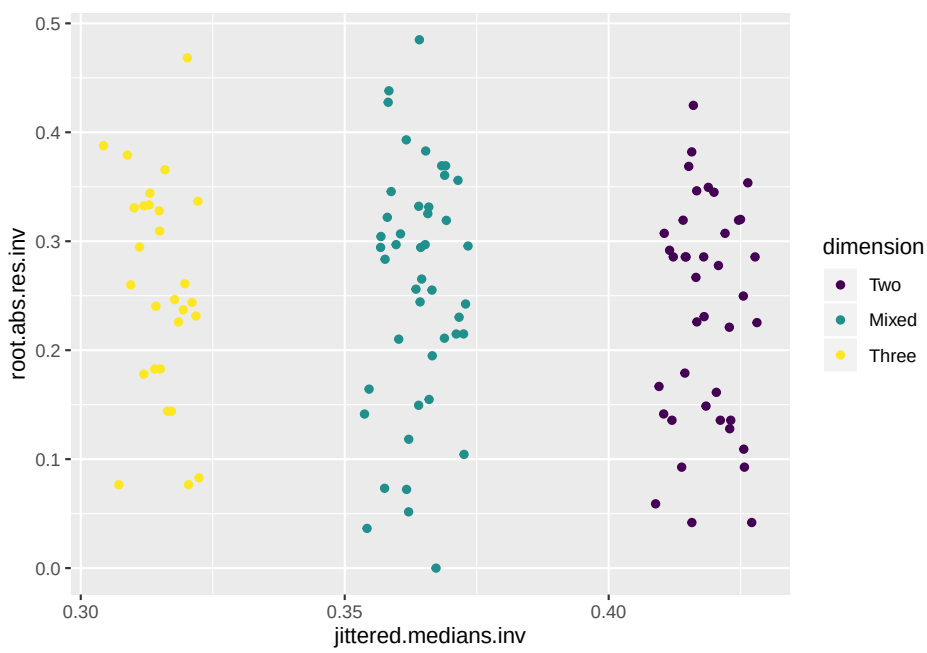
```
ggplot(food.web, aes(sample = log(mean.length))) +
  stat_qq() + facet_wrap(~dimension)
```



The lines still don't look as straight as they could be – the log doesn't do enough to "normalize" the data.

We could try out a number of transformations, but if $\tau = 0$ didn't quite do enough, then $\tau = -1$, the inverse transformation, seems like the next candidate. Try the spread-location plot again:

```
inv.web.length = 1/food.web$mean.length
median.3.inv = median(inv.web.length[dimension=="Three"])
median.2.inv = median(inv.web.length[dimension=="Two"])
median.mixed.inv = median(inv.web.length[dimension=="Mixed"])
group.median.inv = rep(NA, n)
group.median.inv[dimension == "Three"] = median.3.inv
group.median.inv[dimension == "Two"] = median.2.inv
group.median.inv[dimension == "Mixed"] = median.mixed.inv
jittered.medians.inv = group.median.inv + runif(n, -0.01, 0.01)
root.abs.res.inv = sqrt(abs(inv.web.length - group.median.inv))
food.web.inv.sl = data.frame(jittered.medians.inv, root.abs.res.inv, dimension)
ggplot(food.web.inv.sl, aes(jittered.medians.inv, root.abs.res.inv, col=dimension)) +
  geom_point()
```



This looks the best of those we've seen. We can calculate the three group means:

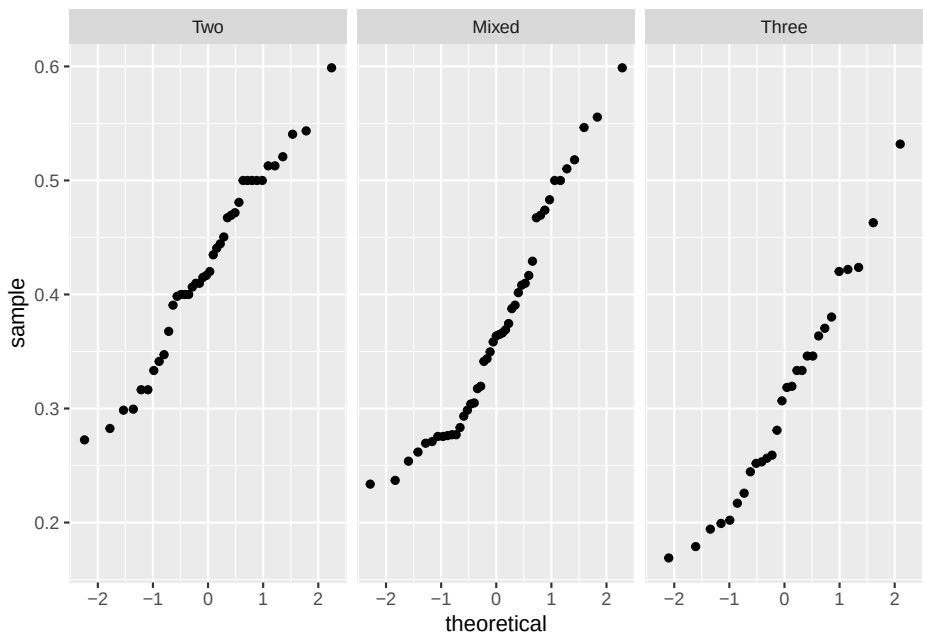
```
aggregate(root.abs.res.inv ~ dimension, FUN=mean)
```

```
##   dimension root.abs.res.inv
## 1       Two        0.2332382
```

```
## 2     Mixed        0.2563367
## 3     Three        0.2579944
```

They're close enough that the differences can be explained as noise. What about normality?

```
ggplot(food.web, aes(sample = 1/mean.length)) +
  stat_qq() + facet_wrap(~dimension)
```

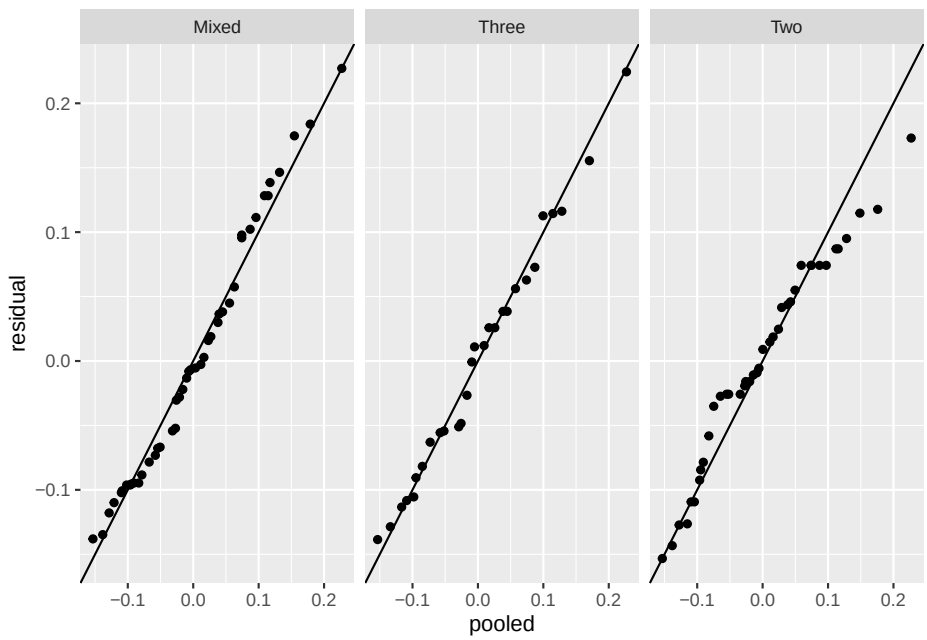


These look straight. We're pretty happy with the inverse transformation. Furthermore, it's interpretable – we only have to think in terms of chains per link rather than links per change (though note we're no longer strictly dealing with a "mean.")

The spread is similar for all three QQ plots, suggesting that we might be able to pool the residuals. To check, this we create a vector of pooled residuals (`food.web.res` in the code below,) then draw a two-sample QQ plot for each of the three sets of residuals against the pooled set. We cheat by using the `qqplot()` function to calculate the quantiles for us. (Of course it would be much easier and only slightly uglier to just use `qqplot()` to draw the plots for us.)

```
food.web.lm = lm(inv.web.length ~ dimension)
food.web.res = residuals(food.web.lm)
res.qq.3 = qqplot(food.web.res, food.web.res[dimension == "Three"], plot.it=FALSE)
res.qq.2 = qqplot(food.web.res, food.web.res[dimension == "Two"], plot.it=FALSE)
res.qq.mixed = qqplot(food.web.res, food.web.res[dimension == "Mixed"], plot.it=FALSE)
```

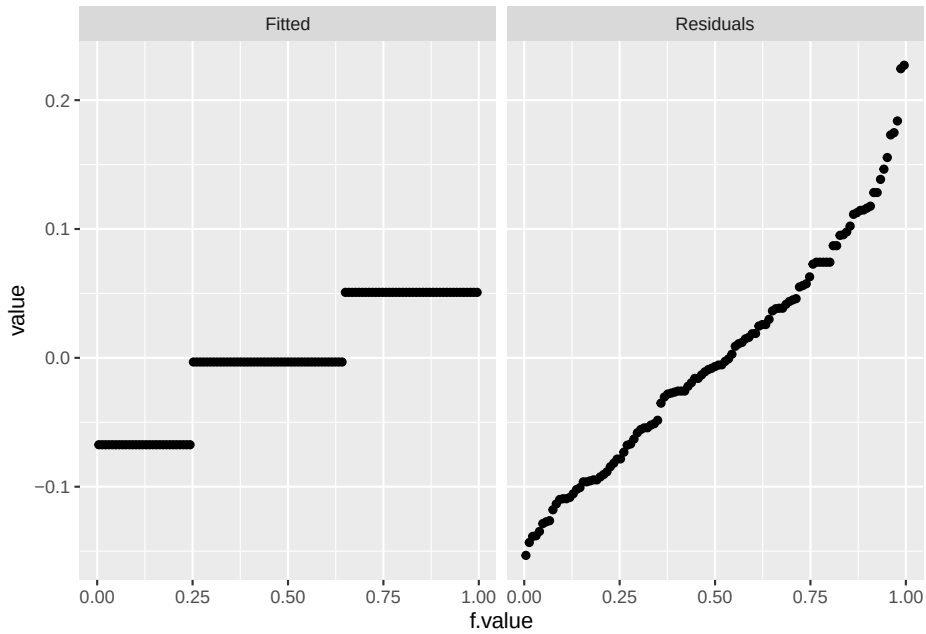
```
food.web.res.qq = data.frame(pooled = c(res.qq.3$x, res.qq.2$x, res.qq.mixed$x),
  residual = c(res.qq.3$y, res.qq.2$y, res.qq.mixed$y),
  dimension=c(rep("Three",length(res.qq.3$x)),
  rep("Two",length(res.qq.2$x)),
  rep("Mixed",length(res.qq.mixed$x))))
ggplot(food.web.res.qq, aes(pooled, residual)) + geom_point() +
  geom_abline() + facet_wrap(~dimension)
```



It's not perfect, but the three graphs look reasonably similar, so pooling is justifiable.

We'll draw a residual-fit plot to see how much variation our model capture. Again, we'll use `gather()` to get our data in the right shape.

```
food.web.fitted = sort(fitted.values(food.web.lm)) - mean(fitted.values(food.web.lm))
n = length(inv.web.length)
f.value = (0.5:(n - 0.5)) / n
food.web.fit = data.frame(f.value, Fitted=food.web.fitted, Residuals=sort(food.web.res))
library(tidyr)
food.web.fit.long = food.web.fit %>% gather(type, value, Fitted:Residuals)
ggplot(food.web.fit.long, aes(x=f.value, y=value)) +
  geom_point() + facet_wrap(~type)
```



The fitted values are close together compared to the residuals. While the model may be useful, we should remember it only accounts for a fraction of the variation in the data.

2.5 Robust fits

READ: Cleveland pp. 68–79.

Load the usual stuff:

```
library(ggplot2)
load("lattice.RData")
```

2.5.1 Bin packing

Suppose that we have a number of files, with sizes IID uniform on $[0, 0.8]$. We want to put these files on to disks of capacity 1 using the fewest number of disks possible, i.e.~wasting the smallest amount of empty space. We want to explore the performance of a "bin packing" algorithm that provides an approximate solution.

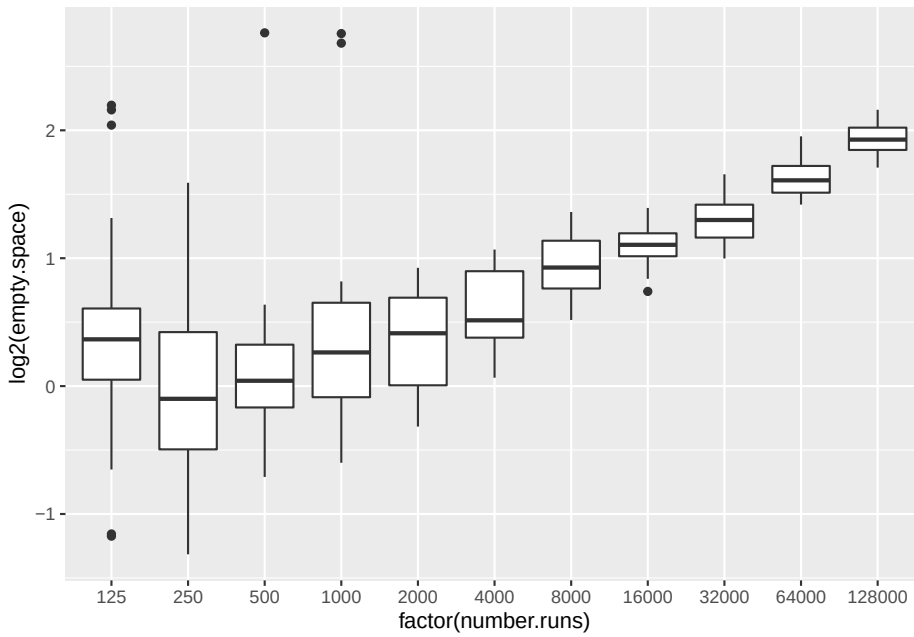
Cleveland's data frame `bin.packing` contains two variables:

- `empty.space`: The amount of empty space wasted.
- `number.runs`: The number of randomly generated files.

For now we'll treat `number.runs` as categorical. We suspect there could be

multiplicative effects, so for now we'll study the base 2 log of `empty.space`. Let's draw boxplots, split by `number.runs`:

```
ggplot(bin.packing, aes(factor(number.runs), log2(empty.space))) + geom_boxplot()
```

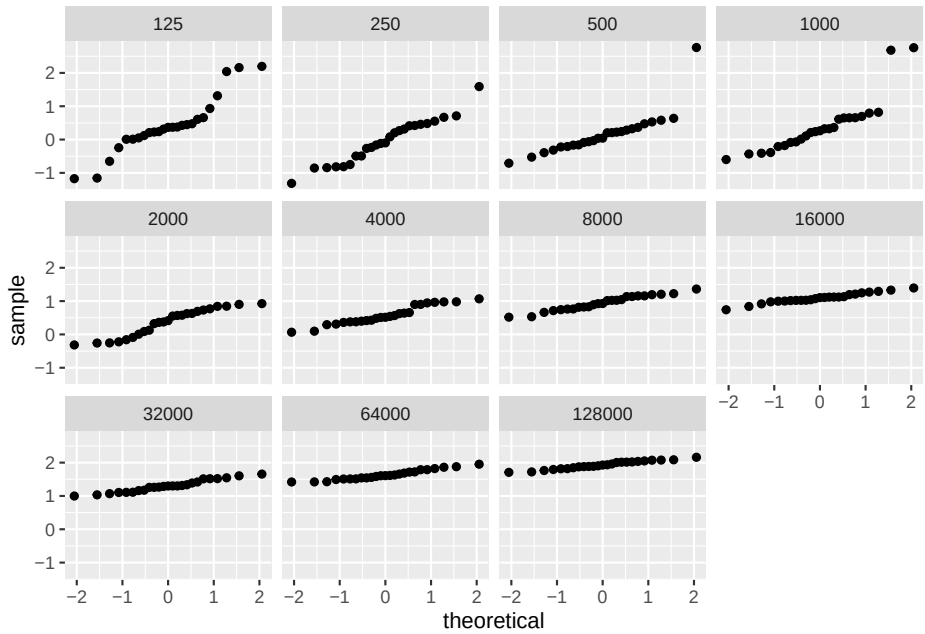


A couple of things are clear:

- Log empty space generally goes up with the number of runs.
- The data isn't homoscedastic: The spread decreases as number of runs increases.

Next we check the normal QQ plots, faceted by the number of runs.

```
ggplot(bin.packing, aes(sample=log2(empty.space))) +
  stat_qq() + facet_wrap(~number.runs)
```



For large numbers of runs, the QQ plots are well-fitted by straight lines. However for smallest numbers of runs there are difficulties – especially for less than 1000 runs, where there are major outliers.

Because of the heteroskedasticity and outliers, we might prefer to both build our model and explore our residuals in a more robust way. The median is more outlier-resistant than the mean, so we'll use those as our fitted values.

In Cleveland's notation: Let b_{in} be the ith log empty space measurement for the bin packing run with n weights. Let l_n be the medians. The fitted values are

$$\hat{b}_{in} = l_n$$

and the residuals are

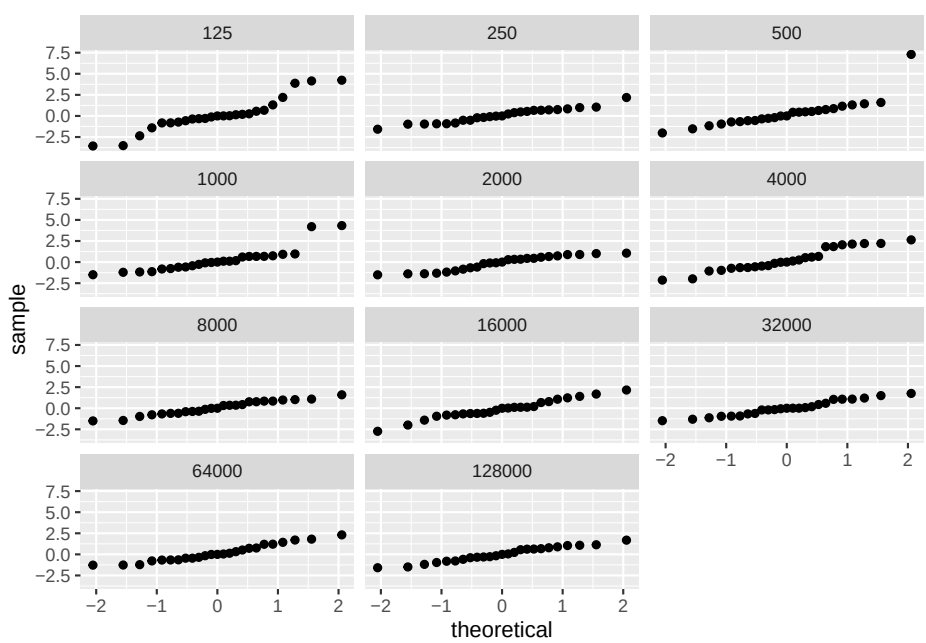
$$\hat{\epsilon}_{in} = b_{in} - \hat{b}_{in}$$

Now we adjust the residuals for heteroscedasticity. Let s_n be the **median absolute deviations** or **mads**: that is, for each n, the median of the absolute value of the residuals. The `mad()` function in R gives the median absolute deviations (multiplied by a constant `1/qnorm(3/4)` to put the estimate on the same scale as the standard deviation.)

The **spread-standardized residuals** are

$$\frac{\hat{\epsilon}}{s_n}$$

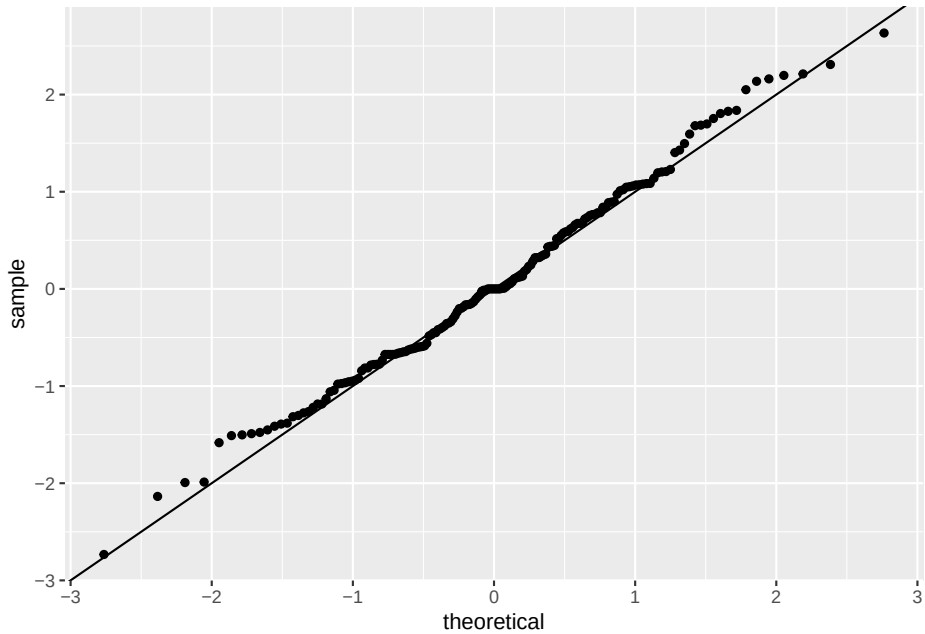
```
number.runs = bin.packing$number.runs
log2.space = log2(bin.packing$empty.space)
log2.packing = data.frame(log2.space, number.runs)
log2.space.medians = aggregate(log2.space~number.runs, median, data=log2.packing)
log2.space.mad = aggregate(log2.space~number.runs, mad, data=log2.packing)
n = nrow(log2.packing)
log2.space.fitted = rep(NA, n)
log2.space.madlist = rep(NA, n)
for(J in 1:n){
  which.runs = which(log2.space.medians$number.runs == number.runs[J])
  log2.space.fitted[J] = log2.space.medians$log2.space[which.runs]
  log2.space.madlist[J] = log2.space.mad$log2.space[which.runs]
}
log2.space.residuals = log2.space - log2.space.fitted
log2.space.standardized = log2.space.residuals / log2.space.madlist
log2.model = data.frame(number.runs,
  residuals=log2.space.residuals,
  std.residuals=log2.space.standardized)
ggplot(log2.model, aes(sample=std.residuals)) +
  stat_qq() + facet_wrap(~number.runs, ncol=3)
```



It's still a bit hard to see what's going on because of the outliers, particularly

when the number of runs is 1000 or more. Let's pool the residuals for those cases and draw a normal QQ plot.

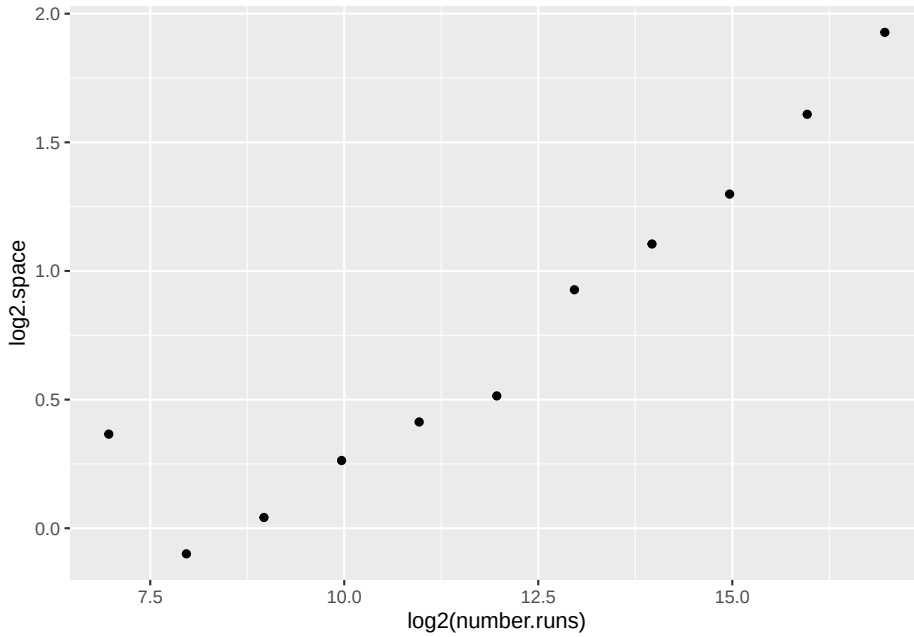
```
log2.model.big.n = log2.model[number.runs > 1000,]
ggplot(log2.model.big.n, aes(sample=std.residuals)) +
  stat_qq() + geom_abline()
```



There's an S-shape to the plot. This can happen for many different reasons, but one possibility is over-transformation: maybe our log transformation is hurting rather than helping.

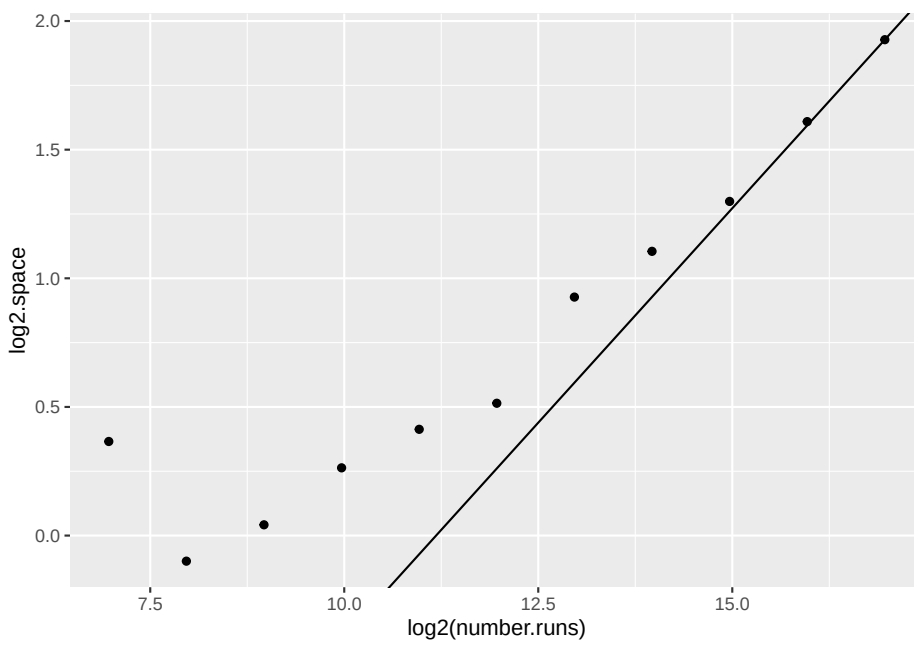
Theory suggests that on a log-log scale, then as the number of runs gets large, empty space approaches a linear function of number of runs with slope 1/3. We plot the median log empty space for each number of runs:

```
k = nrow(log2.space.medians)
plot248 = ggplot(log2.space.medians,
       aes(x=log2(number.runs), y=log2.space)) +
  geom_point()
plot248
```



Then we add a straight line with slope 1/3 through the last point:

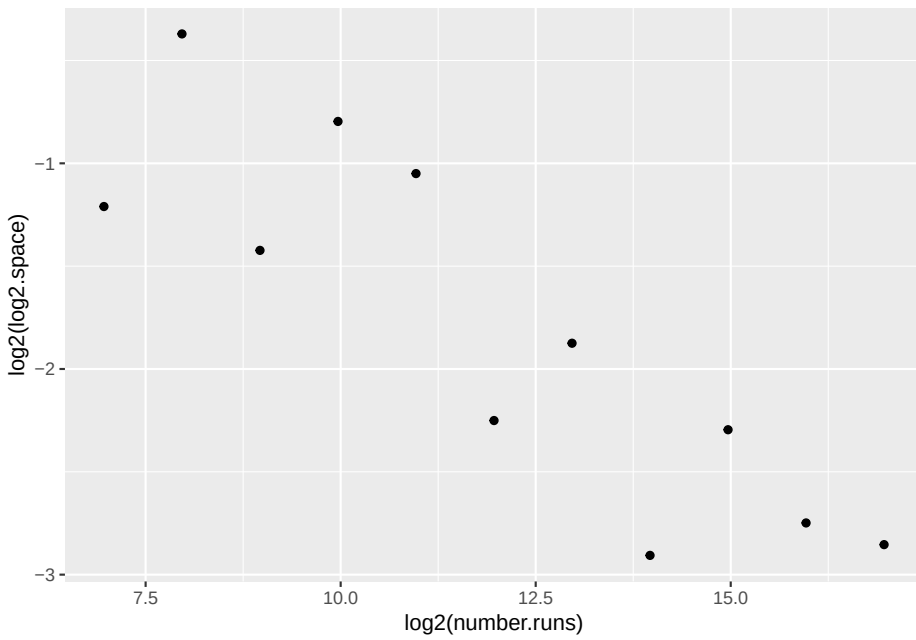
```
plot248 + geom_abline(slope=1/3,
  intercept=log2.space.medians[k,2]-1/3*log2(log2.space.medians[k,1]))
```



The line does eventually provide a good fit, though we would've liked a few more

points on the right-hand side to be sure. What about the spreads? Plot the log mads against the log number of runs:

```
ggplot(log2.space.mad, aes(x=log2(number.runs), y=log2(log2.space))) +
  geom_point()
```



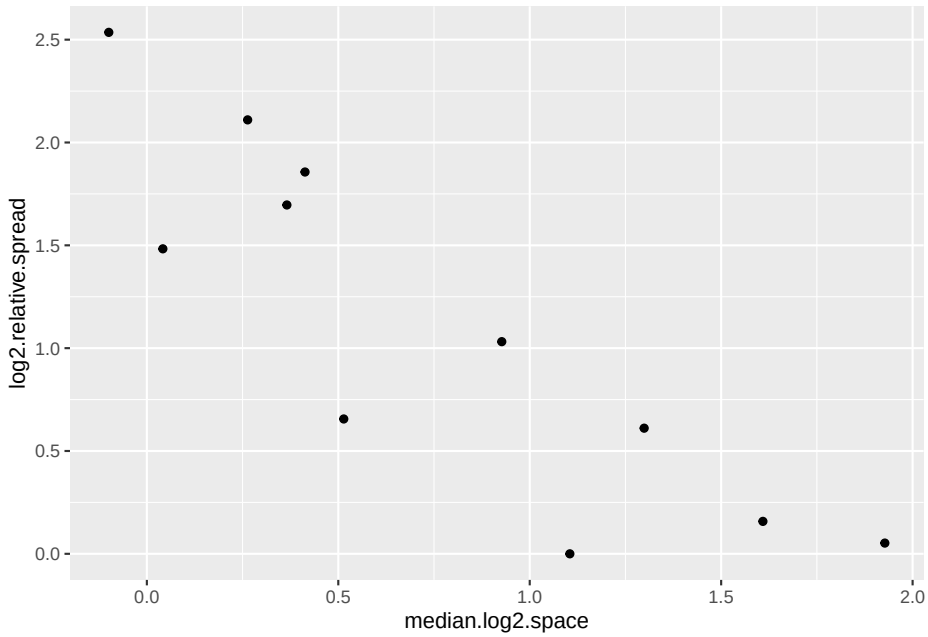
The mads decrease as the number of runs increases. This again is consistent with overtransformation.

To check more thoroughly for heteroscedasticty, we draw spread-location plots.

- For location, we'll use the median log empty space.
- For spread, we'll divide the mads by the smallest mad, then take logs.

If the data is homoscedastic after transformation, this plot should look like random noise.

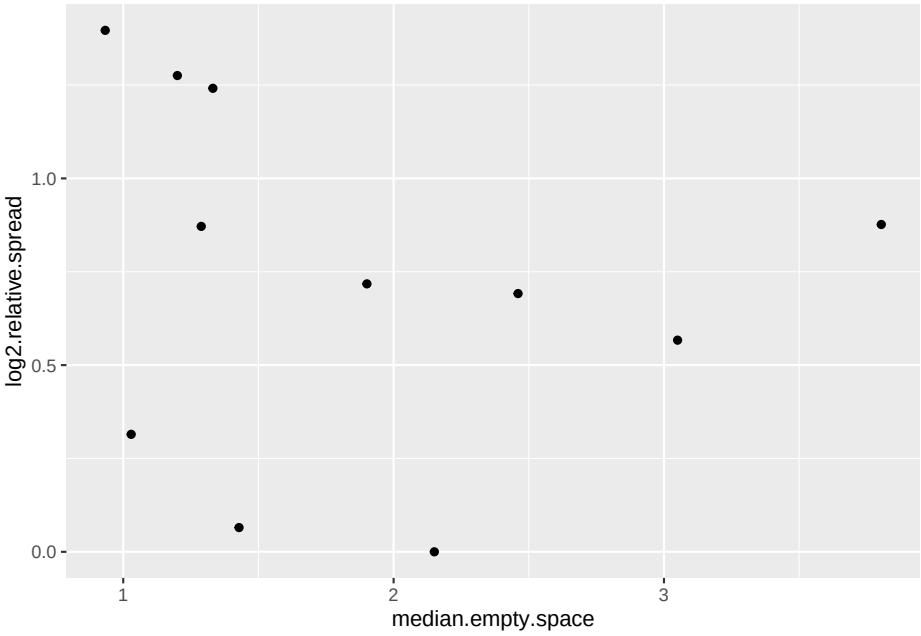
```
median.log2.space = log2.space.medians$log2.space
log2.relative.spread = log2(log2.space.mad$log2.space / min(log2.space.mad$log2.space)
log2.sl = data.frame(median.log2.space, log2.relative.spread)
ggplot(log2.sl, aes(x=median.log2.space, y=log2.relative.spread)) +
  geom_point()
```



It's not random noise – it's a downward sloping line.

We can now be confident that our log transformation has led to heteroscdesticity. In that case, let's start over again *without* transformation. The spread-location plot is:

```
empty.space.medians = aggregate(empty.space~number.runs, median, data=bin.packing)
empty.space.mad = aggregate(empty.space~number.runs, mad, data=bin.packing)
empty.space.sl = data.frame(median.empty.space=empty.space.medians$empty.space,
  log2.relative.spread=log2(empty.space.mad$empty.space/min(empty.space.mad$empty.space)))
ggplot(empty.space.sl, aes(x=median.empty.space, y=log2.relative.spread)) +
  geom_point()
```



This time it looks pretty random. We're much closer to heteroscedasticity without the transformation.

The major moral: In EDA, we're allowed to do dumb things because we're explorers. We do need to check our models so that we'll find out if we're doing dumb things. If we are, there's no shame in starting all over again.

Chapter 3

Bivariate data

In this chapter we'll explore data in which we measure two quantitative variables on individuals, and the individuals are close to independent. (This does not apply to, say, time series, since observations in time are generally dependent. We'll briefly cover time series later.) We'll mostly cover situations where one variable is taken to be a *response* and the other an *explanatory* variable. If we think it's much more plausble that x affect y than that y affect x, then x would be the explanatory and y would be the response. Later in the chapter we'll briefly address situations where this distinction may not be so clear.

Let's load our usual packages and data to start:

```
library(ggplot2)
load("lattice.RData")
```

3.1 Linear modeling: Fly eyes

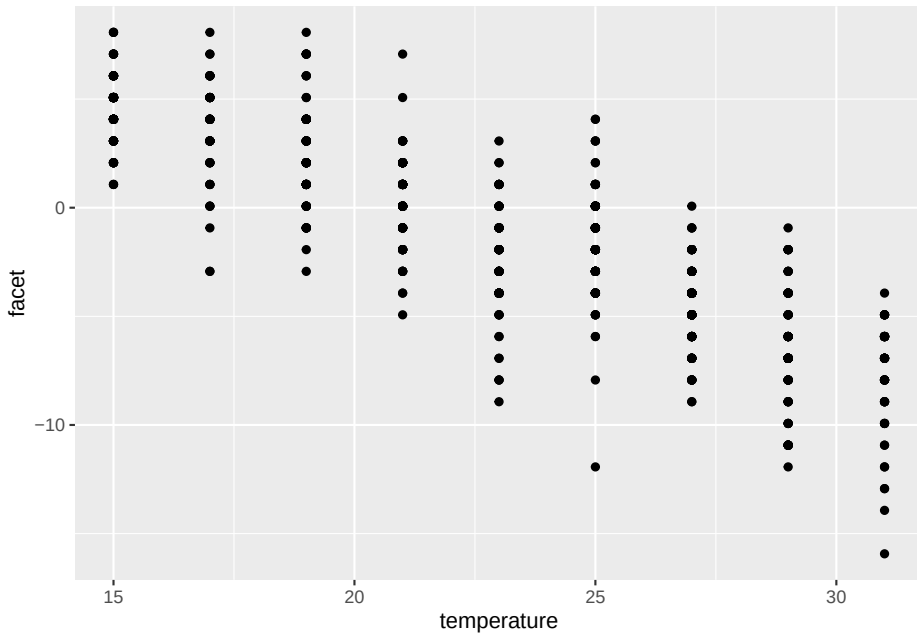
READ: Cleveland pp. 86–119.

The data set `fly` in `lattice.RData` contains two variables from an experiment about breeding flies (see Hersh, The effect of temperature upon the heterozygotes in the series of Drosophila):

- `temperature` of the aquarium in which flies were hatched (in degrees Celsius);
- `facet`: a measure of the number of facets in a fly's eye (on an "essentially logarithmic" scale.)

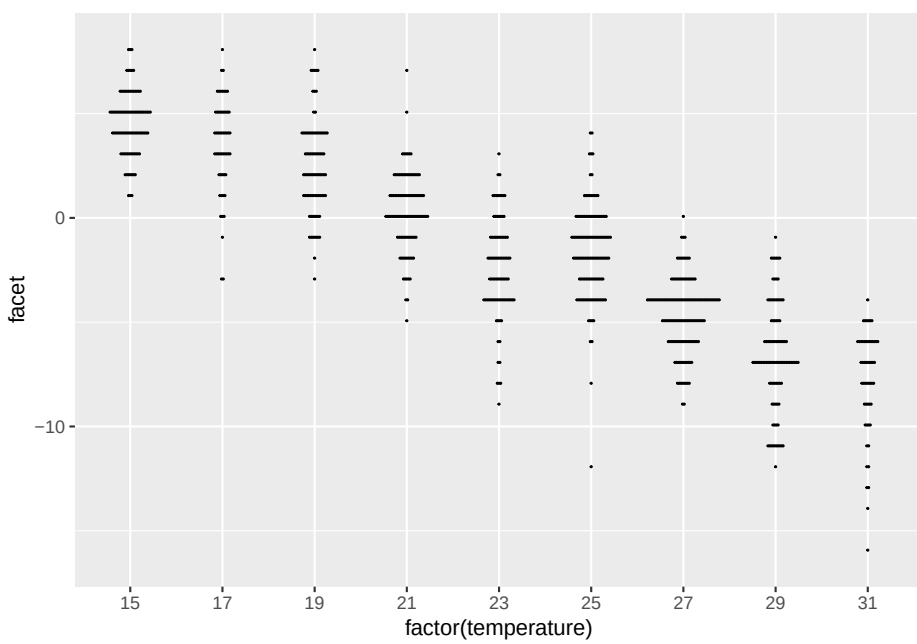
Draw a scatterplot:

```
ggplot(fly, aes(x=temperature, y=facet)) + geom_point()
```



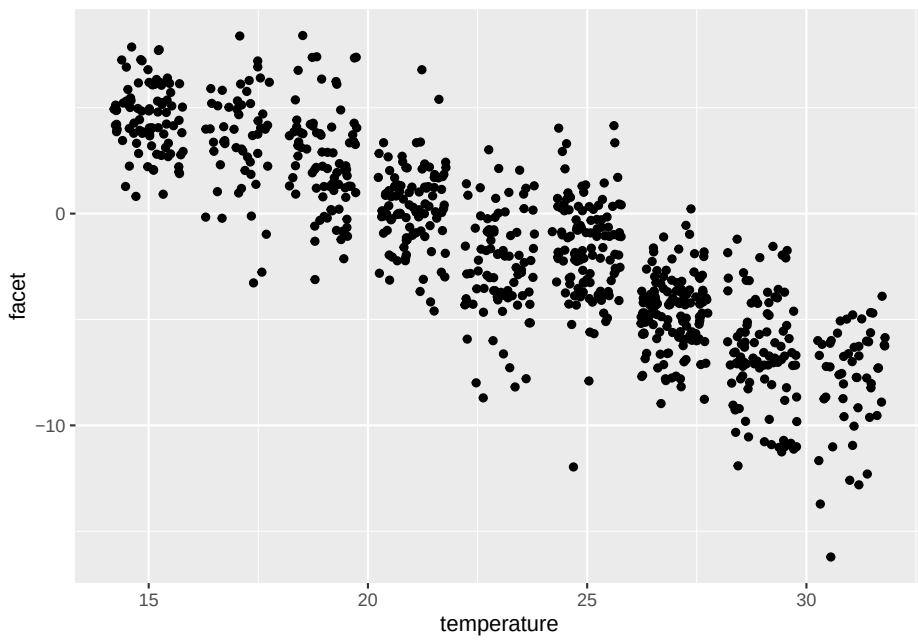
With only nine x-values, many observations are on top of each other. One alternative is a dot plot:

```
ggplot(fly, aes(x = factor(temperature), y = facet)) +
  geom_dotplot(binaxis = "y", binwidth = 1, stackdir = "center", dotsize = 0.08)
```



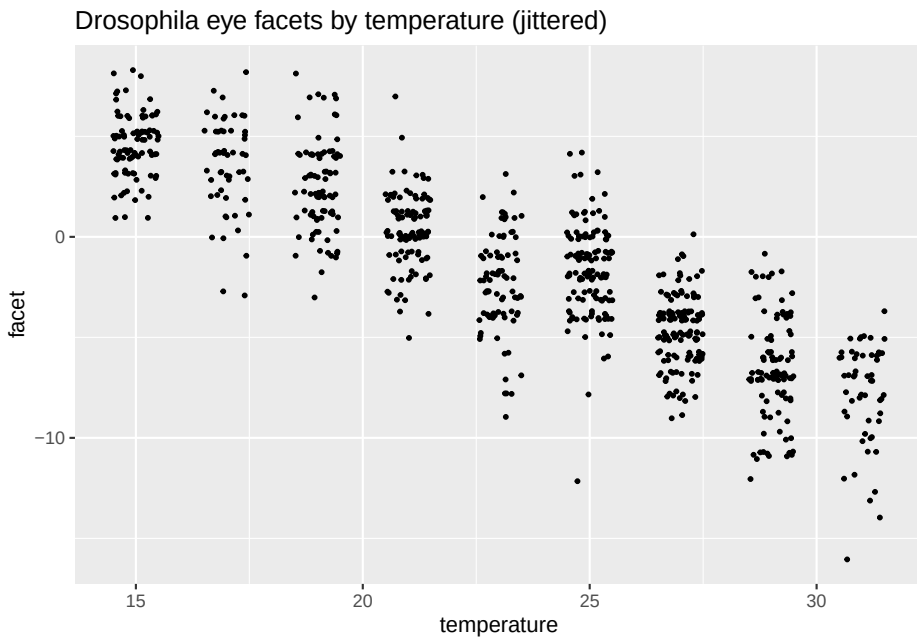
but there are a few too many dots for this to look good. Instead we can *jitter* the data (add random noise) using `geom_jitter()`:

```
ggplot(fly, aes(x=temperature, y=facet)) + geom_jitter()
```



You can control the amount of jittering using `height` and `width` arguments and point size using `size`:

```
gg = ggplot(fly, aes(x=temperature, y=facet)) + geom_jitter(height = 0.25, width = 0.5, size = 0.
  labs(title = "Drosophila eye facets by temperature (jittered)")
gg
```



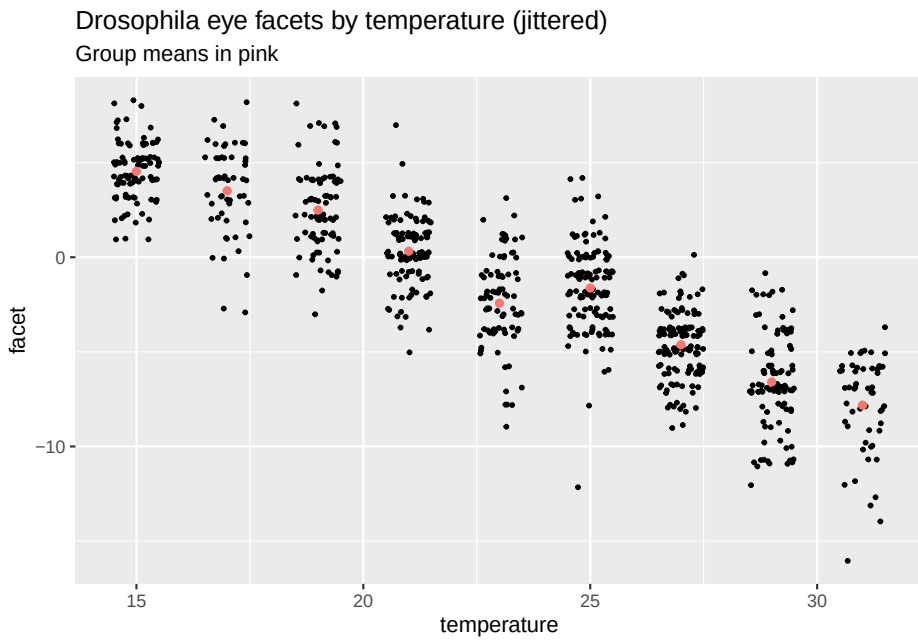
The facet variable looks like it decreases approximately linearly with temperature. However, the data at $x = 23$ seems to be a bit lower than we'd expect given the rest of the data. We can get further insight by calculating the mean of facet for each temperature. I'll use the `aggregate()` function:

```
facet.means = aggregate(facet~temperature, mean, data=fly)
facet.means
```

```
##   temperature     facet
## 1          15  4.547778
## 2          17  3.514444
## 3          19  2.491687
## 4          21  0.320000
## 5          23 -2.430000
## 6          25 -1.626721
## 7          27 -4.623431
## 8          29 -6.589794
## 9          31 -7.818889
```

Add these to the existing jittered plot. (By default, this produces an uninformative legend; we remove this and add a subtitle instead.)

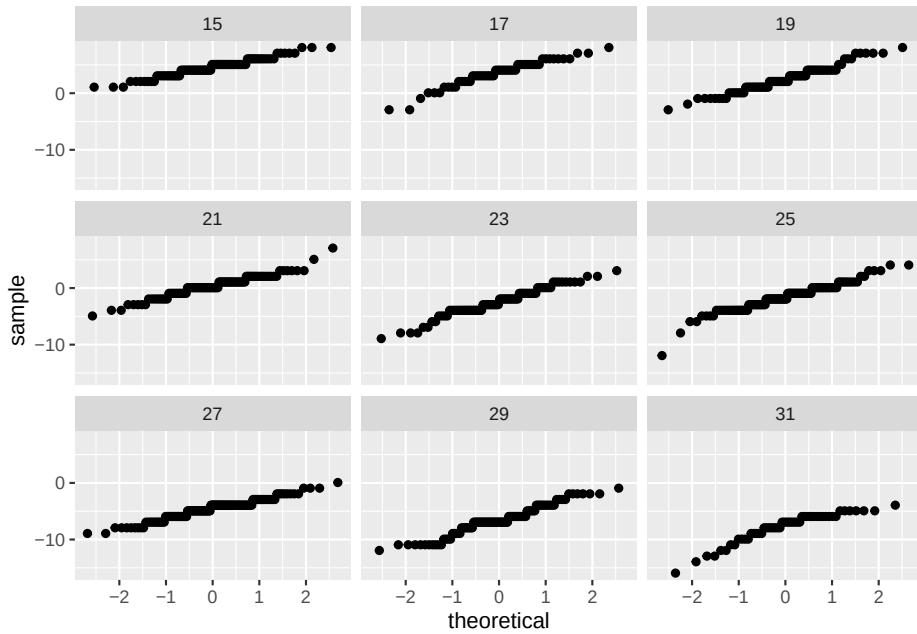
```
gg + geom_point(data = facet.means, aes(color = "pink")) + theme(legend.position="none
  labs(subtitle = "Group means in pink")
```



R.A. Fisher did an analysis of variance that found the deviations from the line were statistically significant, but that doesn't tell you much other than the probability model isn't literally true.

Can the data be well-modeled by a linear model with normal errors? Boxplots aren't a good guide to the shape of distributions, so let's draw normal QQ plots.

```
ggplot(fly, aes(sample=facet)) + stat_qq() + facet_wrap(~temperature, ncol=3)
```



The plots generally look like straight lines (the last one, for 31 degrees, might be a bit off.) Furthermore, the slopes look similar for each plot, which is consistent with homoscedasticity.

Can we fit a simple linear regression model? Remember the classical assumptions are:

1. Linearity
2. Independence of errors
3. Equal variance of errors (homoskedasticity)
4. Normality of errors

For EDA, assumptions 1 and 2 are important. Assumption 3 is only of interest if it's badly violated — minor heteroskedasticty is unlikely to lead to problems. Assumption 4 is generally irrelevant, though there are advantages in having at least an approximately symmetric error distribution.

In this case, all four assumptions look close to true. We may now fit a linear model using the `lm()` function.

```
facet.lm = lm(facet~temperature, data=fly)
```

Putting an `lm` object into `summary()` gives our usual bunch of junk:

```
summary(facet.lm)
```

```
##
## Call:
## lm(formula = facet ~ temperature, data = fly)
```

```
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.0265 -1.2319  0.0272  1.5628  6.9735
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 16.96221    0.39045   43.44   <2e-16 ***
## temperature -0.79463    0.01643  -48.38   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.264 on 821 degrees of freedom
## Multiple R-squared:  0.7403, Adjusted R-squared:   0.74
## F-statistic:  2340 on 1 and 821 DF,  p-value: < 2.2e-16
```

This is just about the best case scenario for seeing the whole output of `summary()`: it's a designed experiment and the classical linear regression assumptions (linear trend with IID Normal$(0, \sigma^2)$ errors) are approximately true, so the P-values are basically correct and conceivably of interest. Nevertheless, the output of `summary()` is probably TMI. (A good rule of thumb for EDA is don't calculate P-values, and if you do calculate P-values, don't show them to other people.)

A more concise alternative is `display()` in `library(arm)` (which you'd need to install):

```
library(arm)
display(facet.lm)
```

```
## lm(formula = facet ~ temperature, data = fly)
##             coef.est coef.se
## (Intercept) 16.96     0.39
## temperature -0.79     0.02
## ---
## n = 823, k = 2
## residual sd = 2.26, R-Squared = 0.74
```

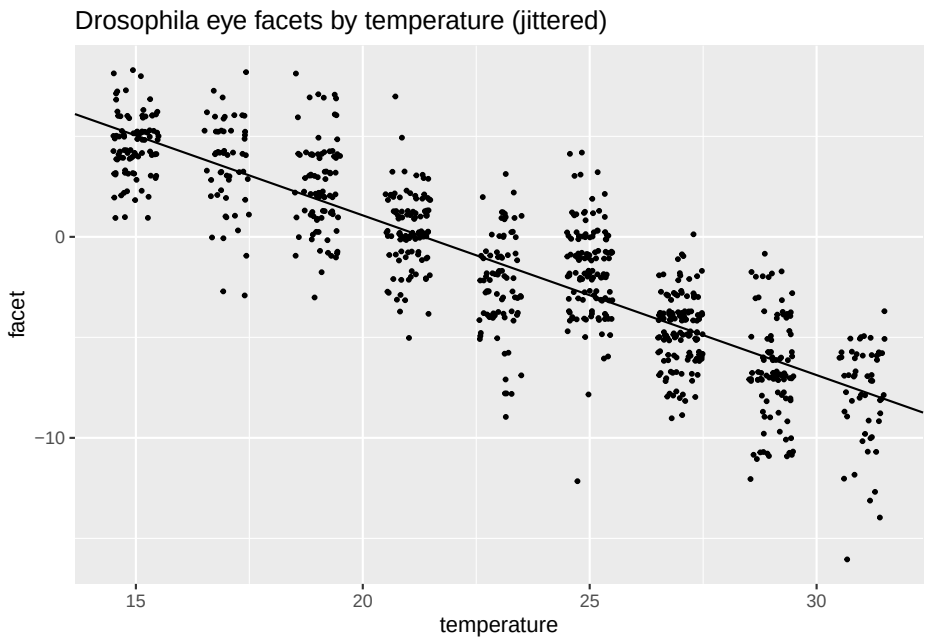
So the regression line is

$$\text{Predicted facet} = 16.96 - 0.79 \times \text{Temperature}$$

and the standard errors here should be reliable.

Plot the line on top of our existing graph:

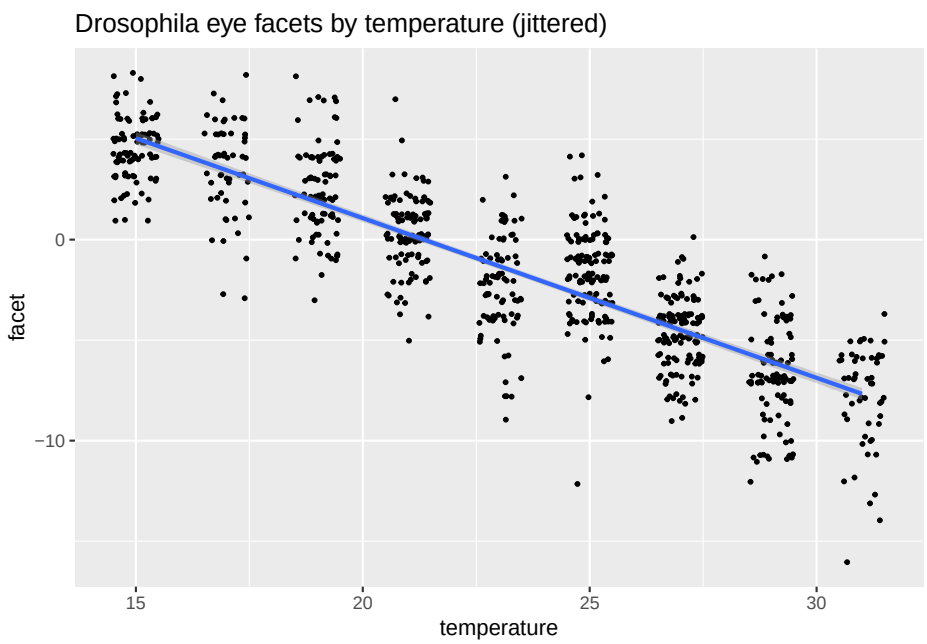
```
gg + geom_abline(intercept=facet.lm$coe[1], slope=facet.lm$coe[2])
```



The line fits the means well, and the mean for 23 degrees doesn't seem like it's too far from the line.

In fact, we could have got this using `geom_smooth()`:

```
gg + geom_smooth(method = "lm")
```

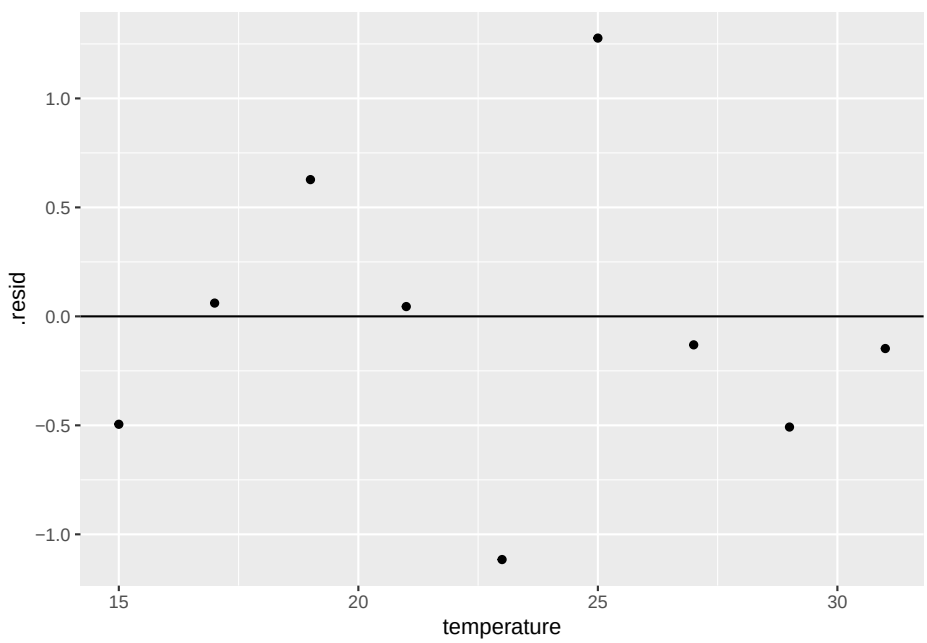


To study the 23 degree case more clearly, we should plot residuals. The `augment()` function in `library(broom)` creates a new data frame with, among other things, the original variables as well as a `.resid` column containing the residuals:

```
library(broom)
facet.lm.df = augment(facet.lm)
```

Use `aggregate()` to find the mean residual for each temperature (which will be the same as finding the mean of `facet` minus the fitted value at each temperature):

```
facet.resid.means = aggregate(.resid~temperature, mean, data=facet.lm.df)
ggplot(facet.resid.means, aes(x=temperature, y=.resid)) + geom_point() + geom_abline(slope=0)
```



The largest residuals are a bit over 1 unit of `facet` in magnitude. According to the original paper, when the experiment was performed, the temperature in the 23 aquarium varied by up to 3 degrees. Looking at the last two residual plots, this would be a sufficient explanation for the large residual for that x-value.

3.2 Nonlinear modeling

READ: Cleveland pp. 86–119.

3.2.1 Ganglion data

A ganglion is a nerve cell cluster. The data set `ganglion` contains data on ganglia in the retina of 14 cat fetuses. There are two variables:

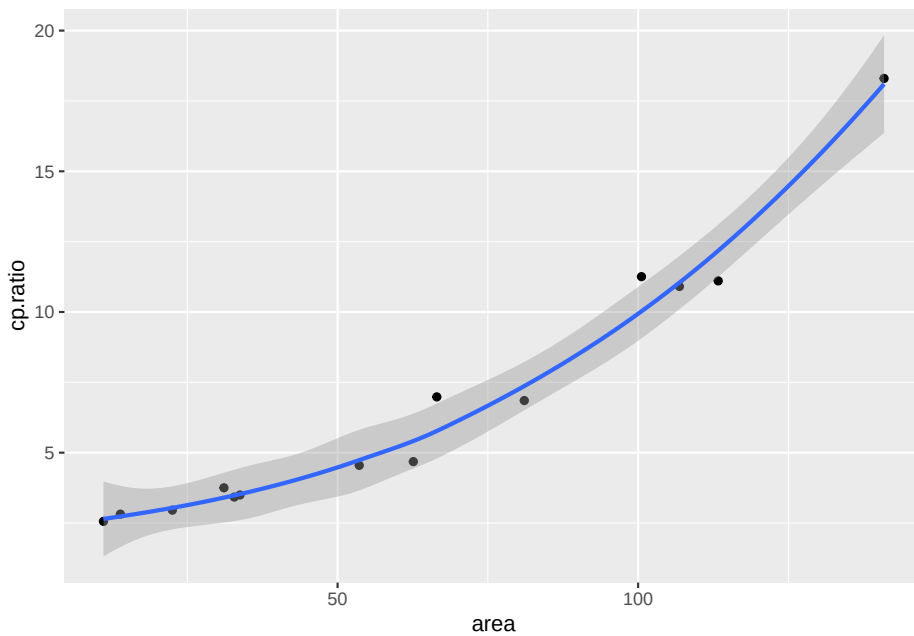
- `area` is retinal area (in mm^2)
- `cp.ratio` is the ratio of ganglion cell density in the center of the retina to the ganglion cell density in the periphery (edge) of the retina.

We draw a scatterplot:

```
ganglion.gg = ggplot(ganglion, aes(x=area, y=cp.ratio)) + geom_point()
```

Let's add a smooth curve to the plot.

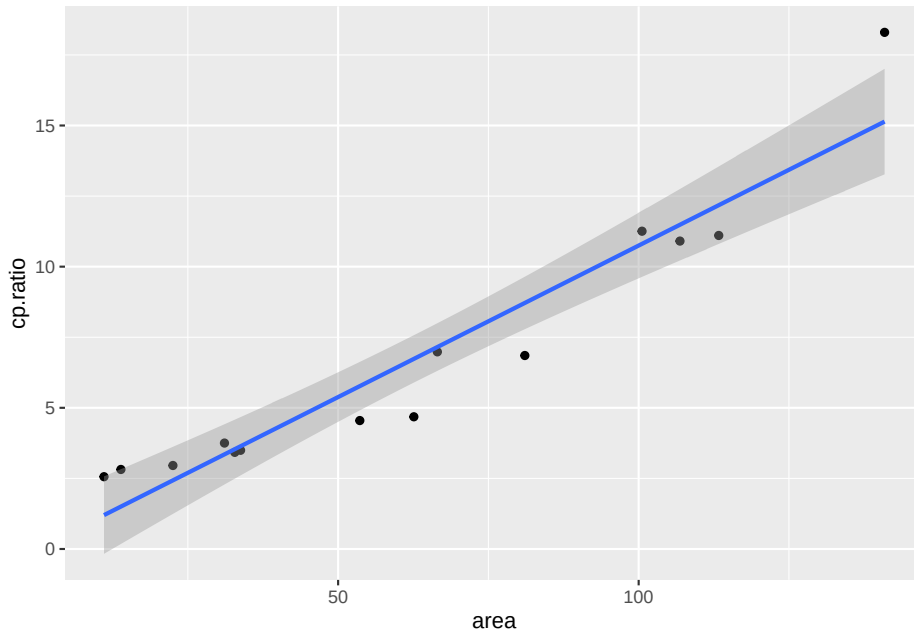
```
ganglion.gg + geom_smooth()
```



By default, the curve is fitted using the "loess" method; we'll talk more about this method later. The 95% confidence bands are for the fit, not for predictions (so we do not expect 95% of the data to be contained in the shaded area) and shouldn't be taken too literally in any case.

Visually, we see the loess fit is quite curved. How well would a straight line do? We can add the regression line, bearing in mind that the shaded area below gives confidence bounds based on assumptions that are likely wrong.

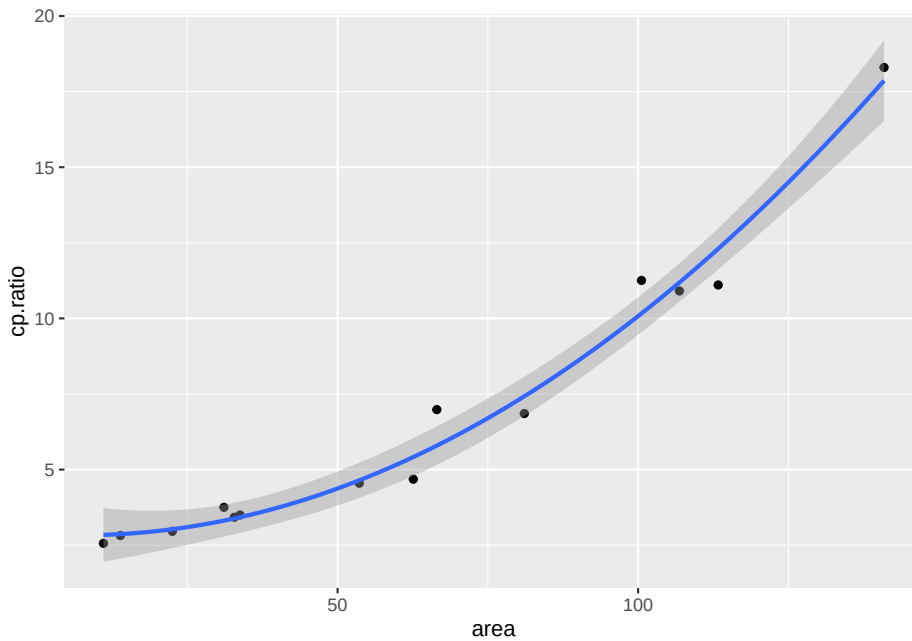
```
ganglion.gg + geom_smooth(method="lm")
```



We could look more carefully at the residuals, but it seems clear that the loess curve provides a better fit. Note also there's no straight line that goes through the entire loess confidence band in the previous graph (though again, `ggplot()`'s loess confidence bands are somewhat mysterious and not obviously reliable.)

If a straight line doesn't work and you still want a parametric fit, try a quadratic.

```
ganglion.gg + geom_smooth(method="lm", formula = y ~ x + I(x^2))
```



The quadratic fit looks more or less the same as the loess fit. The confidence bands are much narrower – they probably underestimate the uncertainty unless we have strong reason to believe the parametric model is correct.

3.2.2 The broom library and augment()

The `augment()` function within the `broom` library (which you should install) puts fitted values and residuals, among other things, into a convenient data frame. Let's fit a linear model and try it out:

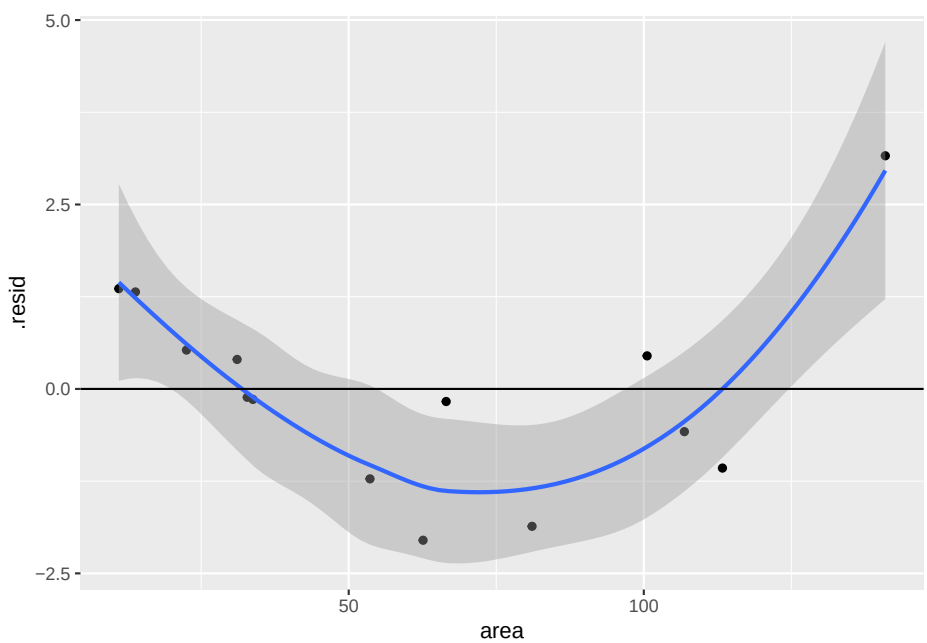
```
ganglion.lm = lm(cp.ratio ~ area, data=ganglion)
# install.packages(broom)
library(broom)
gang.lm.df = augment(ganglion.lm)
summary(gang.lm.df)
```

```
## cp.ratio area .fitted .se.fit
## Min.  : 2.560 Min.  : 11.05 Min.  : 1.200 Min.  :0.3851
## 1st Qu.: 3.441 1st Qu.: 31.52 1st Qu.: 3.397 1st Qu.:0.4386
## Median : 4.616 Median : 58.10 Median : 6.251 Median :0.5132
## Mean : 6.688 Mean : 62.18 Mean : 6.688 Mean :0.5303
## 3rd Qu.: 9.925 3rd Qu.: 95.68 3rd Qu.:10.284 3rd Qu.:0.6013
## Max.  :18.300 Max.  :140.92 Max.  :15.139 Max.  :0.8580
## .resid .hat .sigma .cooksd
## Min.  :-2.0509 Min.  :0.07144 Min.  :0.9262 Min.  :0.0004336
## 1st Qu.:-0.9493 1st Qu.:0.09286 1st Qu.:1.4356 1st
```

```
Qu.:0.0065309
## Median :-0.1277 Median :0.12714 Median :1.4770 Median
:0.0249937
## Mean : 0.0000 Mean :0.14286 Mean :1.4258 Mean :0.1871098
## 3rd Qu.: 0.5072 3rd Qu.:0.17421 3rd Qu.:1.4990 3rd
Qu.:0.0870057
## Max.  : 3.1603 Max.  :0.35459 Max.  :1.5044 Max.  :2.0477601
## .std.resid
## Min.  :-1.47721
## 1st Qu.:-0.73058
## Median :-0.09395
## Mean : 0.03741
## 3rd Qu.: 0.38008
## Max.  : 2.73029
```

We plot the residuals against the explanatory variable, then add a loess curve. If the confidence band contains the line $y = 0$, then maybe the model is fitting well.

```
ggplot(gang.lm.df, aes(x=area, y=.resid)) + geom_point() + geom_smooth() +
  geom_abline(slope=0, intercept=0)
```



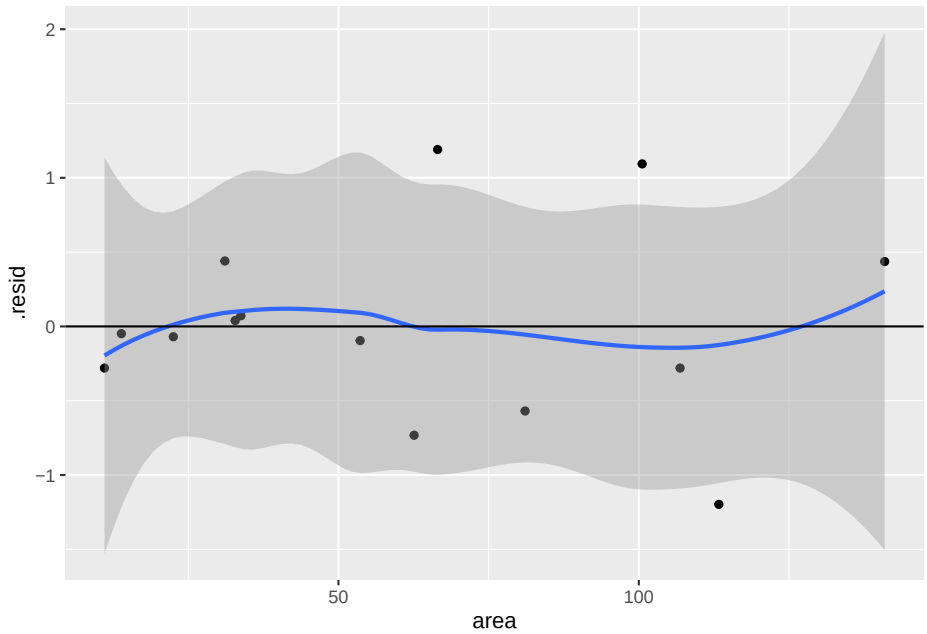
There's a clear curve in the residuals, so the linear model doesn't fit.

Let's try the quadratic:

```r
ganglion.lm2 = lm(cp.ratio ~ area + I(area^2), data=ganglion)
gang.lm2.df = augment(ganglion.lm2)
summary(gang.lm2.df)
```

```
## cp.ratio area I.area.2.  .fitted
## Min.  : 2.560 Min.  : 11.05 Min.  : 122.1 Min.  : 2.840
## 1st Qu.: 3.441 1st Qu.: 31.52 1st Qu.: 994.3 1st Qu.: 3.329
## Median : 4.616 Median : 58.10 Median : 3396.3 Median : 5.029
## Mean : 6.688 Mean : 62.18 Mean : 5430.5 Mean : 6.688
## 3rd Qu.: 9.925 3rd Qu.: 95.68 3rd Qu.: 9226.8 3rd Qu.: 9.477
## Max.  :18.300 Max.  :140.92 Max.  :19857.9 Max.  :17.863
## .se.fit .resid .hat .sigma
## Min.  :0.2346 Min.  :-1.19758 Min.  :0.1107 Min.  :0.6081
## 1st Qu.:0.2683 1st Qu.:-0.28029 1st Qu.:0.1448 1st Qu.:0.6907
## Median :0.2860 Median :-0.05893 Median :0.1644 Median :0.7283
## Mean :0.3130 Mean : 0.00000 Mean :0.2143 Mean :0.7030
## 3rd Qu.:0.3051 3rd Qu.: 0.34532 3rd Qu.:0.1872 3rd Qu.:0.7393
## Max.  :0.6052 Max.  : 1.19049 Max.  :0.7363 Max.  :0.7396
## .cooksd .std.resid
## Min.  :0.0001503 Min.  :-1.88857
## 1st Qu.:0.0008978 1st Qu.:-0.47249
## Median :0.0288280 Median :-0.09404
## Mean :0.1611383 Mean : 0.02964
## 3rd Qu.:0.1569547 3rd Qu.: 0.52490
## Max.  :1.3513989 Max.  : 1.84998
```

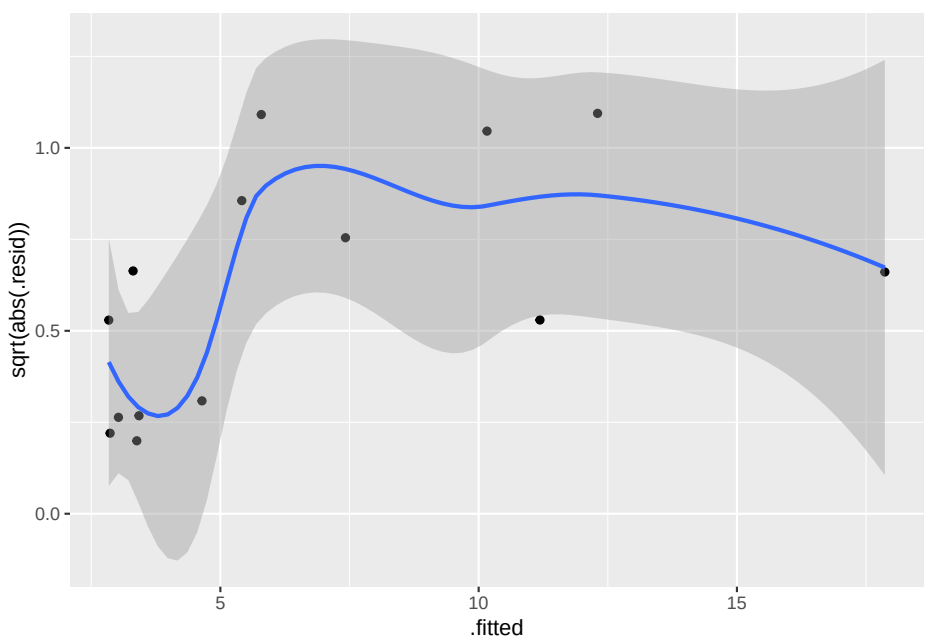
```r
ggplot(gang.lm2.df, aes(x=area, y=.resid)) + geom_point() + geom_smooth() +
  geom_abline(slope=0, intercept=0)
```



The curve for the residuals just wiggles around 0. It's plausible that the quadratic model is correctly specified.

In addition, we check for homoscedasticity using a spread-location plot.

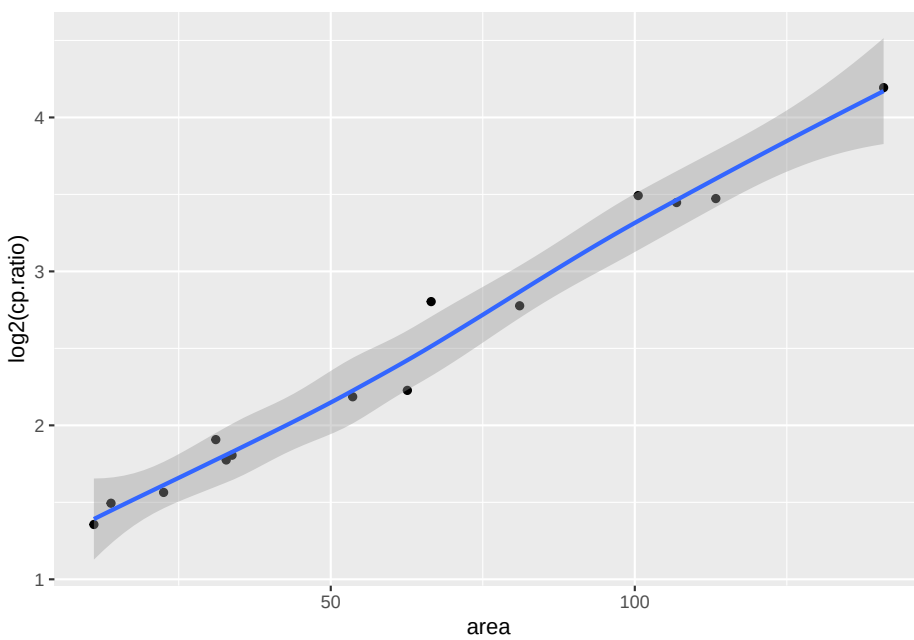
```
ggplot(gang.lm2.df, aes(x=.fitted, y=sqrt(abs(.resid)))) + geom_point() + geom_smooth()
```



It doesn't look like a horizontal line describes the transformed residuals well. The spread of the residuals changes with the fitted values, meaning there's heteroscedasticity. This makes the quadratic model less appealing – maybe a transformation would be better.

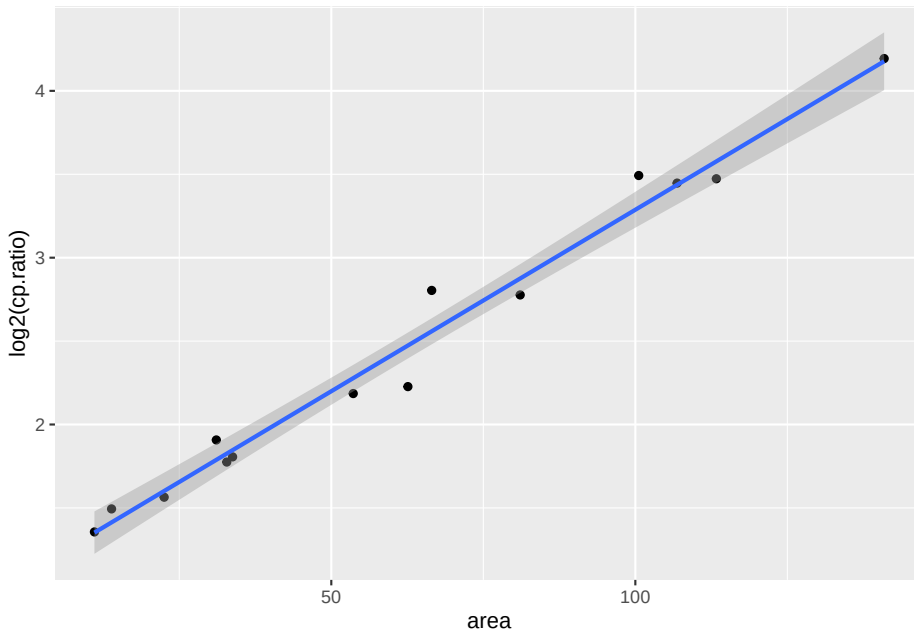
Let's try taking the log of the CP ratio. Add a loess fit:

```
ggplot(ganglion, aes(x=area, y=log2(cp.ratio))) + geom_point() + geom_smooth()
```



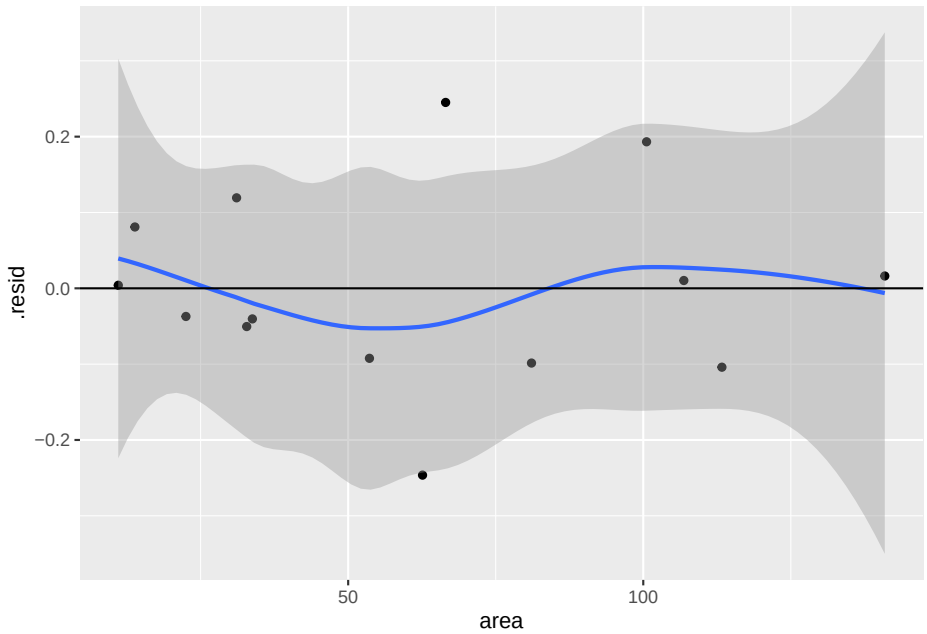
Looks like a straight line would do just as good a job. Let's use `lm()` to fit the model instead.

```
ggplot(ganglion, aes(x=area, y=log2(cp.ratio))) + geom_point() + geom_smooth(method =
```



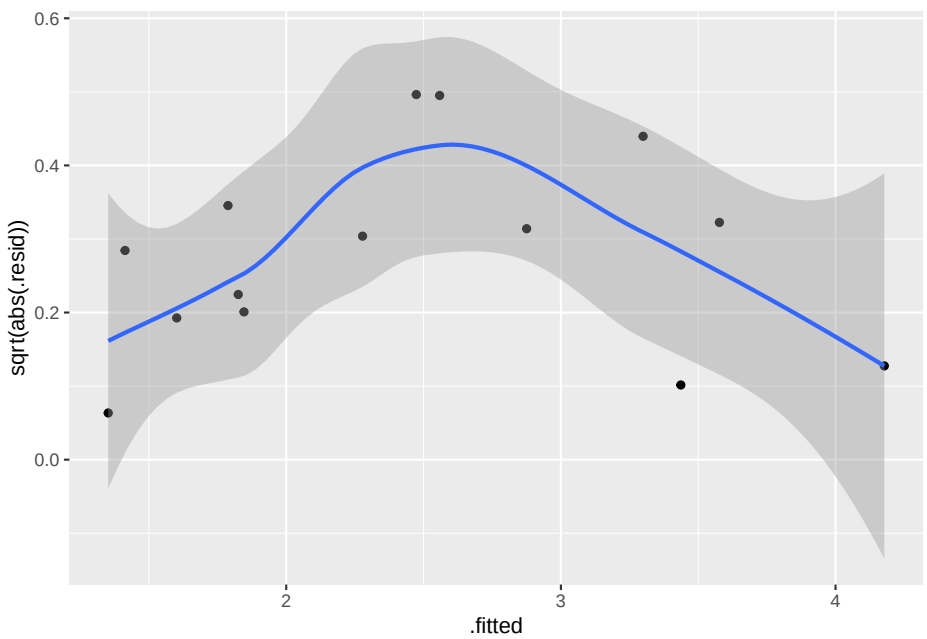
The straight line produced by `lm()` looks more or less the same as the loess fit. So far, so good. To get a better look, plot the residuals agaisnt `area`, the x-variable.

```
ganglion.log.lm = lm(log2(cp.ratio) ~ area, data=ganglion)
gang.log.lm.df = augment(ganglion.log.lm)
ggplot(gang.log.lm.df, aes(x=area, y=.resid)) + geom_point() + geom_smooth() +
  geom_abline(slope=0, intercept=0)
```



The curve just wiggles around zero. The residuals look like noise. Now check for homoscedasticity using a spread-location plot:

```
ggplot(gang.log.lm.df, aes(x=.fitted, y=sqrt(abs(.resid)))) + geom_point() + geom_smoo
```



This time we see a horizontal line goes through the confidence band, so ho-

moscedasticity is a reasonable assumption. (The blue line is curved, but that could just be because of the small amount of data.)

Taking the log of `cp.ratio` has led to an elegant model: linear and homoscedastic. But how much does the model actually explain? Instead of drawing a residual-fit plot, just calculate R^2:

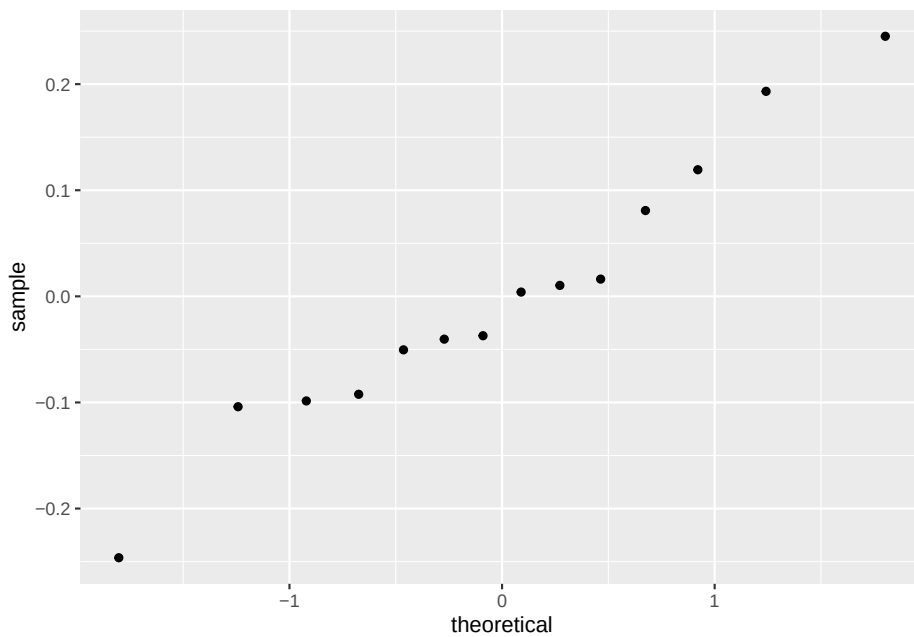
```
var(gang.log.lm.df$.fitted) / var(log2(ganglion$cp.ratio))
```

```
## [1] 0.9797802
```

The model captures 98% of the variance in the data.

Finally, we check if the residuals look normal. If so, this would be good, but it wouldn't necessarily be a dealbreaker if they weren't.

```
ggplot(gang.log.lm.df, aes(sample=.resid)) + stat_qq()
```



They're normal! Great! By taking logs, we get data that fulfills all the usual linear model assumptions, and can do formal inference if we wish.

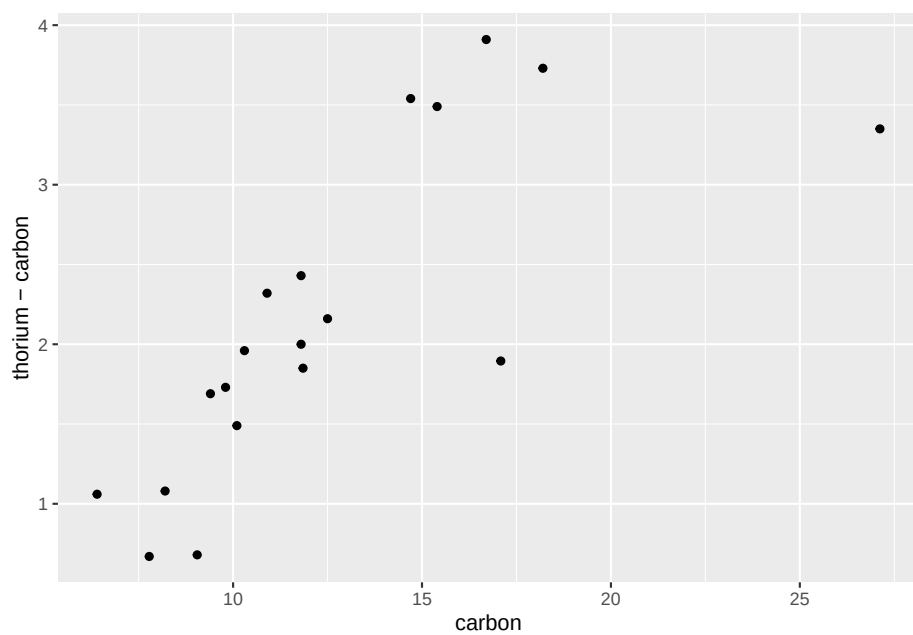
3.3 Robust modeling

READ: Cleveland pp. 110–134.

3.3.1 Example: Carbon dating

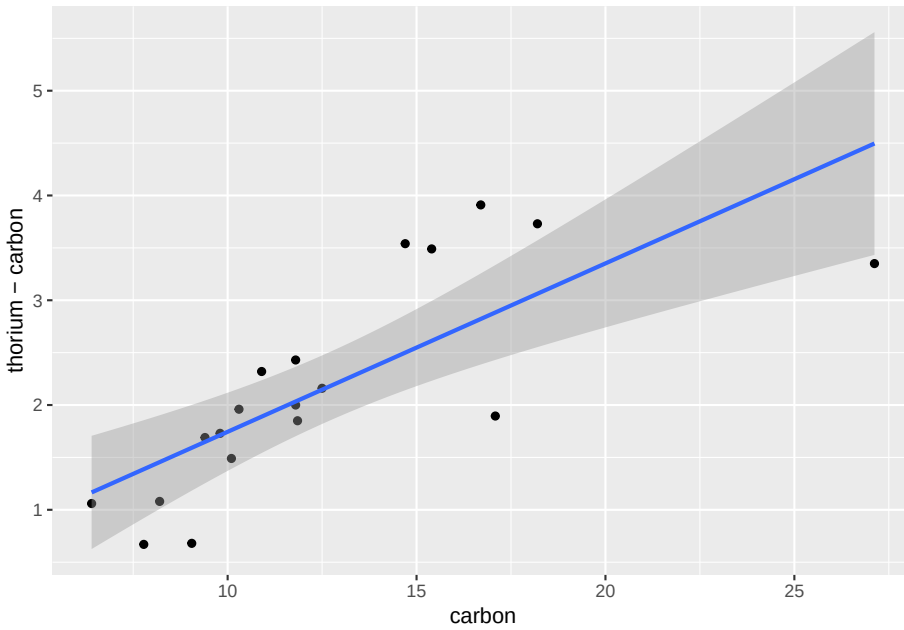
The data set `dating` contains paired observations giving the estimated ages of 19 coral samples in thousands of years using both `carbon` dating (the traditional method) and `thorium` dating (a modern and purportedly more accurate method.) What's the difference between these two methods?

```
load("lattice.RData")
library(ggplot2)
ggplot(dating, aes(x=carbon, y=thorium-carbon)) + geom_point()
```



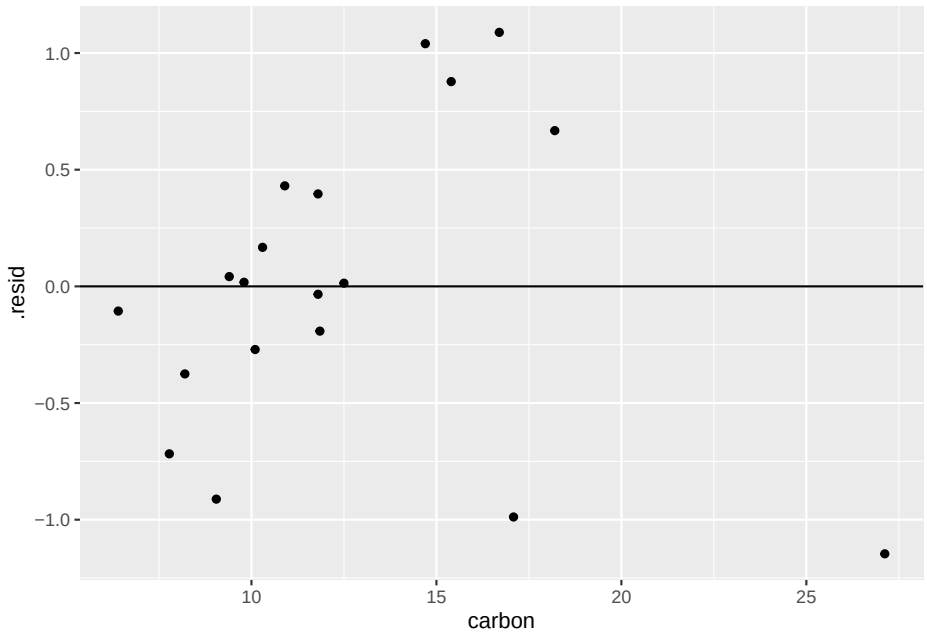
The first thing to note about this graph is that the y-values are always above zero, meaning thorium age is always above zero. The second thing to note is this difference increases with carbon age. Try adding a straight line to the plot:

```
ggplot(dating, aes(x=carbon, y=thorium-carbon)) + geom_point() + geom_smooth(method="lm
```



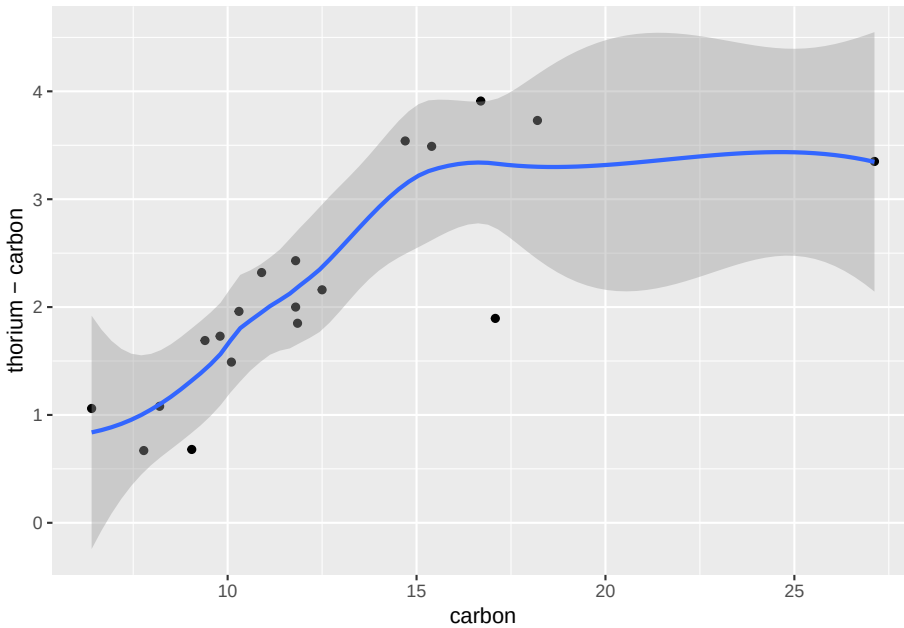
The line doesn't seem like it's doing a good job of describing the bivariate relationship for most of the data – in particular, a lot of points are well above the line. To see why, plot the residuals.

```
dating.lm = lm(thorium - carbon ~ carbon, data=dating)
library(broom)
dating.lm.df = augment(dating.lm)
ggplot(dating.lm.df, aes(x=carbon, y=.resid)) + geom_point() + geom_abline(slope=0)
```



The issue is that the extreme value on the right-hand side is dragging the whole line down. Recall that outliers can have a large effect on a regression line. We could try using loess instead of a linear model, but this still isn't very satisfactory:

```
ggplot(dating, aes(x=carbon, y=thorium-carbon)) + geom_point() + geom_smooth()
```

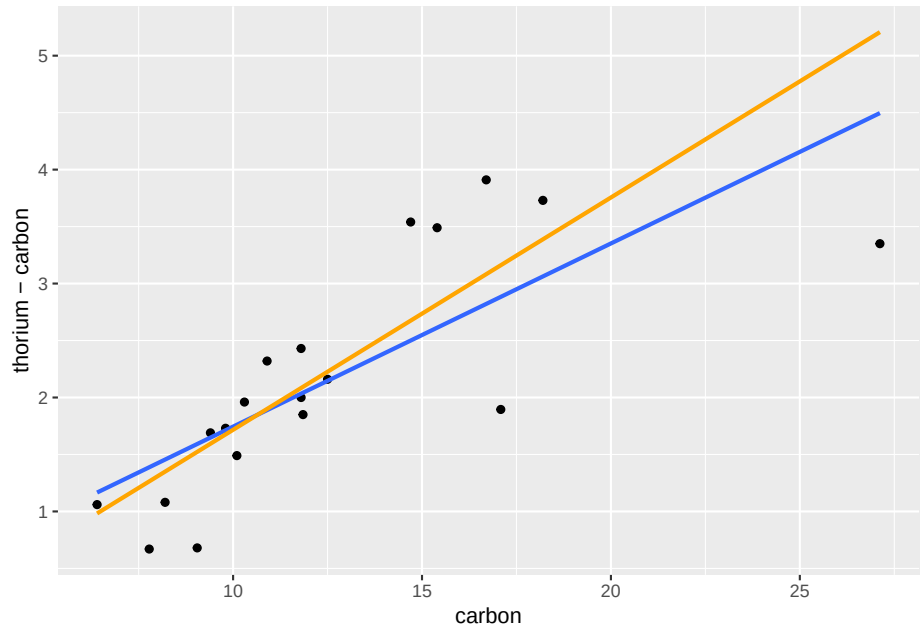


The loess curve is still missing the group of four observations where the difference is about 3.5 to 4 thousand years, possibly because the observation at $(17, 1.9)$ is also something of an outlier.

A better solution is to use a *robust* method to determine the best-fitting line, instead of least squares. The `rlm()` function in the `MASS` library (which should be installed by default) can fit such models, and can be called from within `geom_smooth()`. Let's see if it works just using the defaults:

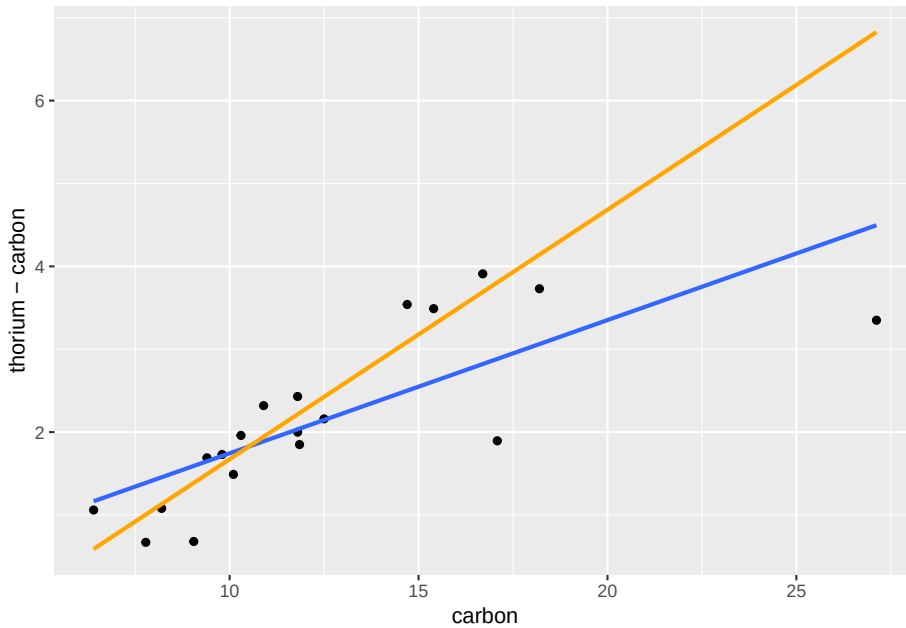
```
library(MASS)
ggplot(dating, aes(x=carbon, y=thorium-carbon)) + geom_point() +
  geom_smooth(method="lm", se=FALSE) + geom_smooth(method="rlm", se=FALSE, col="orange")
```

```
## Warning in rlm.default(x, y, weights, method = method, wt.method = wt.method, :
## 'rlm' failed to converge in 20 steps
```



Not only do we get an error message, it still seems like the robust (orange) line is pulled down a bit too much by the two low-lying points. We can either increase the number of iterations (`maxit`) or try a different kind of fit. Tukey's **bisquare** method (sometimes called **biweight**) can be helpful when there are convergence problems, as it downweights extreme outliers to have zero weight.

```
gg = ggplot(dating, aes(x=carbon, y=thorium-carbon)) + geom_point() +
  geom_smooth(method="lm", se=FALSE)
gg + geom_smooth(method="rlm", se=FALSE, col="orange", method.args=list(psi=psi.bisquar
```

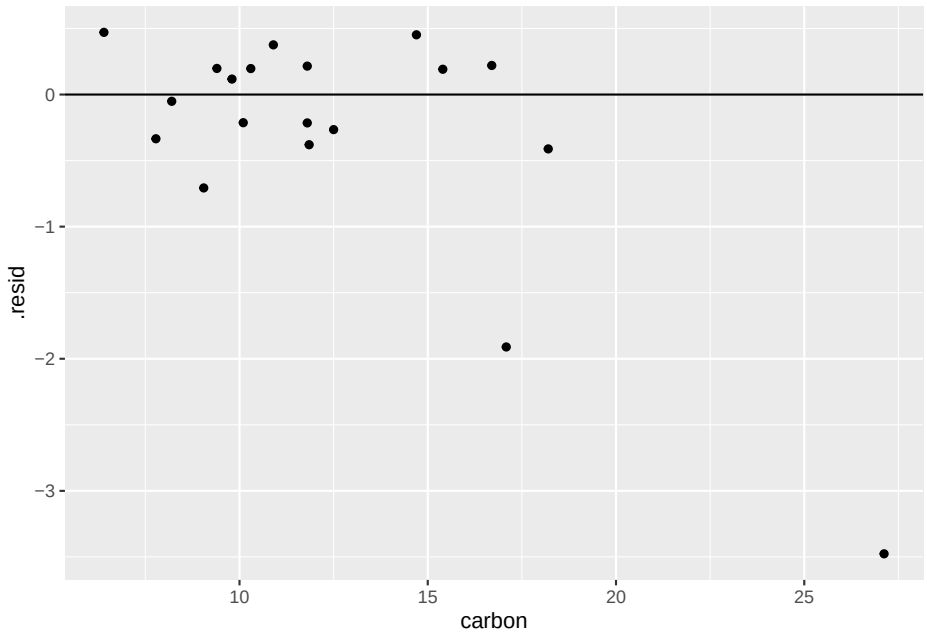


Now the orange line completely misses the two extreme outliers, but it fits all the rest of the data much better. Looking at the residuals makes this obvious:

```
age.diff = dating$thorium - dating$carbon
carbon = dating$carbon
dating.rlm = rlm(age.diff ~ carbon, psi = psi.bisquare)
tidy(dating.rlm)
```

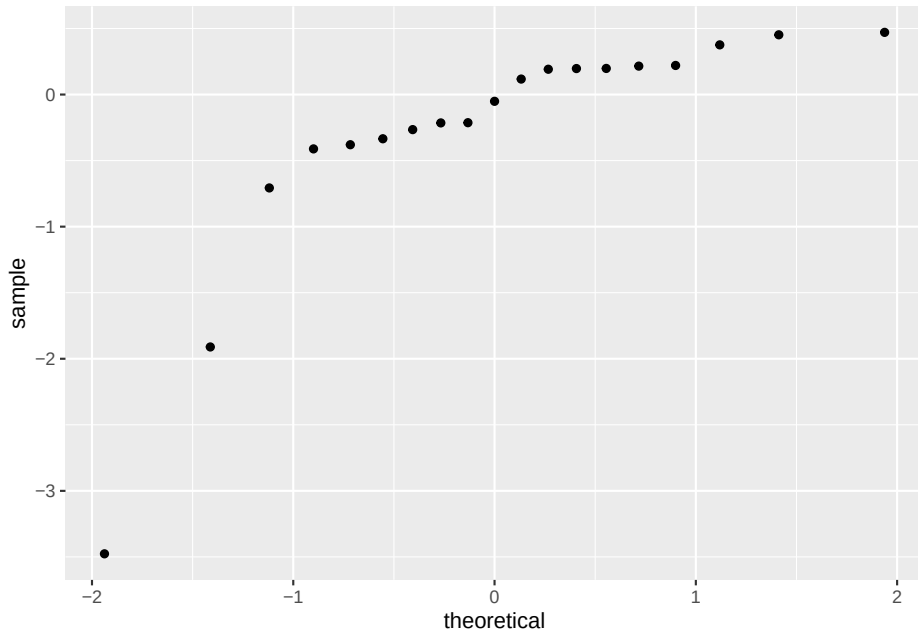
```
## # A tibble: 2 x 4
##   term        estimate std.error statistic
##   <chr>          <dbl>     <dbl>     <dbl>
## 1 (Intercept)   -1.34     0.266      -5.02
## 2 carbon         0.301    0.0198     15.2
```

```
dating.rlm.df = augment(dating.rlm)
ggplot(dating.rlm.df, aes(x=carbon, y=.resid)) + geom_point() + geom_abline(slope=0)
```



Most of the residuals are scattered around zero, with the two exceptions being very large negative values. By using this robust method, we've managed to get a model that provides a very good fit for 17 out of 19 observations, though we must always keep in mind there will be some observations that don't fit the pattern. In addition, formal inference will be somewhat difficult. Look at the normal QQ plot of the residuals:

```
ggplot(dating.rlm.df, aes(sample=.resid)) + stat_qq()
```



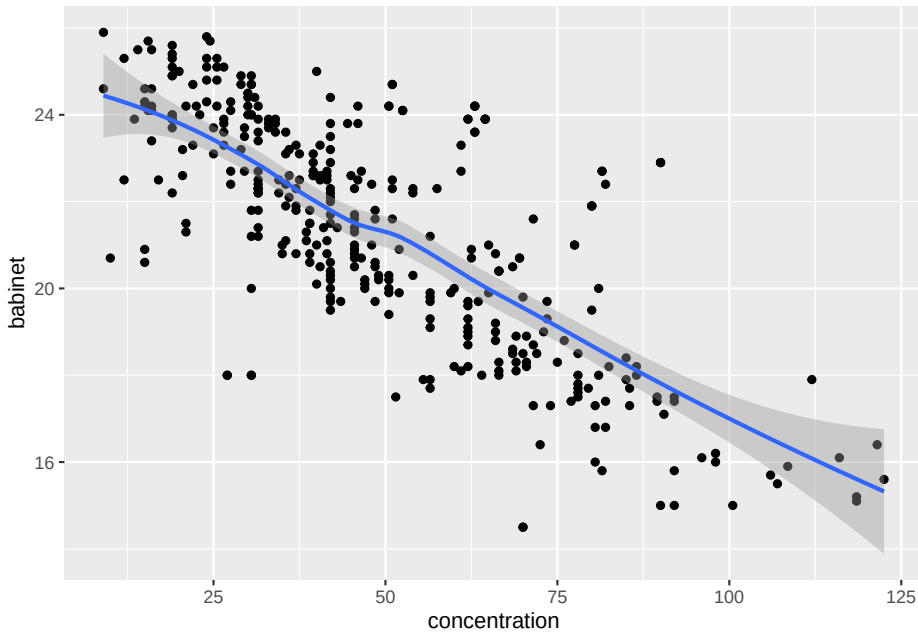
The outliers prevent the residuals from being well-modelled by a normal distribution, or even a symmetric one. We should be very hesitant to make probabilistic predictions, but this is EDA so we don't have to.

3.3.2 Particulates and the Babinet point

Light is *polarized* when the waves oscillate in a plane and *unpolarized* when they vibrate in all directions. Sunlight becomes unpolarized when its angle is greater than the *Babinet point*. The Babinet point depends on the concentration of particulates in the air.

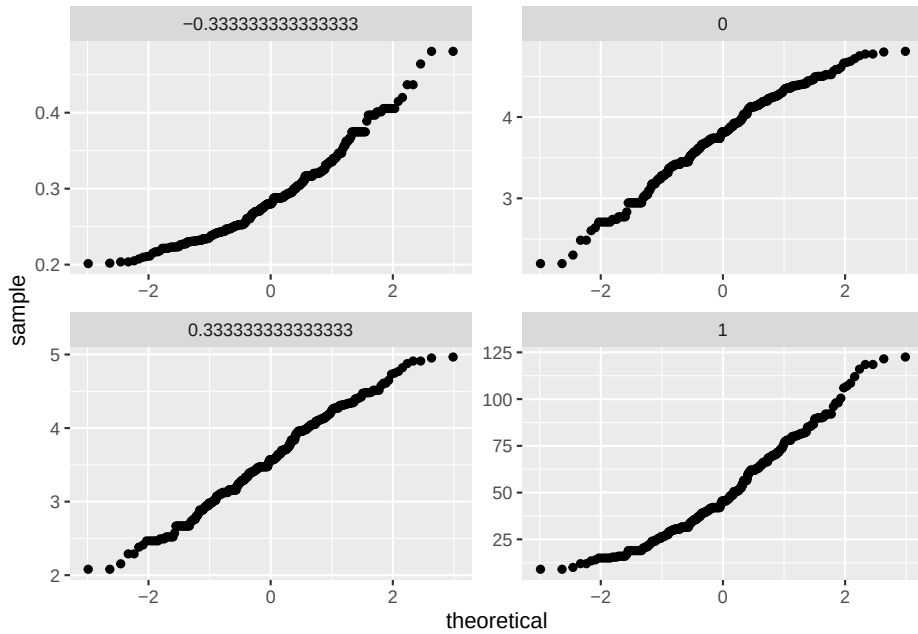
The `polarization` data set in the `lattice.RData` workspace contains 355 observations from an experiment to find the Babinet point (in degrees) while varying the particulate concentration (in micrograms per cubic meter.)

```
ggplot(polarization, aes(x=concentration, y=babinet)) + geom_point() + geom_smooth()
```



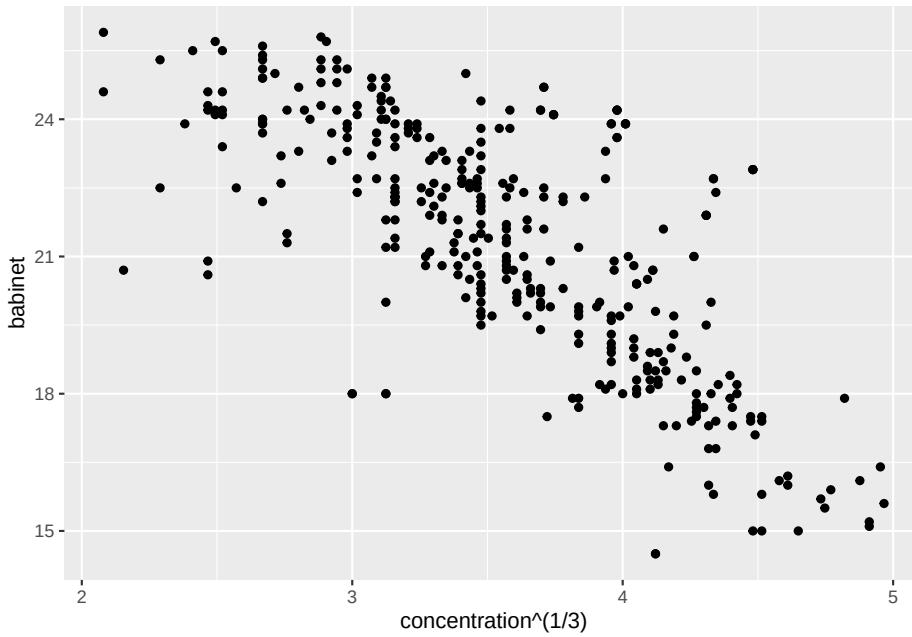
There's clearly a decreasing relationship; it's plausibly linear but there might be a bend in the bottom that the fitted curve isn't picking up. Note that because of rounding, some points are plotted on top of each other, so later we'll jitter the points a bit. First, we'll think about transformations. Fractional powers are usually uninterpretable, but Cleveland argues that because concentration has to do with volume, it's reasonable to consider cube root transformations. We can compare powers of $1/3, -1/3$, and -1 to the untransformed data in terms of normality.

```
n = nrow(polarization)
power = rep(c(1,1/3,0,-1/3), each=n)
concentration = polarization$concentration
conc.trans = c(concentration, concentration^(1/3), log(concentration), concentration^(
ggplot(data.frame(power, conc.trans), aes(sample=conc.trans)) + stat_qq() +
  facet_wrap(~power, scales="free")
```



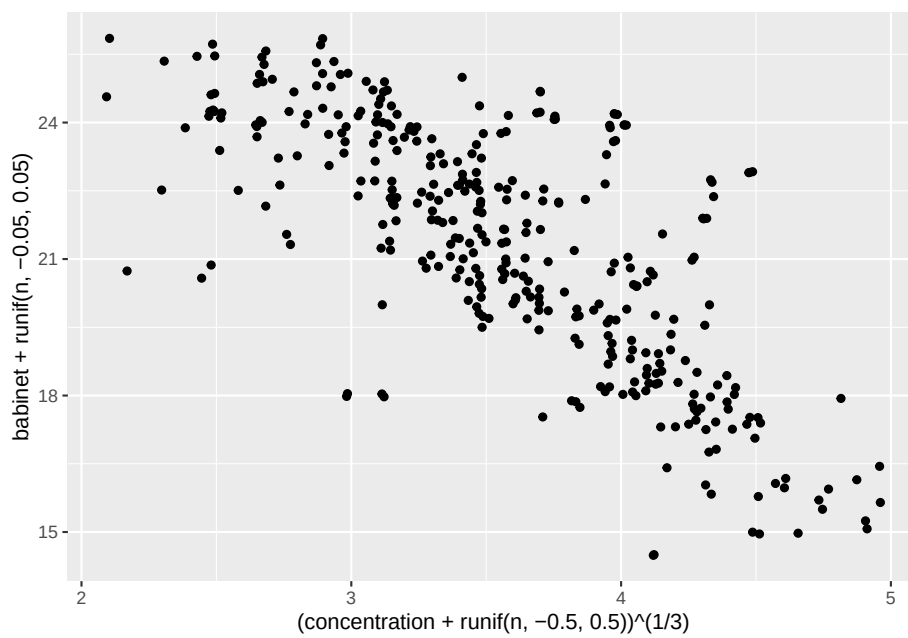
The cube root transformation comes closest to normality. (The tails might be a bit short, but this seems unimportant.) Honestly I don't think you get that much out of transformation and would just work with the raw data, but I'll follow Cleveland and run with the cube root transform. How does this look on a scatterplot?

```
ggplot(polarization, aes(x=concentration^(1/3), y=babinet)) + geom_point()
```



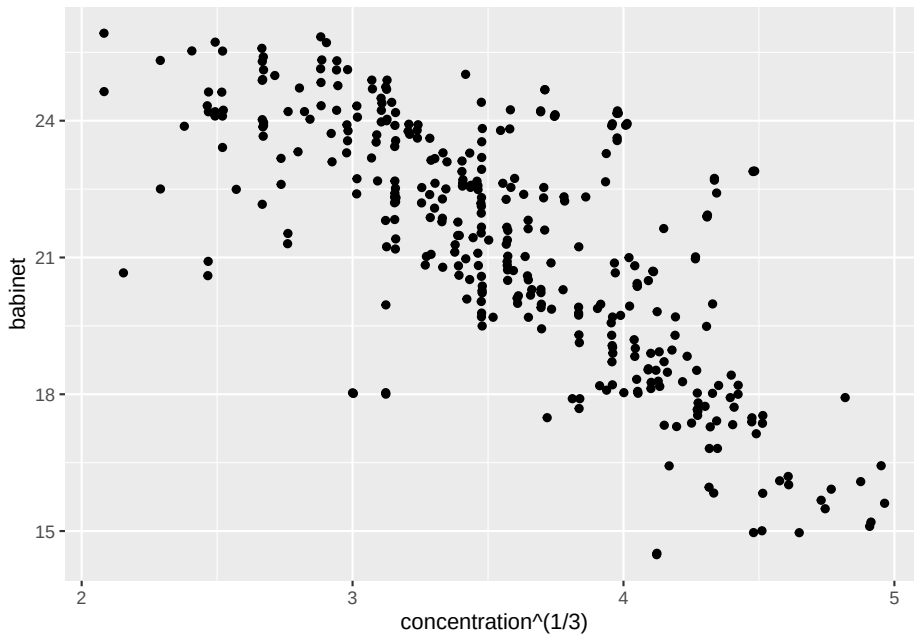
We still have the problem of repeated x- and y-values, so we jitter. We could "unround" the data by adding random noise manually:

```
ggplot(polarization, aes(x=(concentration+runif(n, -0.5, 0.5))^(1/3),
  y=babinet+runif(n, -0.05, 0.05))) + geom_point()
```



Or we could use the automatic `geom_jitter()` function:

```
ggplot(polarization, aes(x=concentration^(1/3), y=babinet)) + geom_jitter()
```



The manual jittering arguably looks a little better, but really it doesn't make much difference.

Let's now fit a **loess** curve to the data. Loess is a form of *local polynomial regression*, meaning that at every x-value, it fits a weighted polynomial model "locally": data at nearby x-values will be weighted heavily, while data at far away x-value will be downweighted or not considered at all if they fall outside a "neighborhood." See a reliable nonparametric statistics text, or failing that, Luen's somewhat less relaible *Some of Nonparametric Statistics* for details. Loess performs well for moderate sized data sets. For large data sets (thousands or millions of observations), loess may require too much memory, in which case alternative smoothing methods (most commonly **splines**) can be used.

There are three main arguments to play around with:

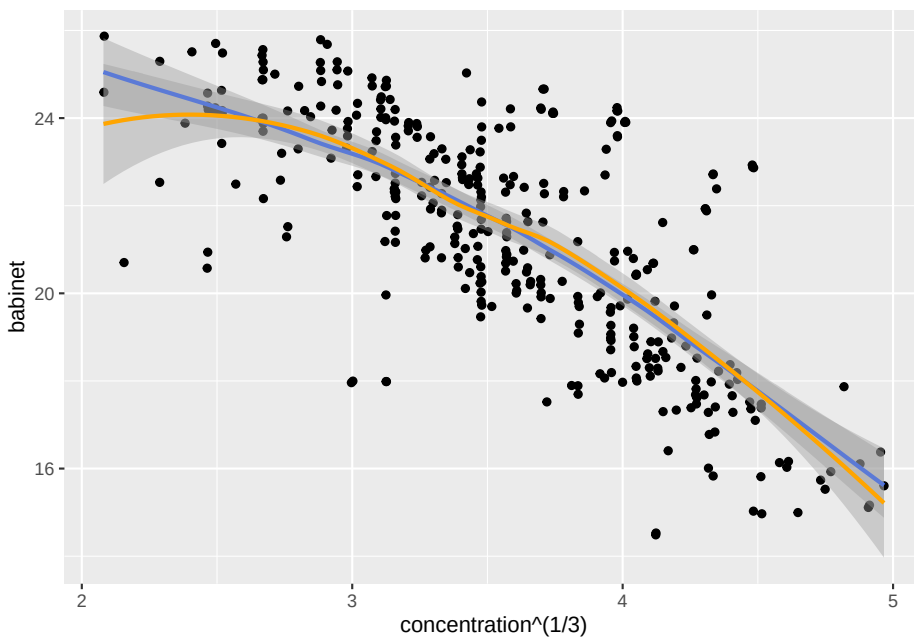
- `span`: The proportion of the data included in the neighborhood. Default is 0.75.
- `degree`: The degree of polynomial fitted. Default is 2 (locally quadratic) but for data that isn't too wiggly, degree 1 (locally linear) may be less likely to do weird stuff at the extremes.
- `family`: The default, `gaussian`, uses (weighted) least squares to fit the local polynomial. `symmetric` uses bisquare to be more resistant to outliers.

If you're optimizing for prediction, then the tuning parameters should be found

using cross-validation or some similar method. If you're doing EDA, playing around with the parameters until you get something that looks good is fine. Generally the default `span` works surprisingly adequately for non-weird data sets, while it's often worth trying both degrees 1 and 2 and visually determining which appears to give the better fit. It's also hard to say if `symmetric` definitely does better than least squares unless there are obvious outlier problems.

Going back to the polarization data, let's compare the degree 1 and 2 fits.

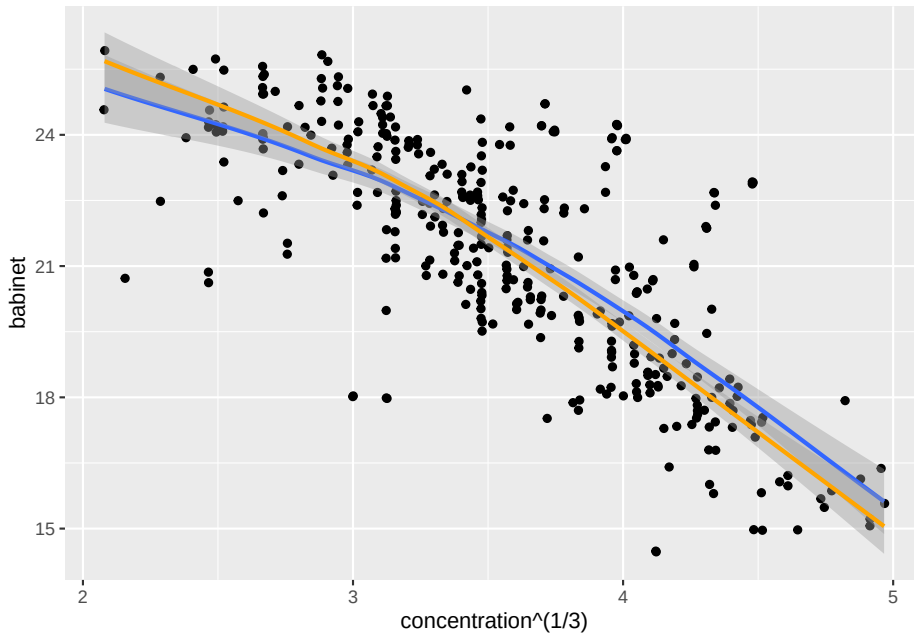
```
ggplot(polarization, aes(x=concentration^(1/3), y=babinet)) +
geom_jitter() + geom_smooth(method.args=list(degree=1)) +
  geom_smooth(method.args=list(degree=2), col="orange")
```



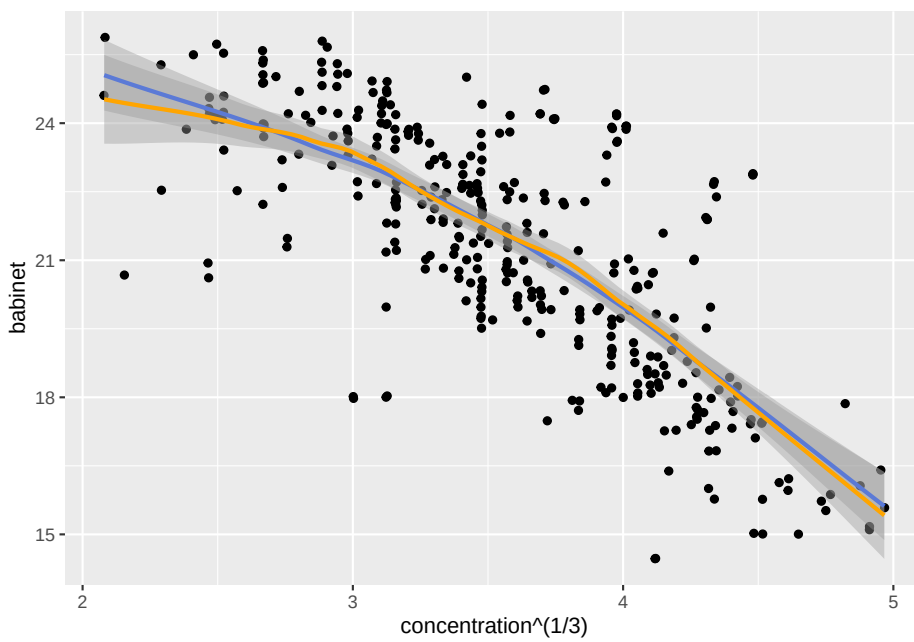
The locally linear fit is in blue, while the locally quadratic fit is in orange. We see the orange fit is bent strong on the left hand side, but there isn't much data to justify this. Subjectively, I'd argue for the degree 1 fit.

Other stuff to try:

```
ggplot(polarization, aes(x=concentration^(1/3), y=babinet)) + geom_jitter() +
  geom_smooth(method.args=list(degree=1)) + geom_smooth(method.args=list(degree=1, fami
```



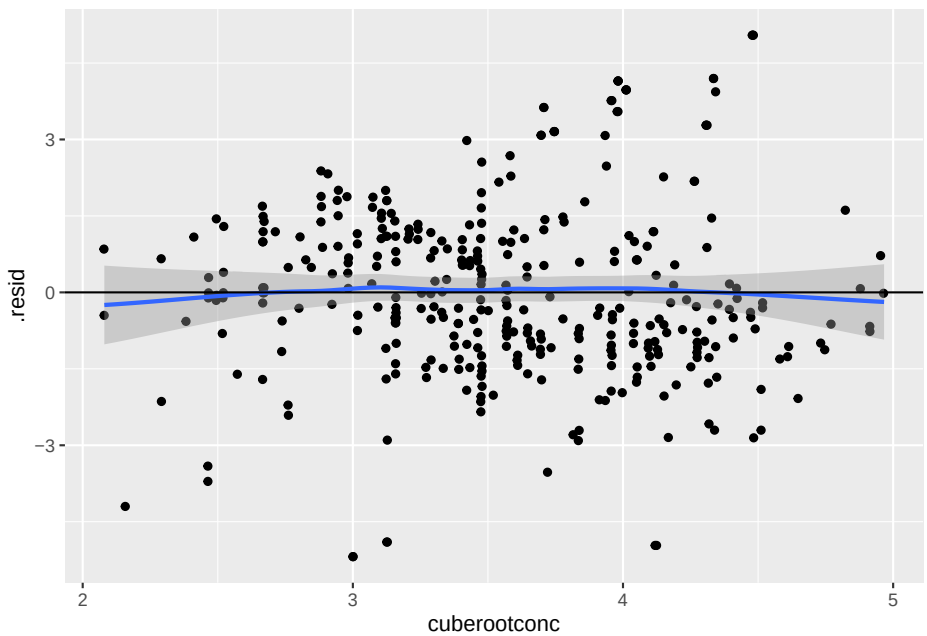

```
gg = ggplot(polarization, aes(x=concentration^(1/3), y=babinet)) + geom_jitter()
gg + geom_smooth(method.args=list(degree=1)) +
  geom_smooth(span=0.5, method.args=list(degree=1), col="orange")
```



There's no single right answer here. Let's try a degree 1 fit and leave everything

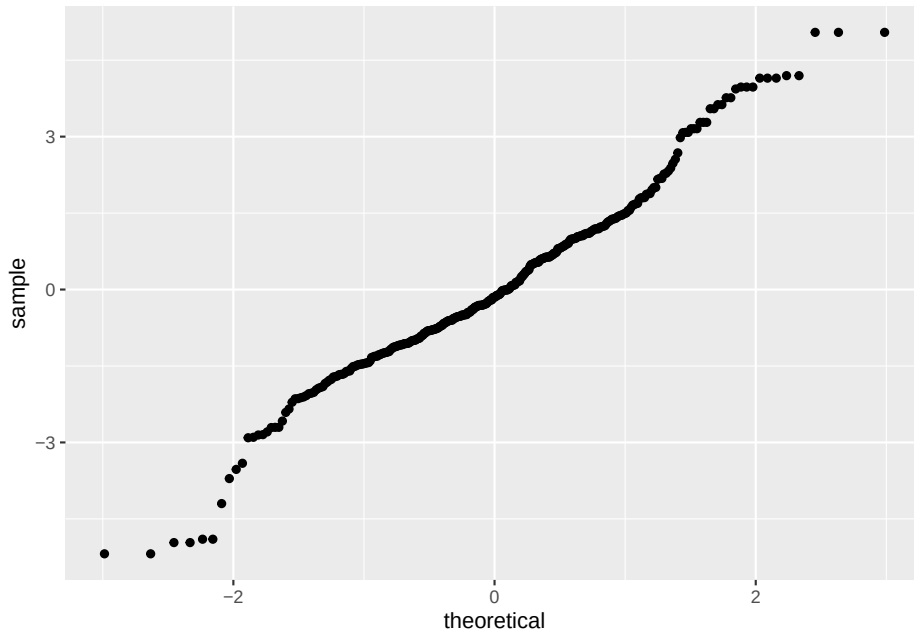
else at its default. Now refit the model and plot the residuals with another loess fit. For residual plots, a degree 1 loess fit is often more appropriate than a quadratic:

```
cuberootconc = (polarization$concentration)^(1/3)
babinet = polarization$babinet
polar.lo = loess(babinet ~ cuberootconc, degree=1)
polar.lo.df = augment(polar.lo)
ggplot(polar.lo.df, aes(x=cuberootconc, y=.resid)) + geom_jitter(height=0) +
  geom_smooth(method.args=list(degree=1)) + geom_abline(slope=0)
```



It doesn't look like there's any trend. The fact that there are quite a few large positive residuals for cube root concentration between 3.7 and 4.5 is a little bit worrying; we'll take a closer look at this later. Before that, we'll check normality:

```
ggplot(polar.lo.df, aes(sample=.resid)) + stat_qq()
```



The normal distribution breaks down a bit at the edges, but this isn't a deal-breaker. Calculate R^2:

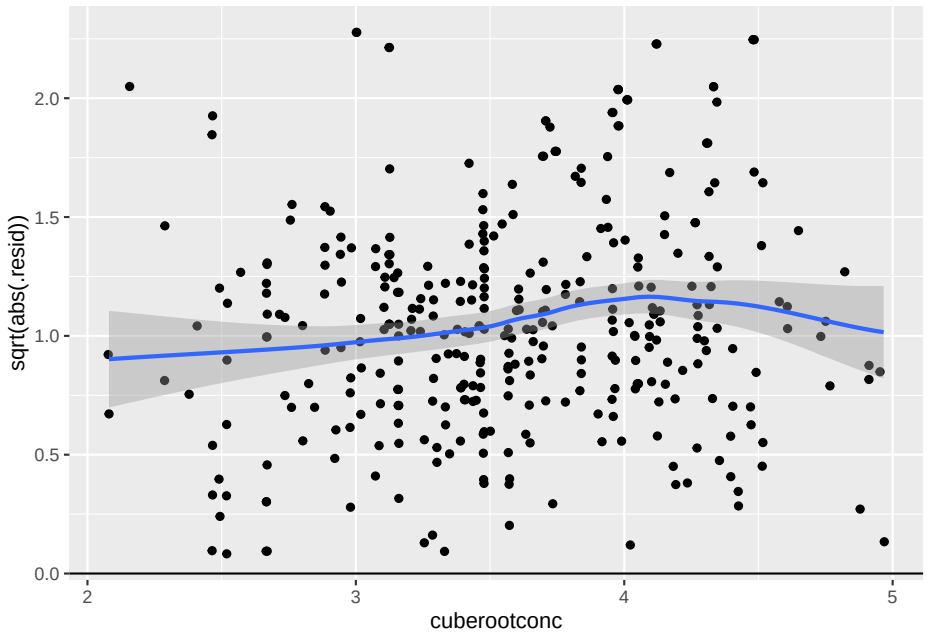
```
var(polar.lo.df$.fitted) / var(babinet)
```

```
## [1] 0.5672304
```

The model captures 57% of the variance of the babinet variable.

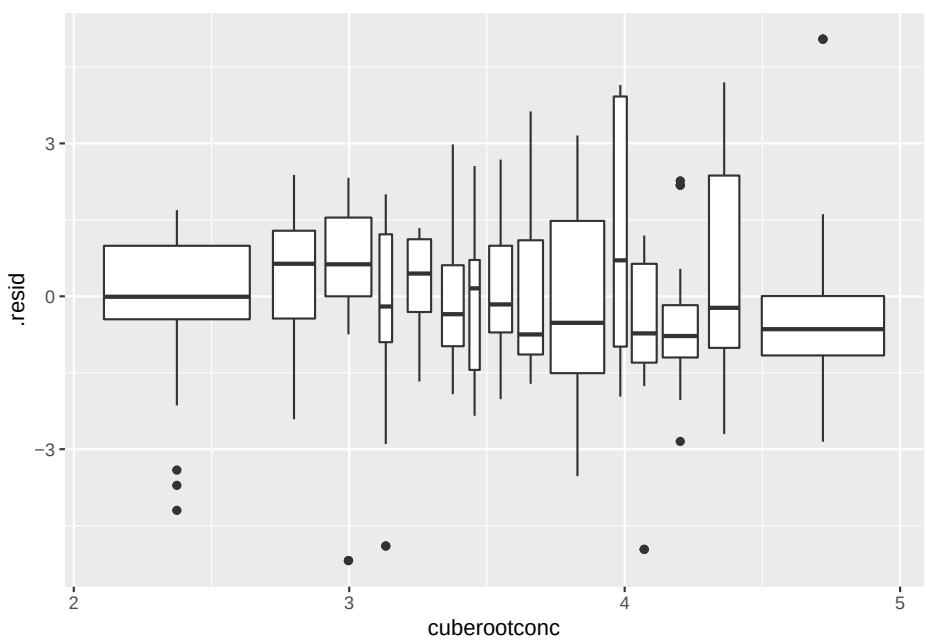
To return to the earlier issue, we want to see if the distribution of the residuals changes with (cube root of) concentration. We can plot root absolute residuals:

```
ggplot(polar.lo.df, aes(x = cuberootconc, y = sqrt(abs(.resid)))) + geom_jitter(height = 0) +
  geom_smooth(method.args = list(degree = 1)) + geom_abline(slope = 0)
```



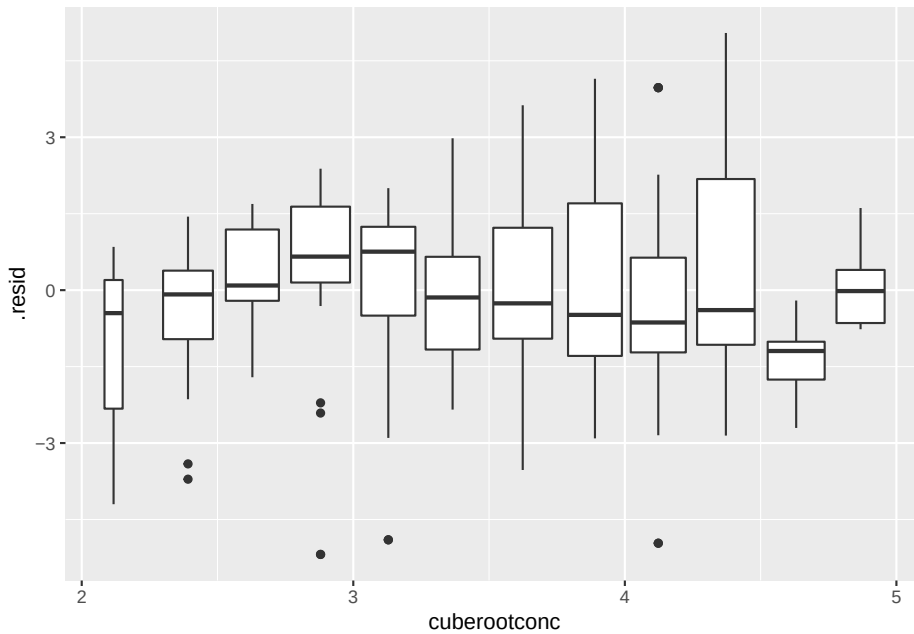
Boxplots are an alternative. We can "cut" the range of x-values into a given number of intervals, each with (approximately) equal numbers of observations:

```
polar.gg = ggplot(polar.lo.df, aes(x=cuberootconc, y=.resid))
polar.gg + geom_boxplot(aes(group = cut_number(cuberootconc, n=15)))
```



Or we can specify the x-axis bins explicitly:

```
polar.gg + geom_boxplot(aes(group = cut(cuberootconc, breaks=seq(2, 5, 0.25))))
```



Either way, it seems like the boxes certainly change as we go from left to right, and the change seems quite complicated (it's not merely heteroskedasticity.) So we've succeeded in modeling the trend, but it's much harder to draw probabilistic inferences. If we wanted to do this, the only good solution with such complex relationships is to get more data.

3.4 Comparing bivariate distributions

In this section we'll describe the relationship between GDP and life expectancy by continent, using data for countries in 2007. To get the data, (install and) load the `gapminder` package:

```
library(gapminder)
summary(gapminder)
```

```
## country continent year lifeExp
## Afghanistan: 12 Africa :624 Min.   :1952 Min.   :23.60
## Albania : 12 Americas:300 1st Qu.:1966 1st Qu.:48.20
## Algeria : 12 Asia :396 Median :1980 Median :60.71
## Angola : 12 Europe :360 Mean :1980 Mean :59.47
## Argentina : 12 Oceania : 24 3rd Qu.:1993 3rd Qu.:70.85
## Australia : 12 Max.   :2007 Max.   :82.60
```

```
## (Other) :1632
## pop gdpPercap
## Min.  :6.001e+04 Min.  : 241.2
## 1st Qu.:2.794e+06 1st Qu.: 1202.1
## Median :7.024e+06 Median : 3531.8
## Mean :2.960e+07 Mean : 7215.3
## 3rd Qu.:1.959e+07 3rd Qu.: 9325.5
## Max.  :1.319e+09 Max.  :113523.1
##
```

Use `subset()` to pick out the 2007 data:

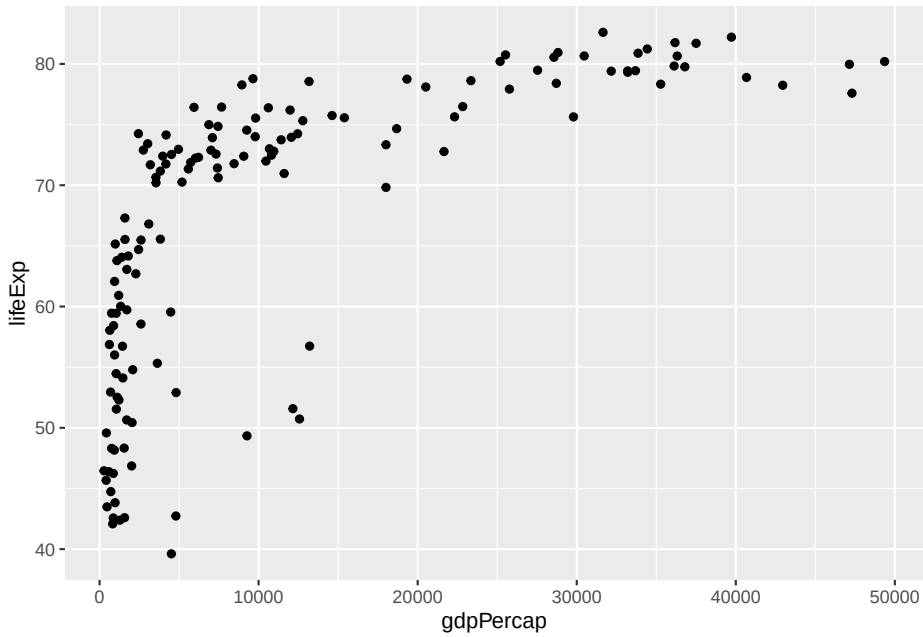
```
gapminder07 = subset(gapminder, year == 2007)
summary(gapminder07)
```

```
## country continent year lifeExp
## Afghanistan: 1 Africa :52 Min.  :2007 Min.  :39.61
## Albania : 1 Americas:25 1st Qu.:2007 1st Qu.:57.16
## Algeria : 1 Asia :33 Median :2007 Median :71.94
## Angola : 1 Europe :30 Mean :2007 Mean :67.01
## Argentina : 1 Oceania : 2 3rd Qu.:2007 3rd Qu.:76.41
## Australia : 1 Max.  :2007 Max.  :82.60
## (Other) :136
## pop gdpPercap
## Min.  :1.996e+05 Min.  : 277.6
## 1st Qu.:4.508e+06 1st Qu.: 1624.8
## Median :1.052e+07 Median : 6124.4
## Mean :4.402e+07 Mean :11680.1
## 3rd Qu.:3.121e+07 3rd Qu.:18008.8
## Max.  :1.319e+09 Max.  :49357.2
##
```

3.4.1 Plotting with colors

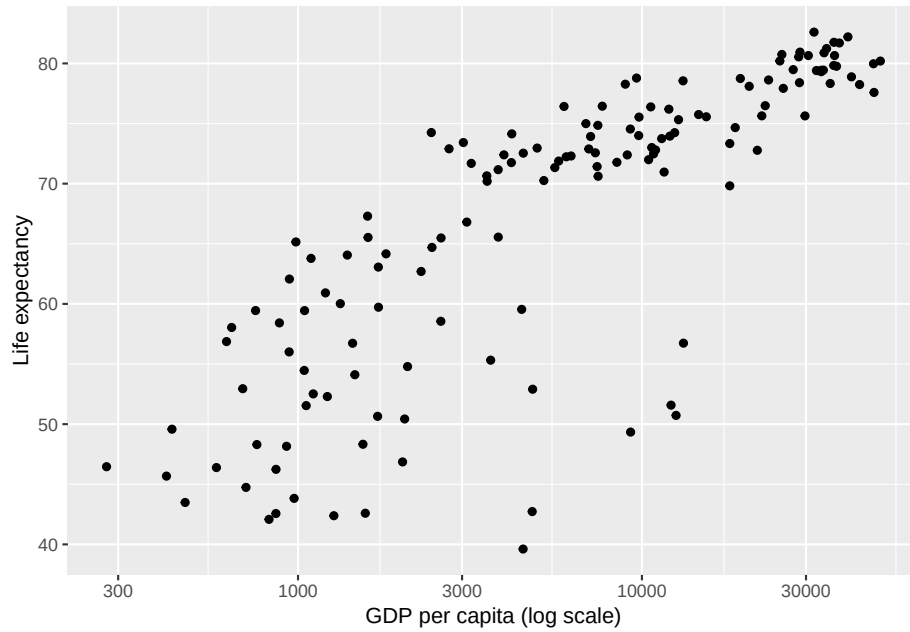
Draw a scatterplot of population against GDP per capita:

```
library(ggplot2)
ggplot(gapminder07, aes(x = gdpPercap, y = lifeExp)) + geom_point()
```



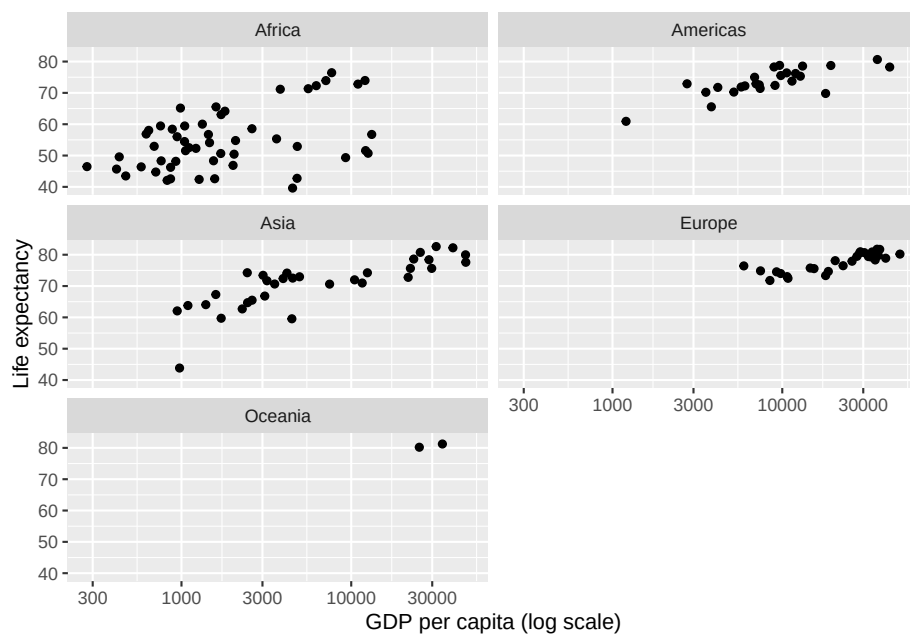
The data is concentrated on the left, suggesting a log transformation of the x-variablee:

```
ggplot(gapminder07, aes(x = gdpPercap, y = lifeExp)) + geom_point() + scale_x_log10() +
  xlab("GDP per capita (log scale)") + ylab("Life expectancy") +
  labs("Life expectancy by country, 2007")
```



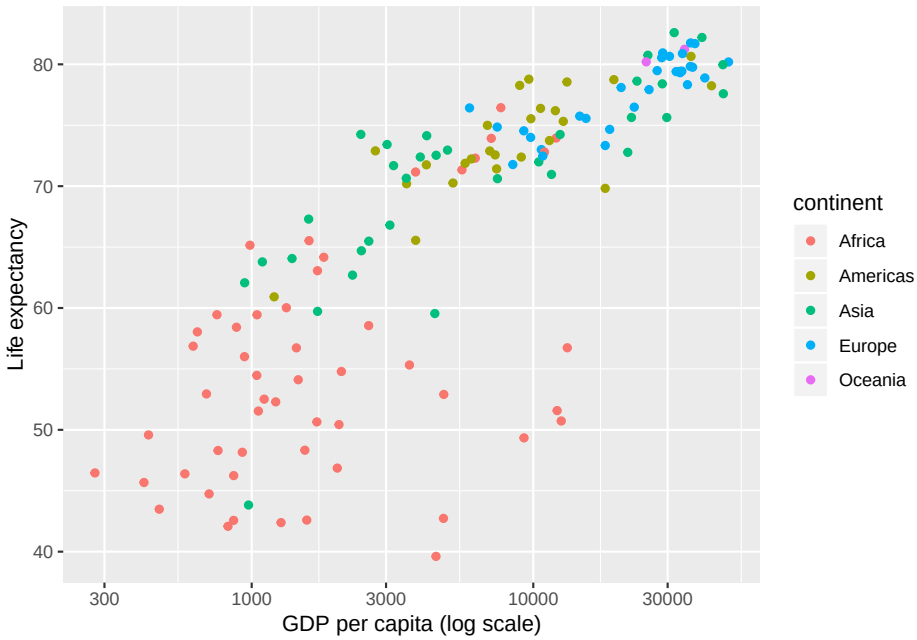
How do we include the `continent` information in the graph? We could facet:

```
ggplot(gapminder07, aes(x = gdpPercap, y = lifeExp)) + geom_point() + scale_x_log10() +
  xlab("GDP per capita (log scale)") + ylab("Life expectancy") +
  labs("Life expectancy by country, 2007") + facet_wrap(~ continent, ncol = 2)
```



Or we could color-code the continents. A variable can be mapped to `color`, either inside the `aes()` (if you want it to apply to everything) or in a new `aes()` inside `geom_point()`:

```
ggplot(gapminder07, aes(x = gdpPercap, y = lifeExp)) + geom_point(aes(color = continent)) +
  scale_x_log10() + xlab("GDP per capita (log scale)") + ylab("Life expectancy") +
  labs("Life expectancy by country, 2007")
```

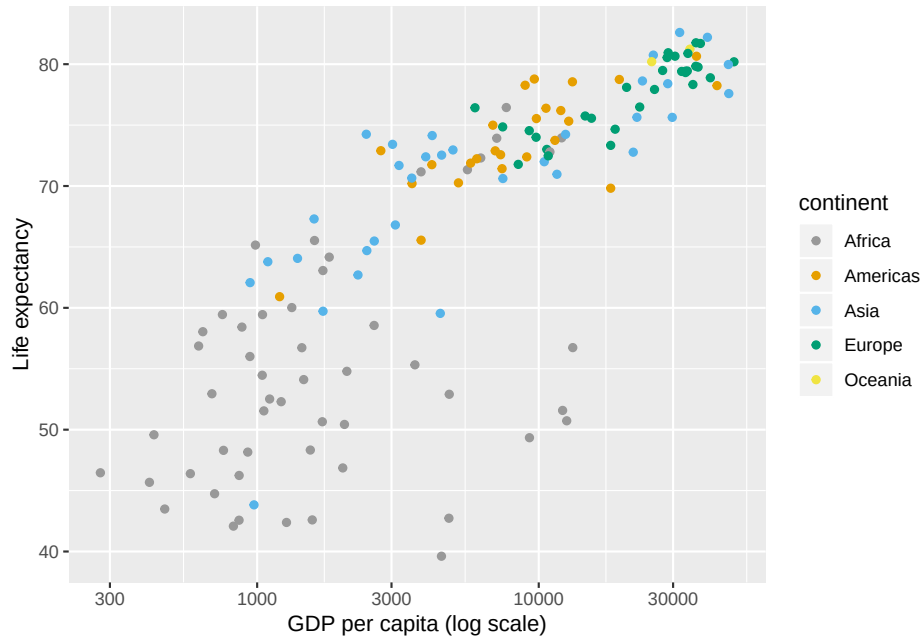


Unfortunately ggplot's default color choices are hard to tell apart, especially for color-blind people (but not just for them.) We should get in the habit of using a color-blind friendly palette:

```
cb_palette = c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC7
```

Now replot the graph with this palette:

```
gg = ggplot(gapminder07, aes(x = gdpPercap, y = lifeExp)) + geom_point(aes(color = continent)) +
  scale_x_log10() + xlab("GDP per capita (log scale)") + ylab("Life expectancy") +
  labs("Life expectancy by country, 2007") + scale_color_manual(values = cb_palette)
gg
```



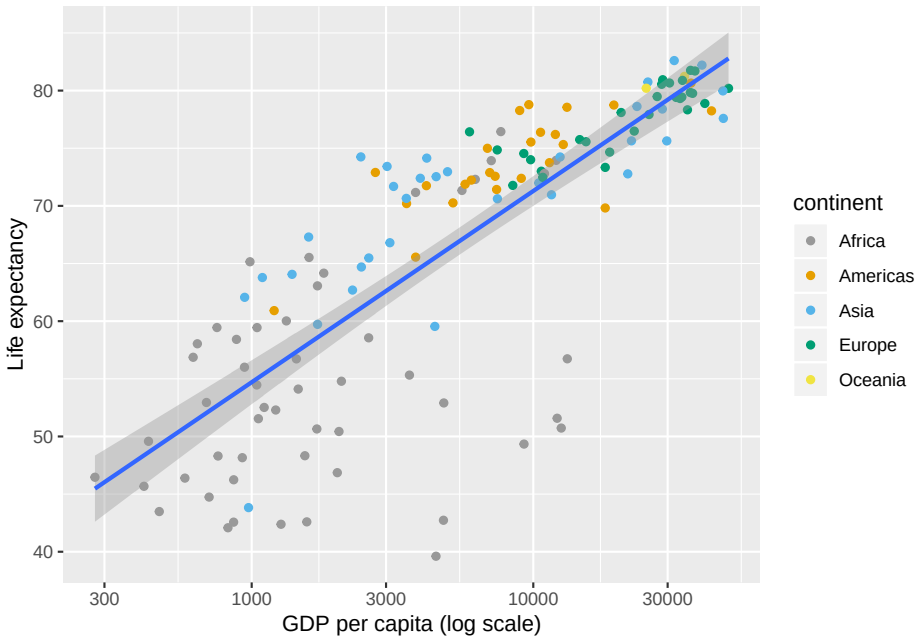
We can now see a few things clearly:

- African countries tend to have low GDP and low life expectancy,
- European countries tend to have high GDP and high life expectancy,
- Countries in the Americas and (especially) Asia are more spread out.

3.4.2 Lots of lines

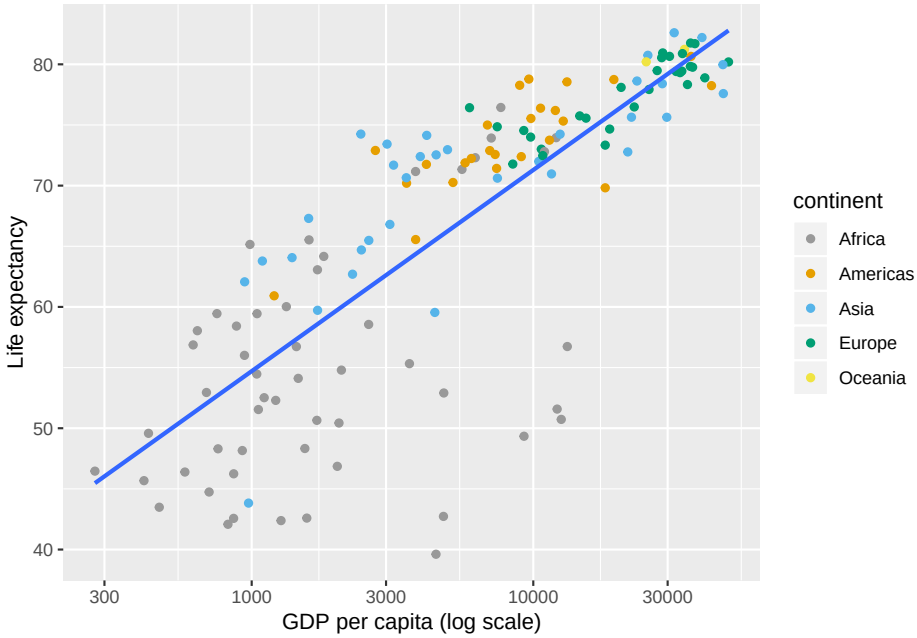
We can add a regression line to our black-and white graph:

```
gg + geom_smooth(method = "lm")
```



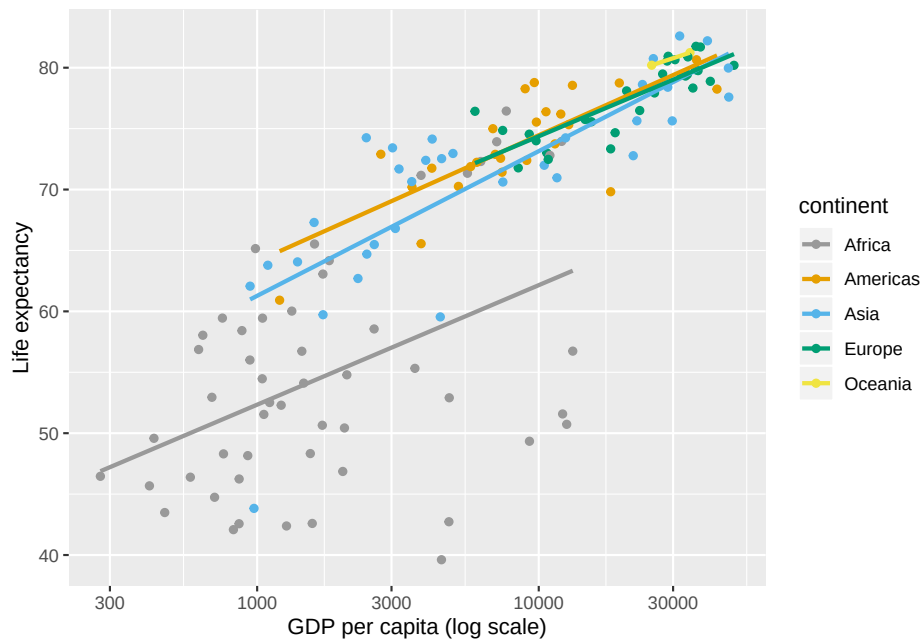
Unless you're willing to make strong assumptions, the shaded confidence band is mostly a distraction, so turn it off:

```
gg + geom_smooth(method = "lm", se = FALSE)
```



If we wanted to fit a model, would this regression line fit all continents well? It's hard to tell from the above graph. We can instead fit separate regression lines for each continent, and compare them:

```
gg + geom_smooth(method = "lm", se = FALSE, aes(color = continent))
```



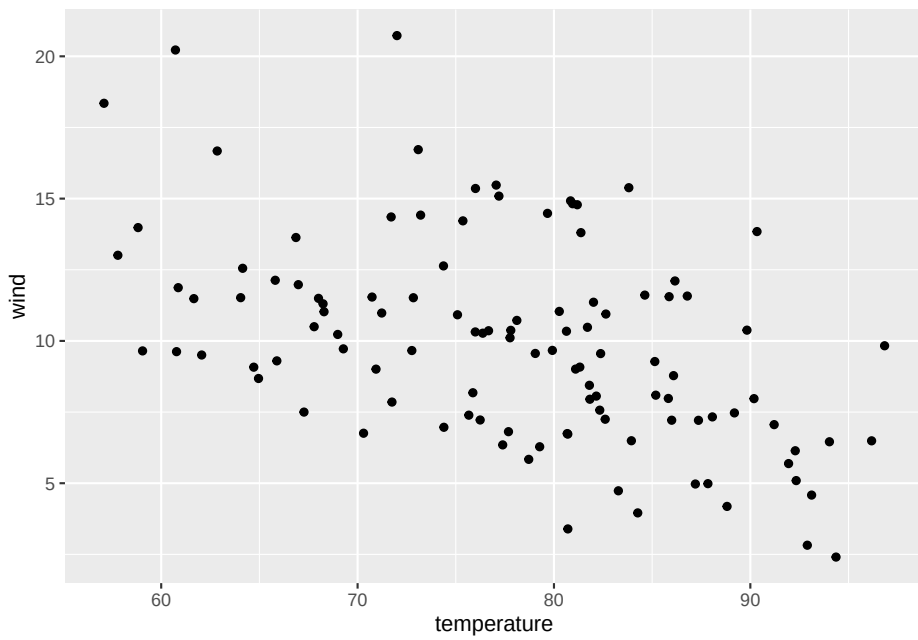
Note that `color` automatically does grouping here. The lines are at very different levels (although the slopes are similar.) This suggests either fitting separate models for each continent, or fitting a more complicated model with both GDP and continent as explanatory variables.

3.5 Truly bivariate data

3.5.1 Wind and temperature

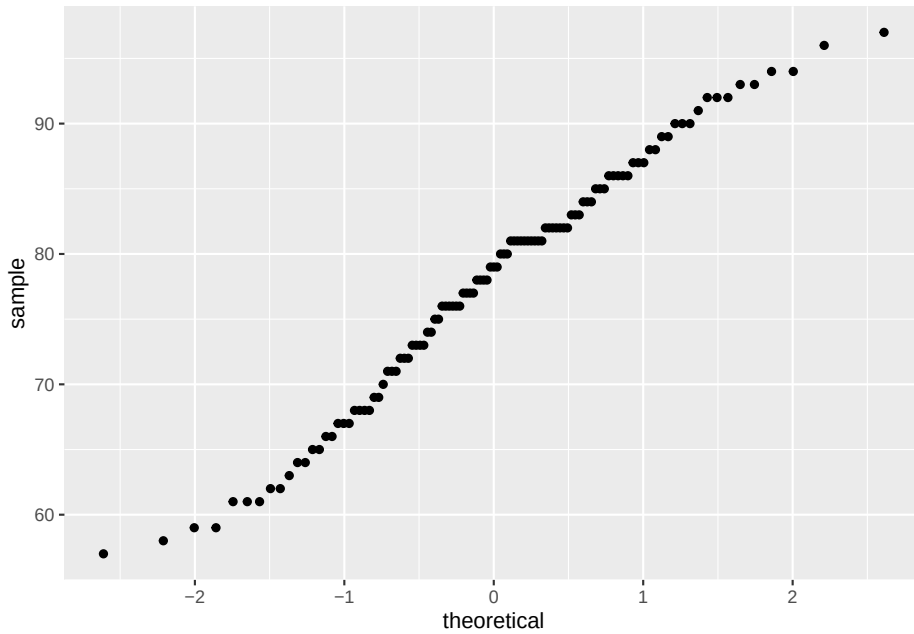
The data set `environmental` contains `temperature` and `wind` measurements on 111 days from May to September 1973. This data is "truly" bivariate, in that it's not appropriate to declare one variable as explanatory and one as the response: both variables depend on each other in some sense. We still start by drawing a (jittered) scatterplot, arbitrarily taking `temperature` as our x-variable:

```
ggplot(environmental, aes(x=temperature, y=wind)) + geom_jitter()
```



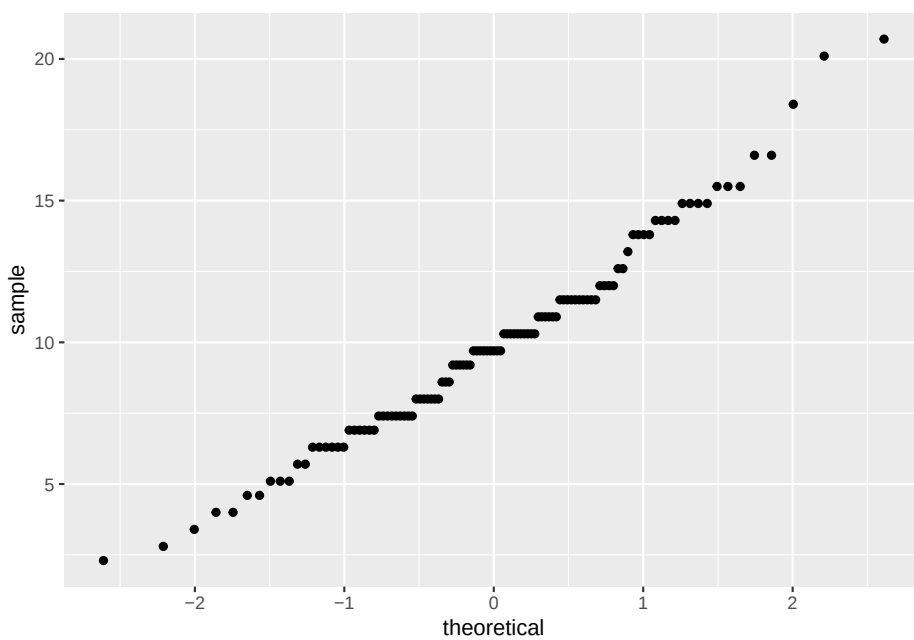
This doesn't too far from the ellipse shape that we see with bivariate normal data. To examine normality more carefully, draw QQ plots. First, temperature:

```
ggplot(environmental, aes(sample=temperature)) + stat_qq()
```



Next, wind:

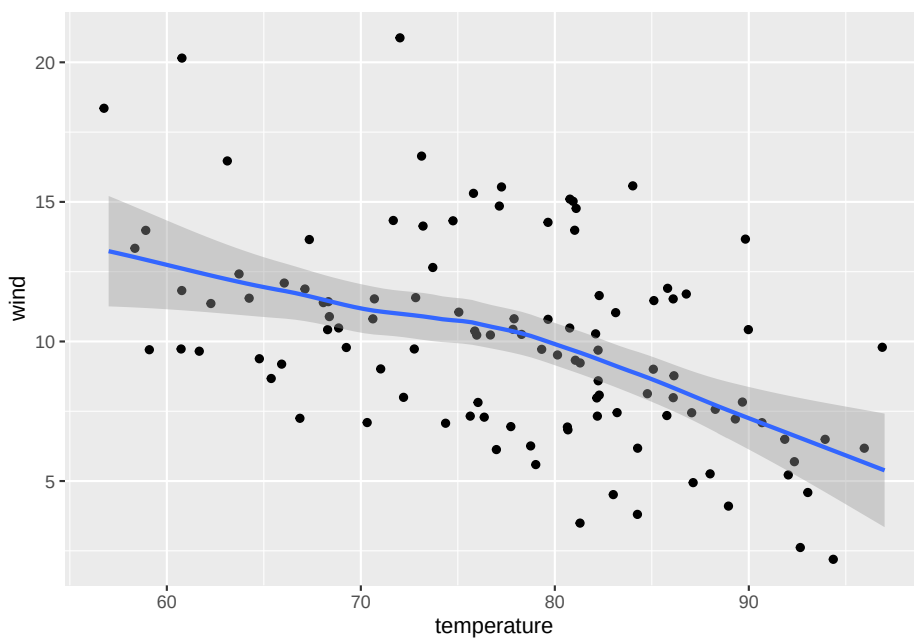
```
ggplot(environmental, aes(sample=wind)) + stat_qq()
```



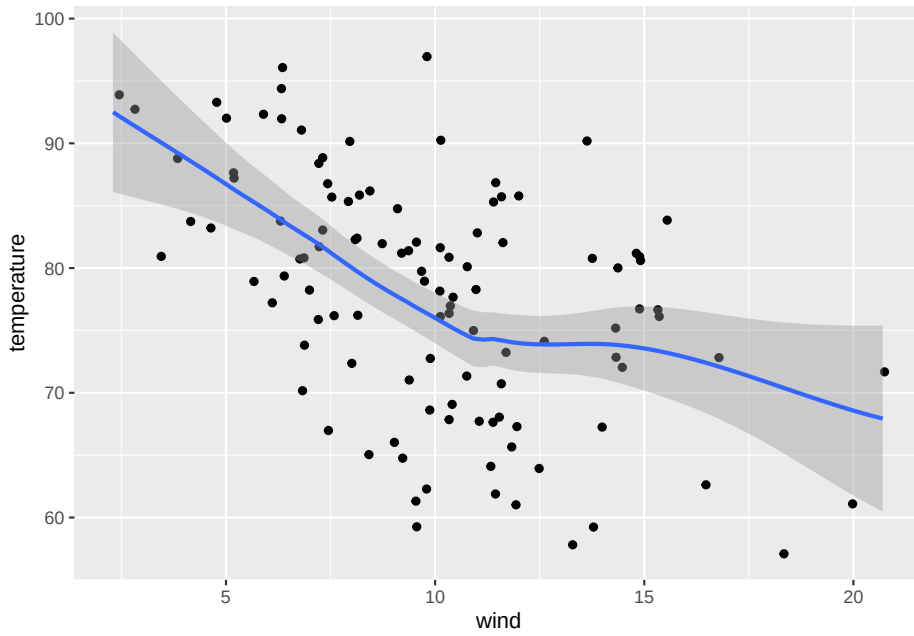
There's a little bit of non-normality in the tails of both distributions, especially temperature. Too bad.

We can model the trend both ways: wind as a function of temperature, and temperature as a wind a function of wind. We could plot both fits on one graph but really that's too much effort, so we'll just do the two separately.

```
gg = ggplot(environmental, aes(x=temperature, y=wind)) + geom_jitter()
gg + geom_smooth(method.args=list(degree=1))
```




```
gg = ggplot(environmental, aes(x=wind, y=temperature)) + geom_jitter()
gg + geom_smooth(method.args=list(degree=1))
```

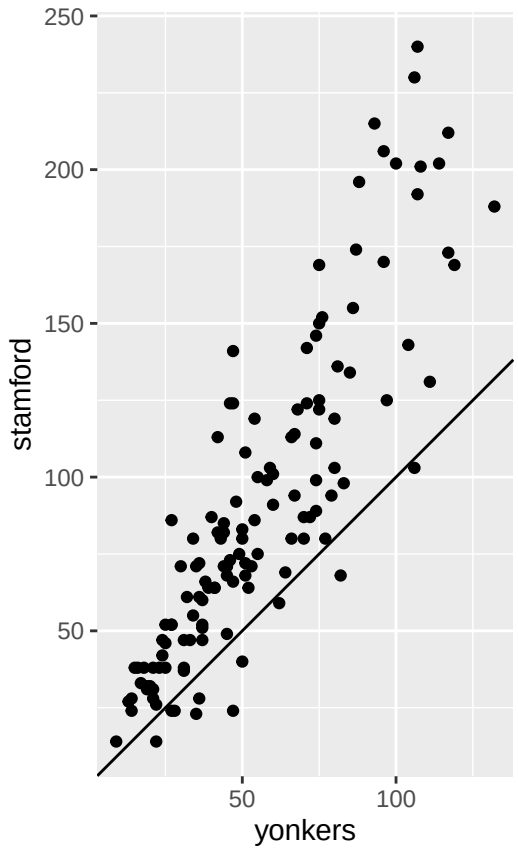


The average wind decreases just about linearly in temperature, while the average temperature has a curved decreasing relationship with wind. This isn't a paradox: when we leave the comfortable world of bivariate normal data, relationships may not be nice and symmetric when you interchange the variables.

3.5.2 Equal scales: Ozone

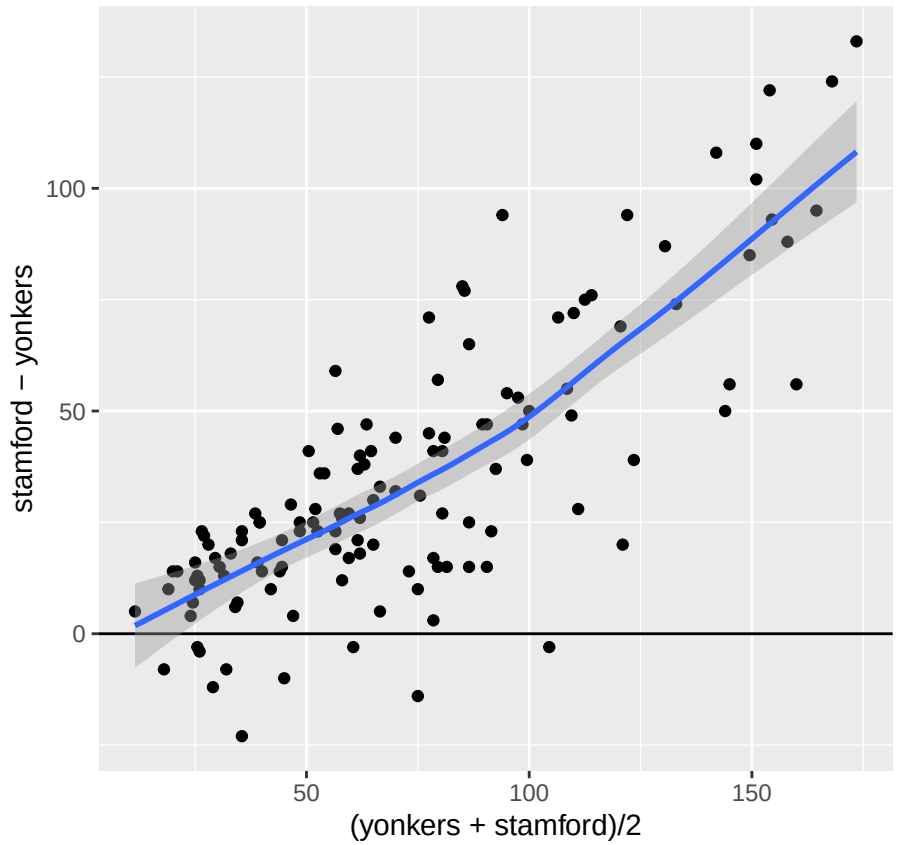
The `ozone` data set contains paired measurements of ozone concentration (in parts per billion) at Yonkers, NY and Stamford, CT. Since we've measuring the same variable at the two locations, it makes sense to use the same scales for both axes. The `coord_fixed()` function takes care of this.

```
ggplot(ozone, aes(x=yonkers, y=stamford)) + geom_point() + geom_abline() + coord_fixed
```



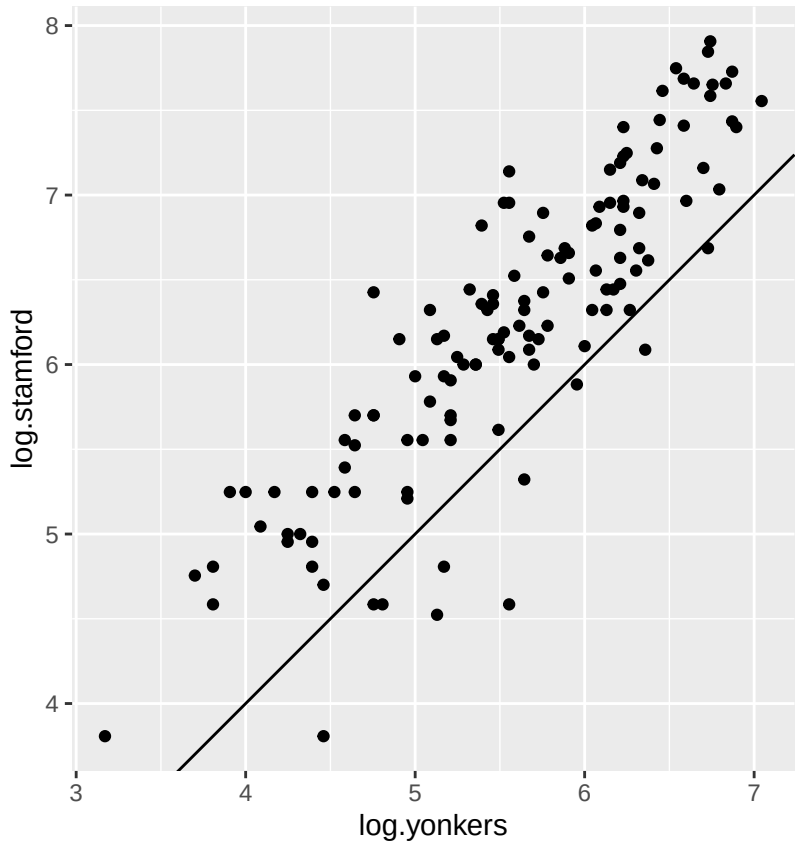
The points are almost all above the line $y = x$, so ozone is typically higher at Stamford. The data looks like it could be well-modeled by a straight line. To investigate further, we could draw a Tukey mean-difference plot.

```
ggplot(ozone, aes(x=(yonkers+stamford)/2, y=stamford-yonkers)) + geom_point() +
  geom_abline(slope=0) + geom_smooth(method.args=list(degree=1)) + coord_fixed()
```



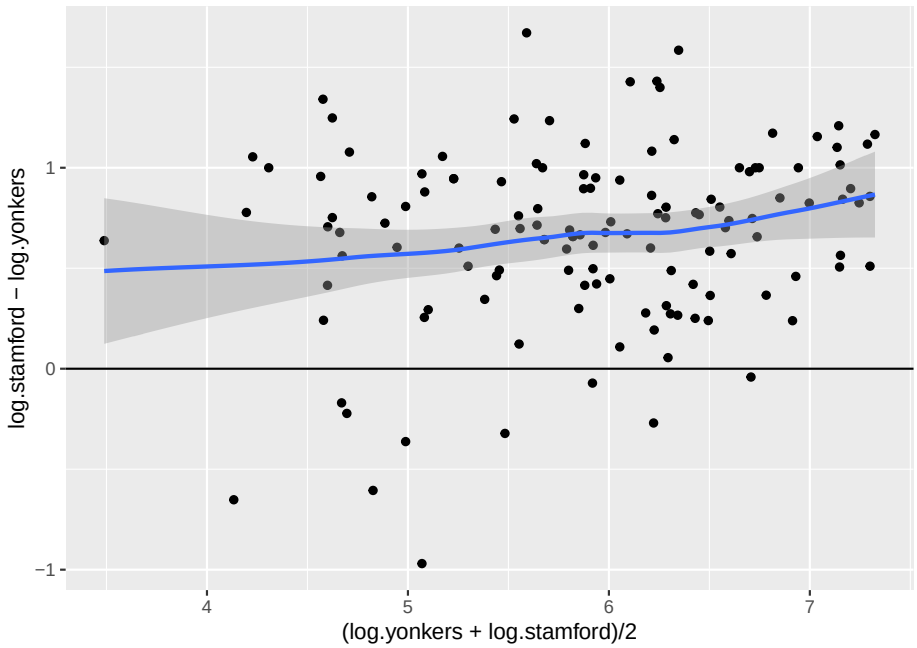
The loess fit is roughly consistent with an upward-sloping straight line. Since the line isn't horizontal, an additive shift isn't appropriate; perhaps a multiplicative shift would be better. Redraw the scatterplot with both variables logged:

```
log.ozone = data.frame(log.yonkers = log2(ozone$yonkers), log.stamford = log2(ozone$sta
ggplot(log.ozone, aes(x=log.yonkers, y=log.stamford)) + geom_point() + geom_abline() +
```



It looks like a line parallel to $y = x$ might be a good fit. To get a better idea, draw the mean-difference plot.

```
gg = ggplot(log.ozone, aes(x=(log.yonkers+log.stamford)/2, y=log.stamford-log.yonkers)) +
  geom_point() + geom_abline(slope=0)
gg + geom_smooth(method.args = list(degree=1)) + coord_fixed()
```



We see that a multiplicative shift is nearly right, but we can do a bit better. There's a little bit of an upward slope in the fitted curve, going from about 0.5 on the left to 0.9 on the right. Back-transformed, that means on low-ozone days in Yonkers, the Stamford ozone concentration is typically about 40% higher; while on high-ozone days in Yonkers, the Stamford ozone concentration is typically about 90% higher.

Chapter 4

Time series data

READ: Cleveland pp. 152–171.

CAUTION. Time series data is very common and a bit dangerous. Observations are not independent, so all the intuition we've built up about how to interpret independent data can be misleading with time series data: it's easy to see patterns that are just random noise. So be wary about drwaing conclusions about things like apparent cycles unless you have solid subject matter knowledge.

RELATED CAUTION. The section of the notes is not as polished as the rest.

Load the usual:

```
library(tidyverse)
cb_palette = c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC7
```

4.1 Time series objects

Time series can be stored in R either as data frames or as `ts` objects. To create a `ts` object:

```
# Real GDP 2012-2017 in 2012 dollars
gdp.ts = ts(data = c(16.2, 16.6, 17.1, 17.5, 17.8, 18.2), start=2012)
gdp.ts
```

```
## Time Series:
## Start = 2012
## End = 2017
## Frequency = 1
## [1] 16.2 16.6 17.1 17.5 17.8 18.2
```

To turn a `ts` object into a data frame:

```
gdp.df = data.frame(x=as.matrix(gdp.ts), year=time(gdp.ts))
gdp.df
```

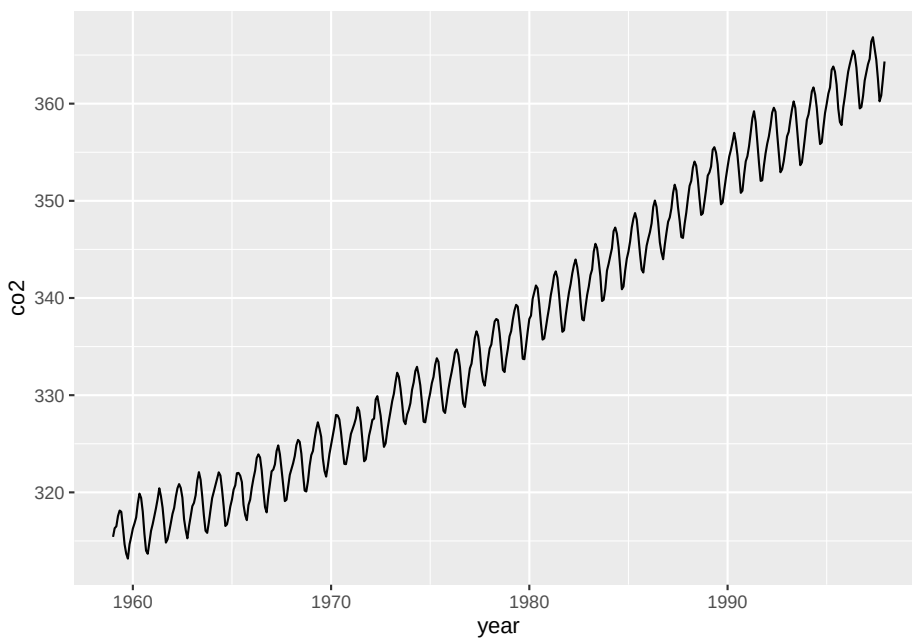
```
##      x year
## 1 16.2 2012
## 2 16.6 2013
## 3 17.1 2014
## 4 17.5 2015
## 5 17.8 2016
## 6 18.2 2017
```

The `ts` format is good for doing traditional time series analysis, such as fitting ARIMAs. The data frame format is necessary to use `ggplot()`.

4.2 Carbon dioxide: Seasonal time series

The ts object `co2` contains the monthly average carbon dioxide concentration (in ppm) at Mauna Loa, Hawaii from 1959 to 1997. Draw a line graph:

```
co2.df = data.frame(year=time(co2), co2)
ggplot(co2.df, aes(x=year, y=co2)) + geom_line()
```



Clearly there's a seasonal component in addition to the trend. We wish to decompose the time series into trend, seasonal, and residual components. The

`stl()` function performs such a seasonal decomposition on a ts object via loess. Two key arguments are:

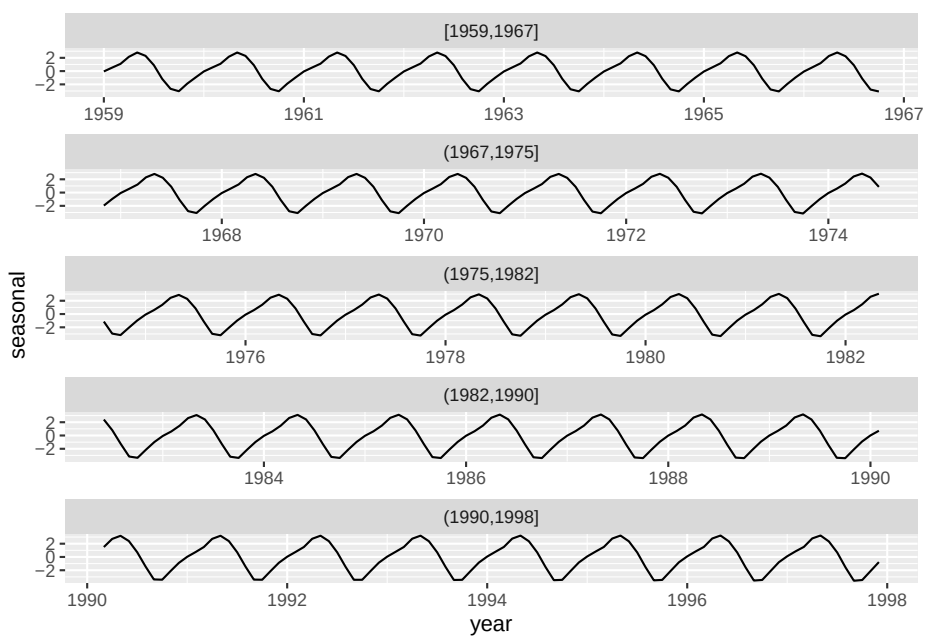
- `s.window`: either "periodic" or an (odd) number giving the size of the loess window for the seasonal component. "Periodic" means the values repeat exactly in a cycle, while a number means the size of the seasonal components will vary somewhat.
- `s.degree`: the degree of local polynomial for the seasonal component. 0 (constant) by default, 1 for linear.

```
co2.stl = stl(co2, s.window=25, s.degree=1)
head(co2.stl$time.series)
```

```
##        seasonal    trend   remainder
## [1,] -0.0507468 315.3117  0.15903765
## [2,]  0.5295002 315.4005  0.37998652
## [3,]  1.0902950 315.4893 -0.07961229
## [4,]  2.1473187 315.5781 -0.16544004
## [5,]  2.8016286 315.6670 -0.33867857
## [6,]  2.2995644 315.7560 -0.05554291
```

The plot of the seasonal component is highly repetitive. To get a better look, we cut the series into several pieces, then make a stack of time series graphs.

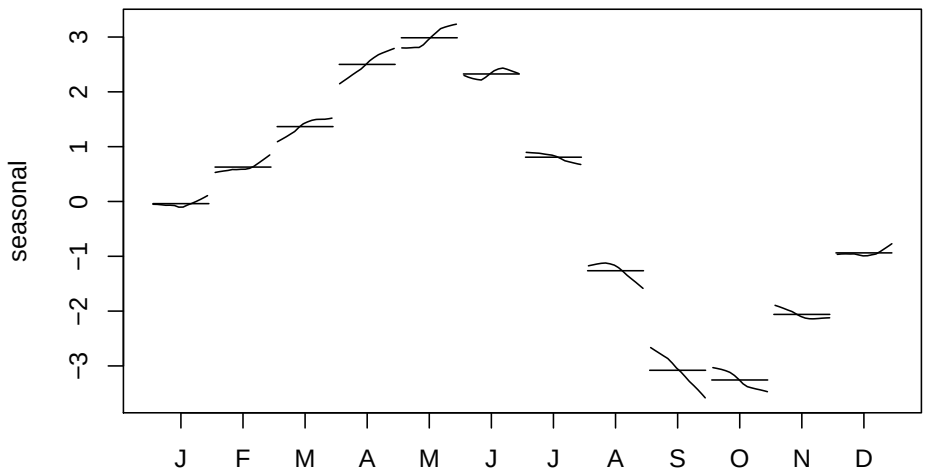
```
year.cut = cut_number(time(co2), n=5)
co2.df2 = data.frame(year=time(co2), co2.stl$time.series, year.cut)
ggplot(co2.df2, aes(x=year, y=seasonal)) + geom_line() + facet_wrap(~year.cut, ncol=1, scales="fr
```



Looking carefully at the scales, it looks like the seasonal oscillations get a bit bigger with time, but it's a pretty small change.

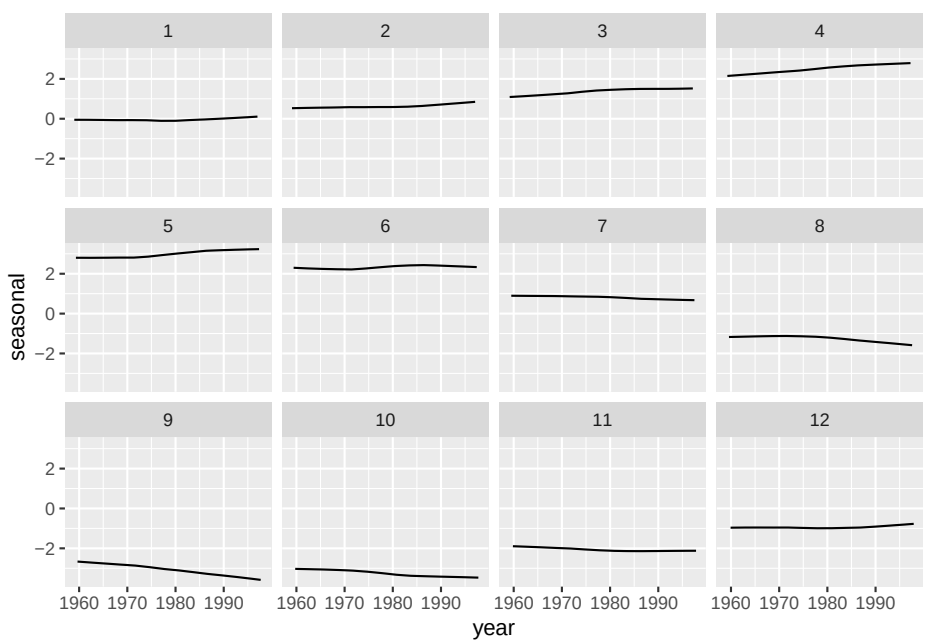
To better understand the seasonal component, we want to see how the value for each month varies by year. The `monthplot()` function in base R does this:

```
monthplot(co2.stl, choice="seasonal")
```



We can do something similar with `ggplot`:

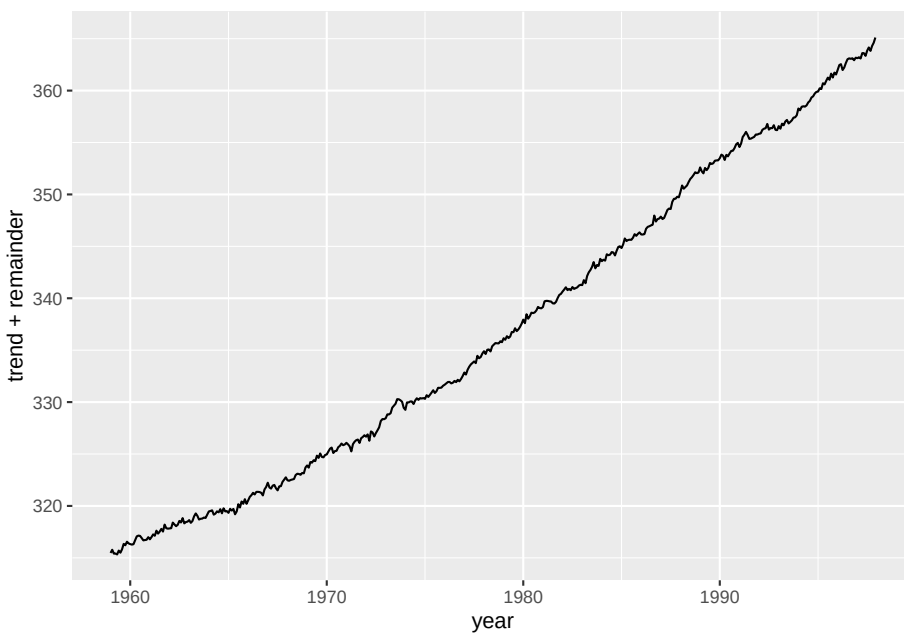
```
co2.month = data.frame(co2.df2, month=factor(1:12))
ggplot(co2.month, aes(x=year, y=seasonal)) + geom_line() + facet_wrap(~month)
```



Either way, the seasonal pattern is clear: CO2 is highest in May and lowest in September and October. The fact that the May line points up and the September/October lines point down again shows the size of the seasonal variation is increasing.

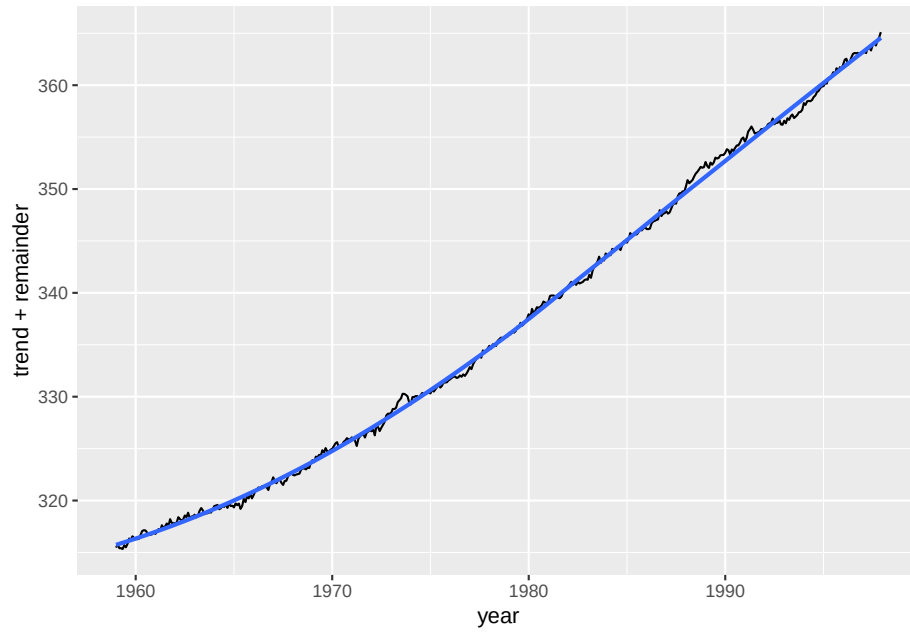
We now plot what's left over after removing the seasonal component. That is, add together the "trend" and "remainder" and plot the time series.

```
ggplot(co2.month, aes(x=year, y=trend+remainder)) + geom_line()
```



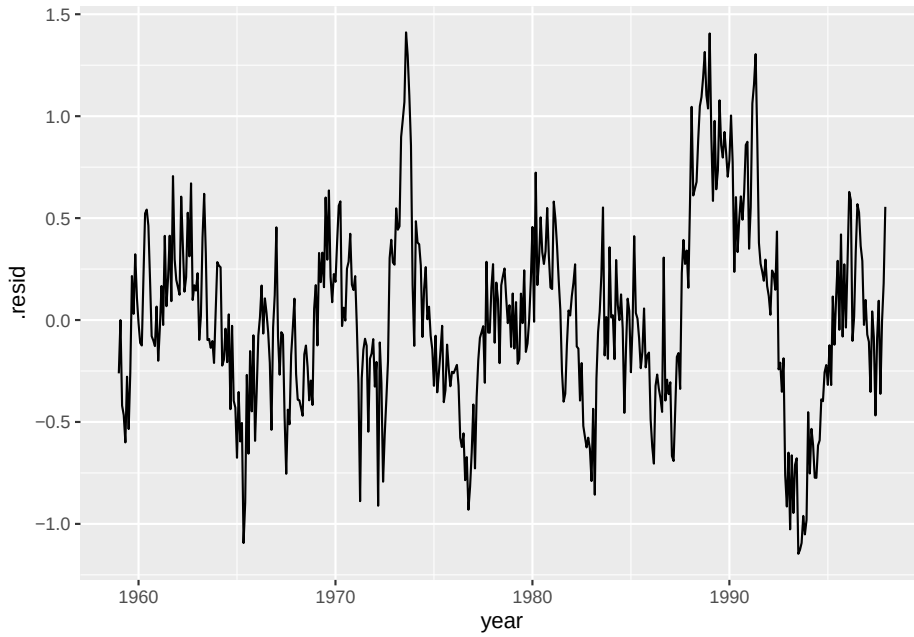
Clearly there's a big trend component and comparatively small residuals. Add a loess fit:

```
ggplot(co2.month, aes(x=year, y=trend+remainder)) + geom_line() + geom_smooth()
```



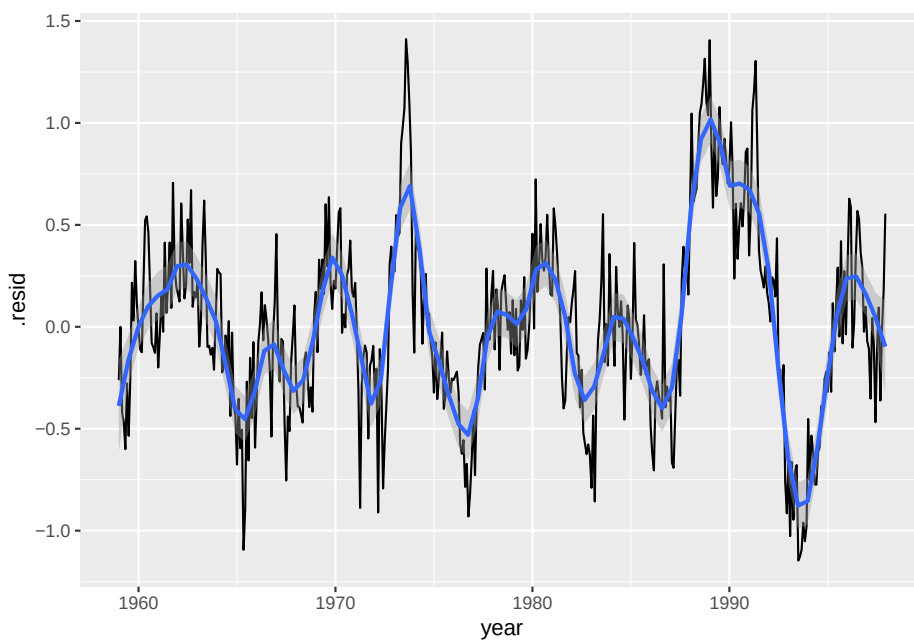
The smooth fit curves slightly upward, suggesting that the increase in CO2 levels is accelerating. Next, we either throw up our hands and say "we're all doomed" or we look at the residuals. Pursuing the latter approach, we re-fit the loess and use `augment` to incorporate the residuals into a data frame.

```
library(broom)
co2.lo = loess(trend+remainder~year, data=co2.month)
co2.lo.df = augment(co2.lo)
ggplot(co2.lo.df, aes(x=year, y=.resid)) + geom_line()
```



It still looks like the residuals oscillate up and down a bit. We once again fit a loess model with a smaller span. After some trial and error, a span of 0.1 looks about right.

```
ggplot(co2.lo.df, aes(x=year, y=.resid)) + geom_line() + geom_smooth(span=0.1)
```



There are around eight or nine oscillatory "cycles" over 39 years. However, the size and length of the cycles varies quite a lot, and irregular cycles are difficult to distinguish from random noise. So unless subject matter knowledge gives you a good reason to continue (in which case you might look at third-order residuals), I would just stop here.

4.3 Distributions in time

For decades, the Census Bureau has collected data on the education level of U.S. adults. I found the "Years of School Completed by People 25 Years and Over, by Age and Sex: Selected Years 1940 to 2016" here:

https://www2.census.gov/programs-surveys/demo/tables/educational-attainment/2016/cps-detailed-tables/histtab-A-01.xlsx

The Census Bureau is notorious for releasing its data in weirdly organized Excel spreadsheets. Load the spreadsheet:

```
library(rio)
educ = import("histtab-A-01.xlsx")
head(educ)
```

```
## Table with row headers in column A and column headers in rows
5 through 7.
## 1 Table A-1. Years of School Completed by People 25 Years and
Over, by Age and Sex: Selected Years 1940 to 2016
## 2 (Numbers in thousands. Noninstitutionalized population
except where otherwise specified.)
## 3 <NA>
## 4 Age, sex, and years
## 5 <NA>
## 6 <NA>
## ...2 ...3 ...4 ...5 ...6 ...7 ...8
## 1 <NA> NA <NA> <NA> <NA> <NA> <NA>
## 2 <NA> NA <NA> <NA> <NA> <NA> <NA>
## 3 <NA> NA <NA> <NA> <NA> <NA> <NA>
## 4 <NA> NA Total Years of School Completed <NA> <NA> <NA>
## 5 <NA> NA <NA> Elementary <NA> High school <NA>
## 6 <NA> NA <NA> 0 to 4 years 5 to 8 years 1 to 3 years 4 years
## ...9 ...10 ...11 ...12 ...13
## 1 <NA> <NA> <NA> NA <NA>
## 2 <NA> <NA> <NA> NA <NA>
## 3 <NA> <NA> <NA> NA <NA>
## 4 <NA> <NA> <NA> NA <NA>
## 5 College <NA> Median NA <NA>
## 6 1 to 3 years 4 years or more <NA> NA <NA>
```

Yeah this is a disaster. The most interesting numbers are in columns C to J and rows 12 to 42. Extract this region of the data:

```
educ.counts = educ[12:42, 3:10]
```

But does R recognize the columns as numeric?

```
is.numeric(educ.counts[,1])
```

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

```
is.numeric(educ.counts[,2])
```

```
## [1] FALSE
```

Usually the `as.numeric()` function is sufficient to make recognize data as numeric. However, in this case, the data read in from Excel has been stored as a *list*. To change the list into a numeric data frame, we use `lapply()` ("list apply"), then coerce the results into a data frame:

```
educ.counts = as.data.frame(lapply(educ.counts, as.numeric))
is.numeric(educ.counts[,2])
```

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

Now add column names:

```
names(educ.counts) = c("Year", "Total", "Elem 0-4", "Elem 5-8", "HS 1-3", "High school",
   "Some college", "College degree")
```

It's hard to keep track of a categorical variable over more than about four categories. To simplify, we combine all the categories for less than a high school degree.

```
educ.counts$"Less than high school" = educ.counts$"Elem 0-4" + educ.counts$"Elem 5-8" +
  educ.counts$"HS 1-3"
head(educ.counts)
```

```
## Year Total Elem 0-4 Elem 5-8 HS 1-3 High school Some college
College degree
## 1 2015 212132 2601 7295 14686 62575 56031 68945
## 2 2014 209287 2525 7388 14545 62240 55709 66879
## 3 2013 206899 2344 7578 14595 61704 55173 65506
## 4 2012 204579 2484 7800 14993 62113 53900 63291
## 5 2011 201543 2589 7688 14763 61911 53249 61343
## 6 2010 199928 2615 7836 15260 62456 51920 59840
## Less than high school
## 1 24582
## 2 24458
## 3 24517
## 4 25277
```

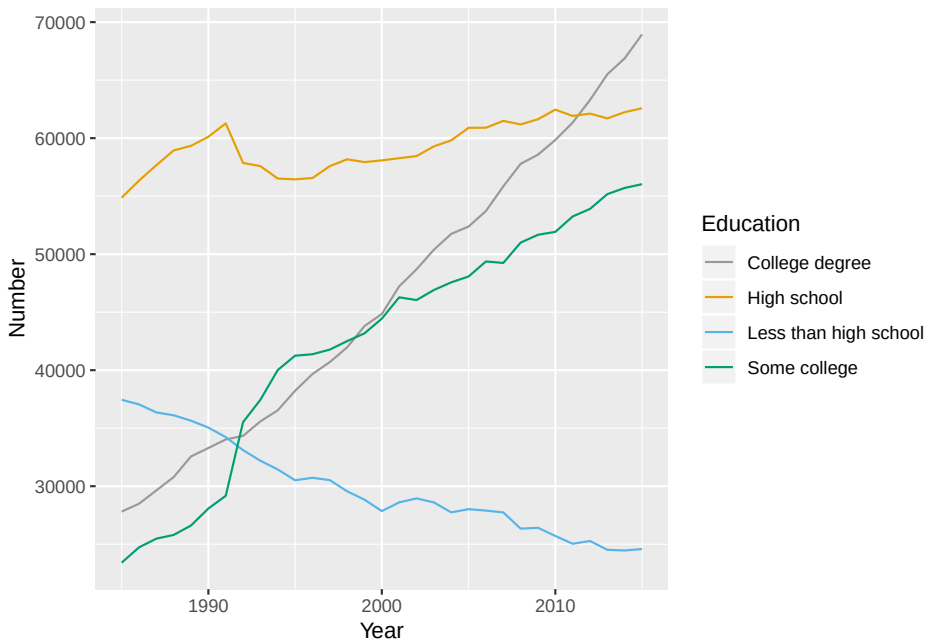
```
## 5 25040
## 6 25711
```

We `gather()` the columns with the counts for less than high school, high school degree, some college, and college degree. Then we draw a line for each category.

```
educ.long  = gather(educ.counts, key = Education, value = Number, 6:9)
head(educ.long)
```

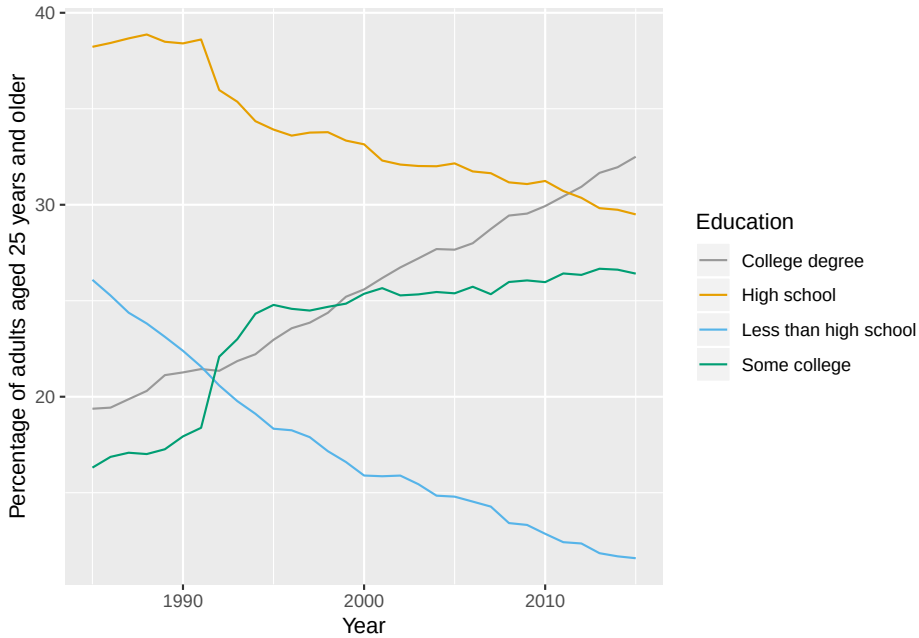
```
##   Year  Total Elem 0-4 Elem 5-8 HS 1-3   Education Number
## 1 2015 212132     2601     7295  14686 High school  62575
## 2 2014 209287     2525     7388  14545 High school  62240
## 3 2013 206899     2344     7578  14595 High school  61704
## 4 2012 204579     2484     7800  14993 High school  62113
## 5 2011 201543     2589     7688  14763 High school  61911
## 6 2010 199928     2615     7836  15260 High school  62456
```

```
ggplot(educ.long, aes(x = Year, y = Number, group = Education, color = Education)) +
  geom_line() + scale_color_manual(values = cb_palette)
```



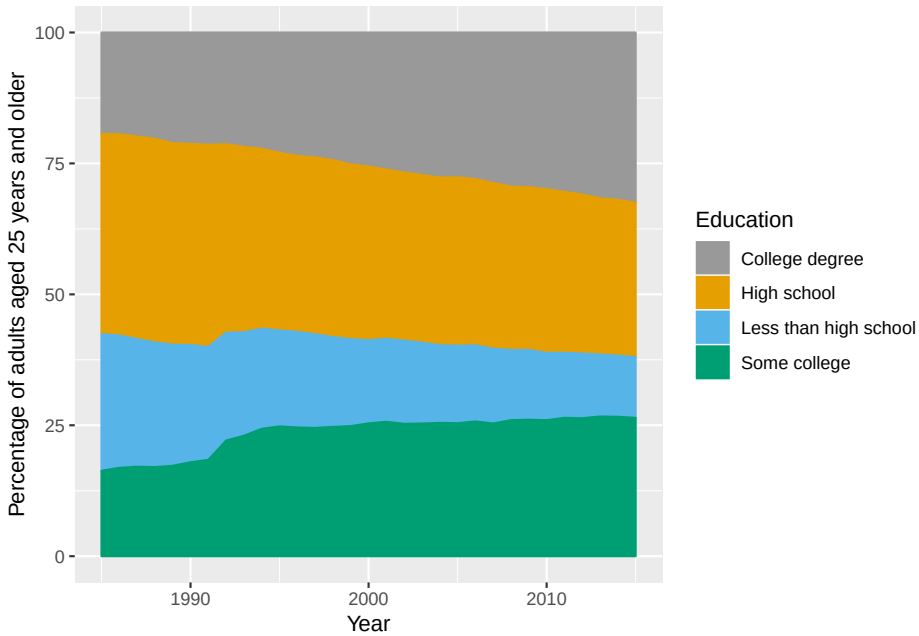
Of course, the total number of adults increased greatly over this time period. So it's usually better to display percentages rather that counts to show changes in a population distribution over time.

```
gg = ggplot(educ.long, aes(x = Year, y = Number/Total*100, group = Education, color = E
  geom_line()
gg + ylab("Percentage of adults aged 25 years and older") + scale_color_manual(values =
```



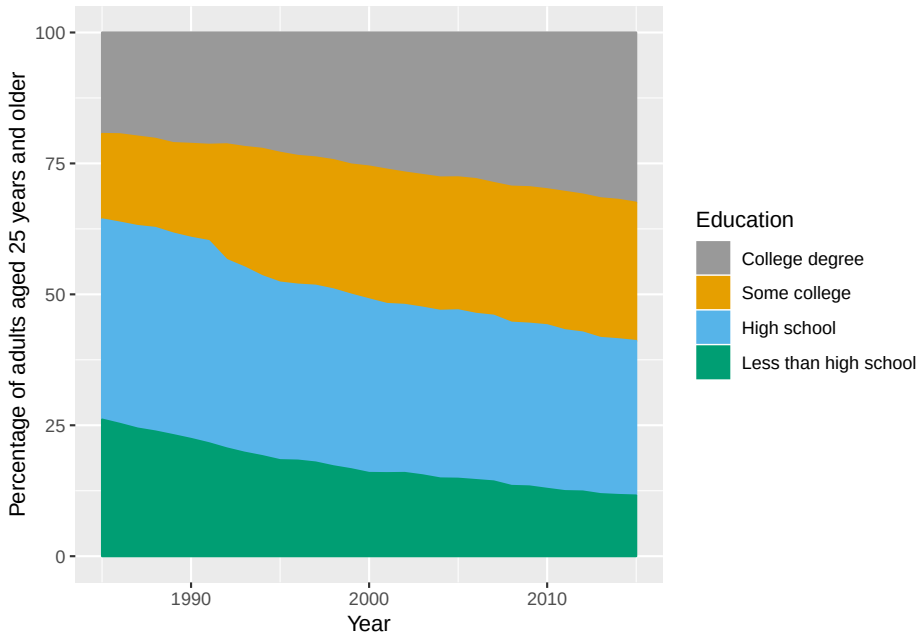
While statisticians tend to like lines for the ease of seeing trends, many datavis people prefer **stacked** plots, especially when the parts sum to a whole.

```
gg = ggplot(educ.long, aes(x = Year, y = Number/Total*100, group = Education, color = Education,
  fill = Education)) + geom_area(position = 'stack')
gg + ylab("Percentage of adults aged 25 years and older") + scale_color_manual(values = cb_palett
  scale_fill_manual(values = cb_palette)
```



The above graph is hard to interpret because the areas are stacked in alphabetical order, which isn't meaningful here. We can reorder the `Education` variable by making it a factor and explicitly giving an ordering for the levels (note that here I use reverse order, from most to least education, as the first level listed appears at the top.)

```
educ.long$Education = factor(educ.long$Education, levels = c("College degree", "Some co
  "High school", "Less than high school"))
gg = ggplot(educ.long, aes(x = Year, y = Number/Total*100, group = Education,
  color = Education, fill = Education)) + geom_area(position = 'stack')
gg + ylab("Percentage of adults aged 25 years and older") + scale_color_manual(values =
  scale_fill_manual(values = cb_palette)
```



Compared to the line graph, it's harder to judge the sizes of the percentages in the middle categories, but easier to judge trends in combinations like "at least some college."

4.4 Comparing time series: Presidential elections

Data: https://docs.google.com/spreadsheets/d/1D-edaVHTnZNhVU840EPUhz3Cgd7m39Urx7HM8Pq6Pus/edit#gid=1024689221

Download the 2-party sheet as a .csv. Read the data in (skip the first two lines):

```
library(readr)
presElections = read_csv("elections-1828-2016.csv", col_names = FALSE, skip = 2)
```

Get Democratic share of 2-party vote:

```
which.columns = c(1, seq(2, 158, 4), 163)
DemShare = presElections[, which.columns]
```

Get rid of rows 53, 54:

```
DemShare = DemShare[-c(53, 54), ]
```

Missing (not ordinary 2-party elections): 1968, 1948, 1928, 1912, 1892, 1860, 1856

```
names(DemShare) = c("State", seq(2016, 1972, -4), seq(1964, 1952, -4), seq(1944, 1932,
  seq(1924, 1916, -4), seq(1908, 1896, -4), seq(1888, 1864, -4), seq(1852, 1828, -4))
```

Gather the data into "long form":

```
DemLong = gather(DemShare, key = Year, value = DemVote, "2016":"1828")
```

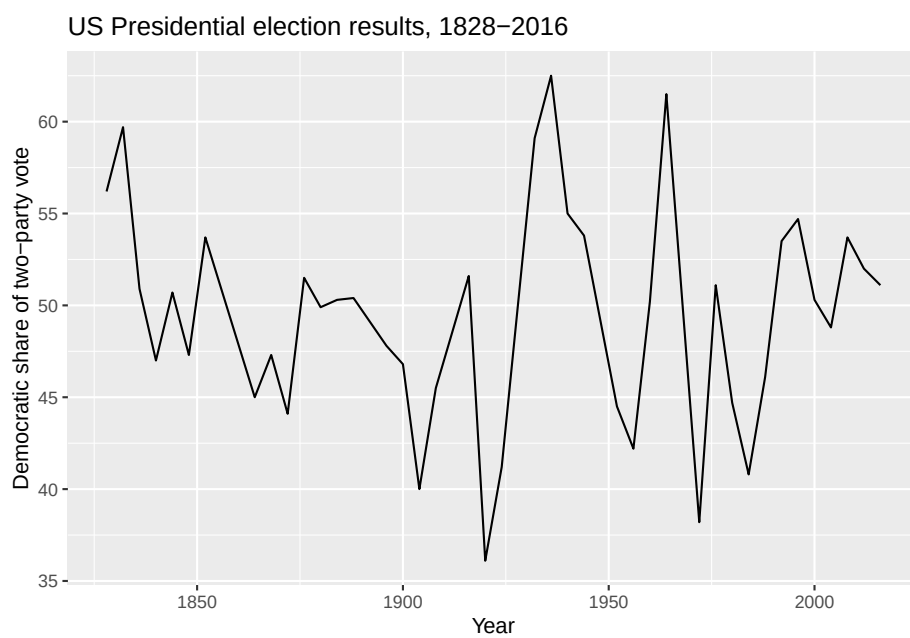
Make numeric:

```
DemLong$Year = as.numeric(DemLong$Year)
DemLong$DemVote = as.numeric(DemLong$DemVote)
```

```
## Warning: NAs introduced by coercion
```

Plot the nationwide vote:

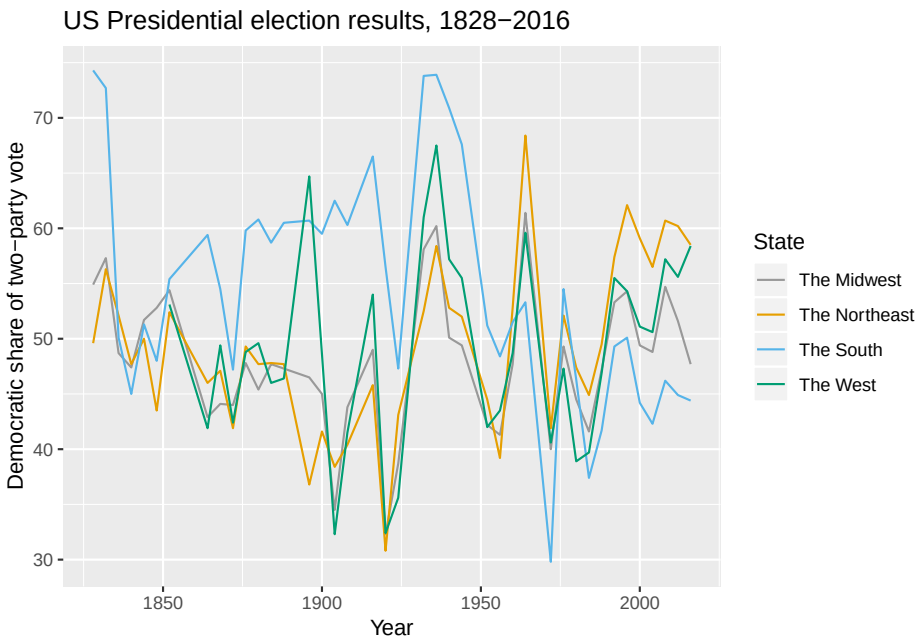
```
gg = ggplot(subset(DemLong, State == "Nationwide"), aes(x = Year, y = DemVote)) +
  geom_line() + ylab("Democratic share of two-party vote")
gg + labs(title = "US Presidential election results, 1828-2016")
```



Plot regional results:

```
RegionalVote = subset(DemLong, State %in% c("The Midwest", "The Northeast", "The South"
gg = ggplot(RegionalVote, aes(x = Year, y = DemVote, color = State)) + geom_line() +
  ylab("Democratic share of two-party vote")
gg = gg + labs(title = "US Presidential election results, 1828-2016")
gg + scale_color_manual(values = cb_palette)
```

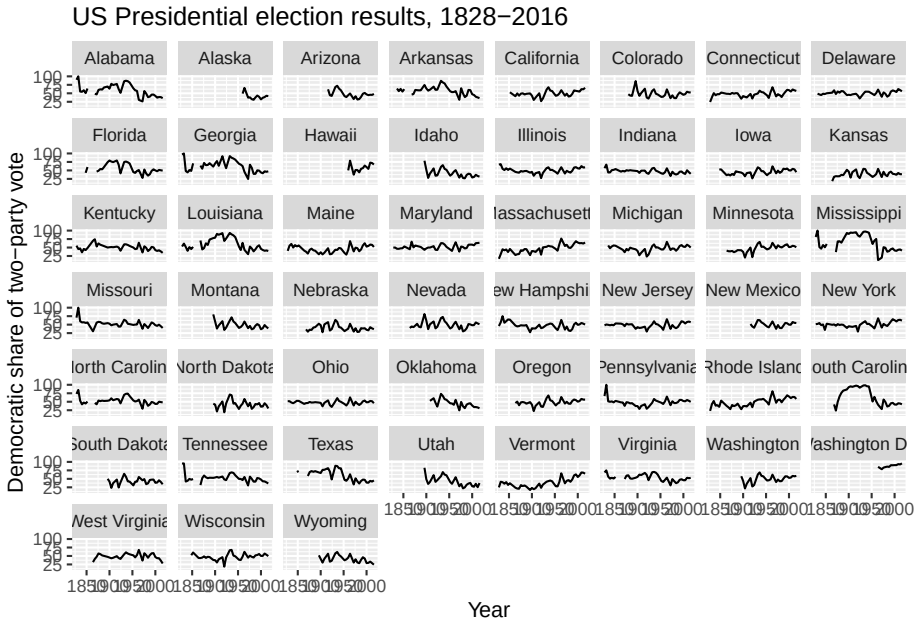
```
## Warning: Removed 6 rows containing missing values (geom_path).
```



(Would be better to subtract off the nationwide vote.)

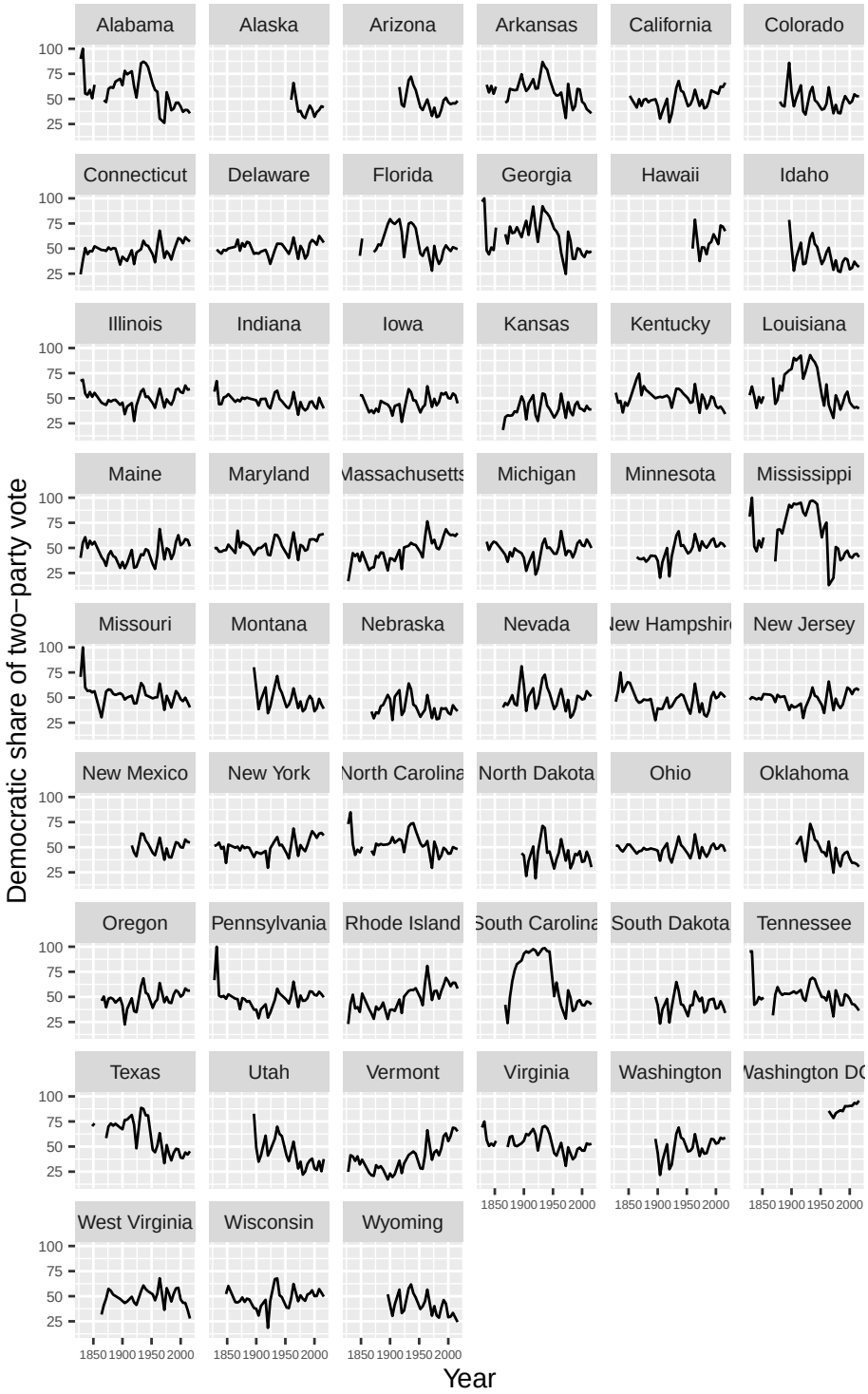
By state:

```
StateVote = subset(DemLong, !(State %in% c("Nationwide", "The Midwest", "The Northeast",
  "The South", "The West")))
gg = ggplot(StateVote, aes(x = Year, y = DemVote)) + geom_line()
gg = gg + ylab("Democratic share of two-party vote")
gg + labs(title = "US Presidential election results, 1828-2016") + facet_wrap(~ State)
```



Make it nicer:

```
gg + facet_wrap(~ State, ncol = 6)+ theme(axis.text.x = element_text(size = 5),
  axis.text.y = element_text(size = 6), strip.text.x = element_text(size = 8))
```


Alabama
Alaska
Arizona
Arkansas
California
Colorado
Connecticut
Delaware
Florida
Georgia
Hawaii
Idaho
Illinois
Indiana
Iowa
Kansas
Kentucky
Louisiana
Maine
Maryland
Massachusetts
Michigan
Minnesota
Mississippi
Missouri
Montana
Nebraska
Nevada
New Hampshire
New Jersey
New Mexico
New York
North Carolina
North Dakota
Ohio
Oklahoma
Oregon
Pennsylvania
Rhode Island
South Carolina
South Dakota
Tennessee
Texas
Utah
Vermont
Virginia
Washington
Washington DC
West Virginia
Wisconsin
Wyoming
100
75
50
25
1850 1900 1950 2000
Democratic share of two−party vote
Year

Chapter 5

Trivariate data

Load the usual:

```
library(tidyverse)
library(lattice)
cb_palette = c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC7
```

5.1 Trivariate data: Ethanol

READ: Cleveland pp. 188–190, 196–199, 214–217, 254–255.

The data frame `ethanol` in the `lattice` package contains three measurements from 88 runs of an experiment testing an ethanol-fueled engine:

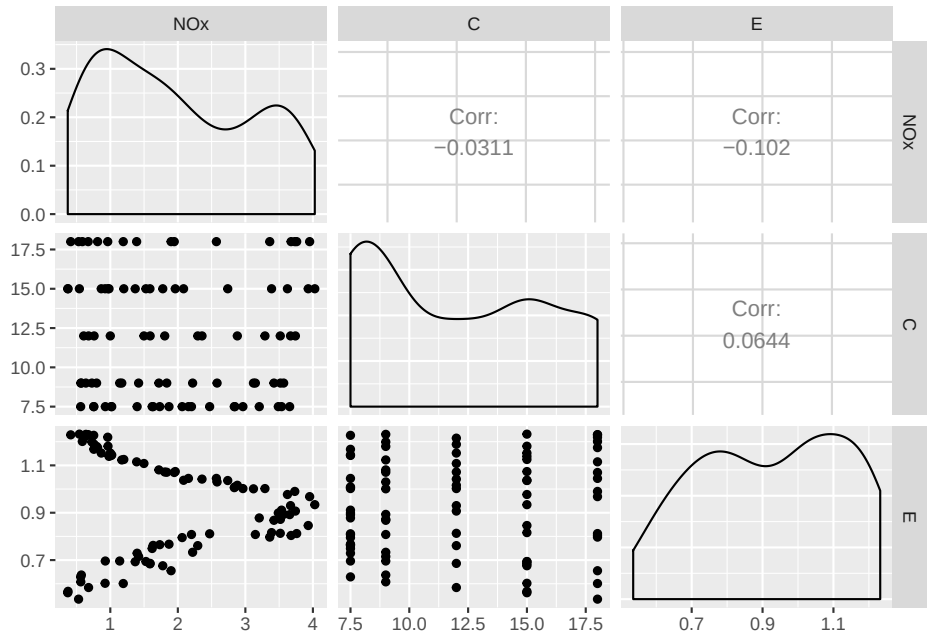
- `NOx`: the amount of oxides of nitrogen produced by the engine per unit of work, in micrograms per joule (the response.)
- `C`: the compression ratio the engine was set at.
- `E`: the equivalence ratio the engine was set at (a measure of how fuel is in the fuel-air mixture.)

We first calculate summary statistics, then plot the pairwise relationships using `ggpairs()` in the `GGally` package.

```
summary(ethanol)
```

```
##       NOx              C                E
##  Min.   :0.370   Min.   : 7.500   Min.   :0.5350
##  1st Qu.:0.953   1st Qu.: 8.625   1st Qu.:0.7618
##  Median :1.754   Median :12.000   Median :0.9320
##  Mean   :1.957   Mean   :12.034   Mean   :0.9265
##  3rd Qu.:3.003   3rd Qu.:15.000   3rd Qu.:1.1098
##  Max.   :4.028   Max.   :18.000   Max.   :1.2320
```

```
library(GGally)
ggpairs(ethanol)
```

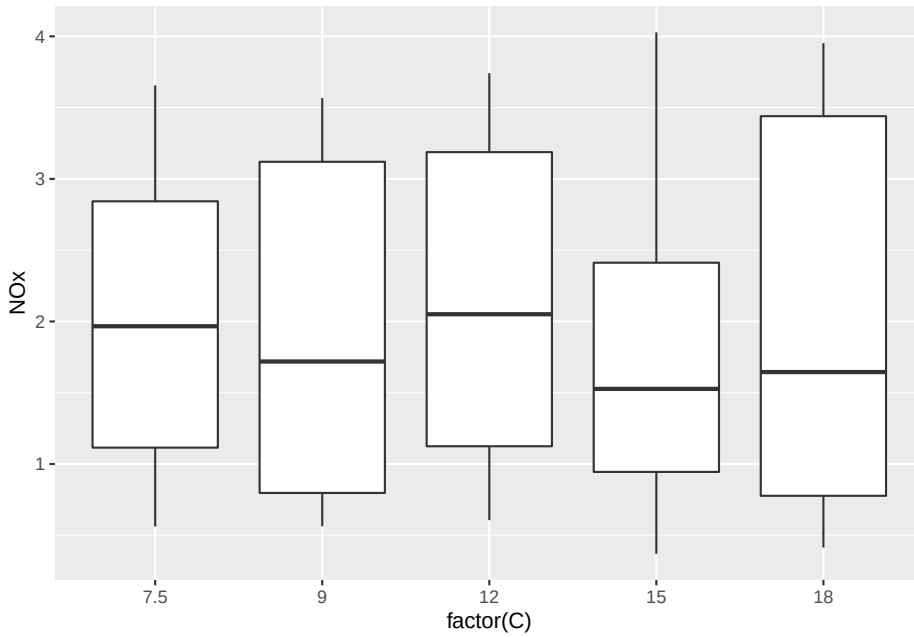


Some features are immediately obvious:

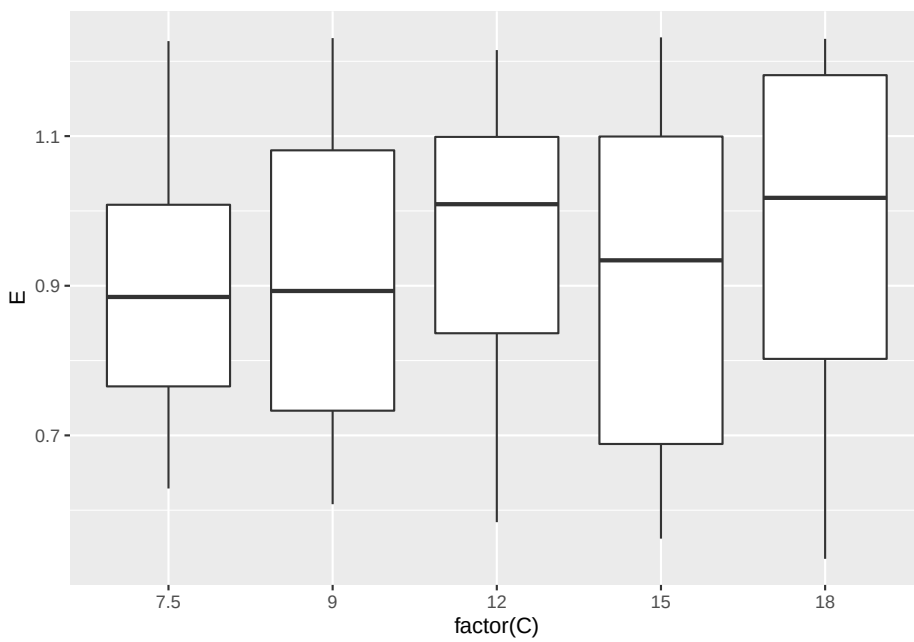
- Compression takes only five different values (it was directly controlled in the experiments.)
- The relationship of NOx as a function of equivalence ratio is non-linear and non-monotonic. (It actually looks kind of like a bell-shaped curve.)

On the other hand, the relationships of equivalence ratio and NOx with compression are not obvious. With five levels of C, we could try boxplots:

```
ggplot(ethanol, aes(x=factor(C), y=NOx)) + geom_boxplot()
```




```
ggplot(ethanol, aes(x=factor(C), y=E)) + geom_boxplot()
```



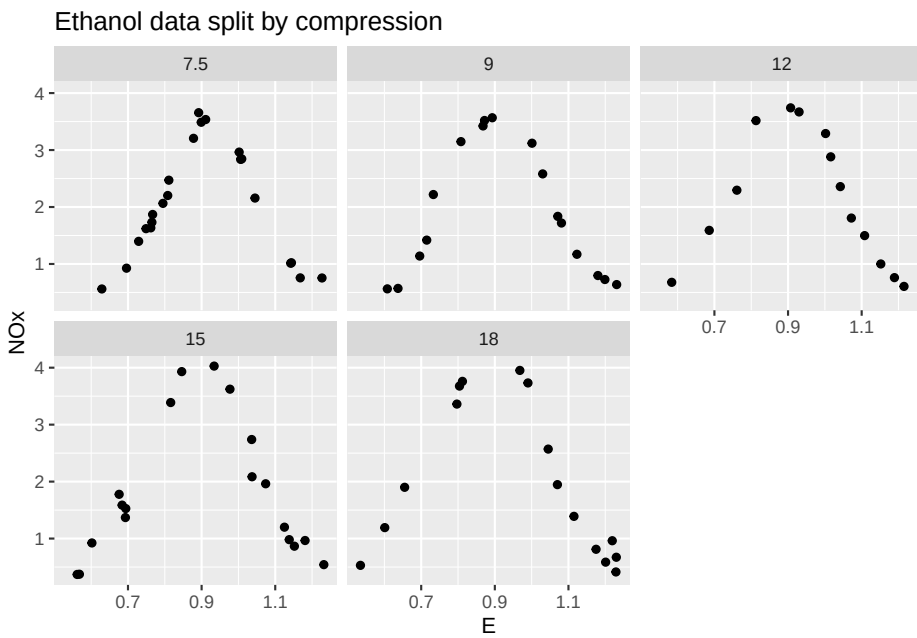
This isn't much clearer. Instead we can try:

- Faceting on one of the explanatory variables
- Mapping one of the explanatory variables to a color scheme

5.1.1 Conditioning on and coloring by compression ratio

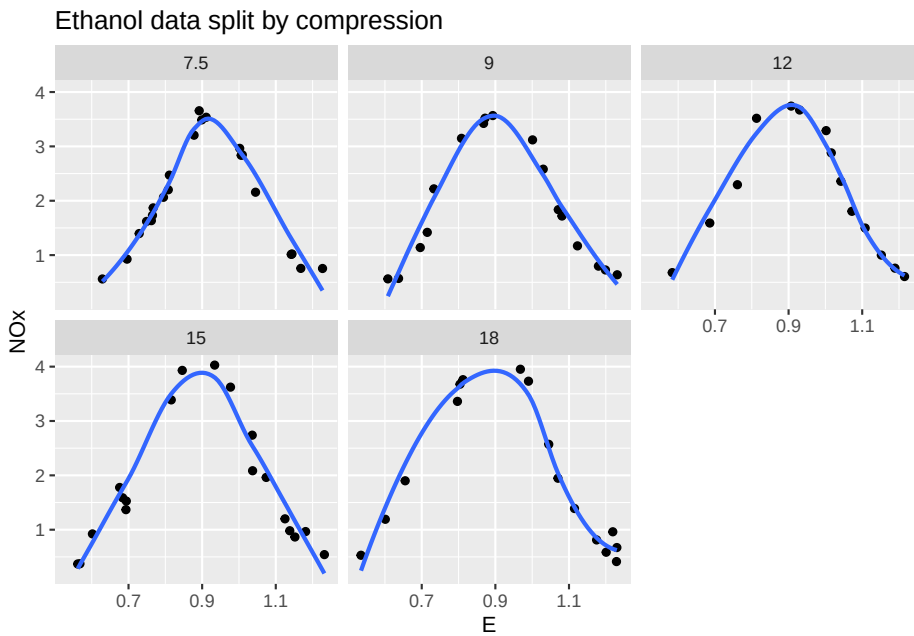
Let's first condition on C: since there are only five values, this is more straightforward. For each C, what's the relationship between E and NOx?

```
gg = ggplot(ethanol, aes(x=E, y=NOx)) + geom_point() + facet_wrap(~C, ncol=3) +
  labs(title="Ethanol data split by compression")
gg
```



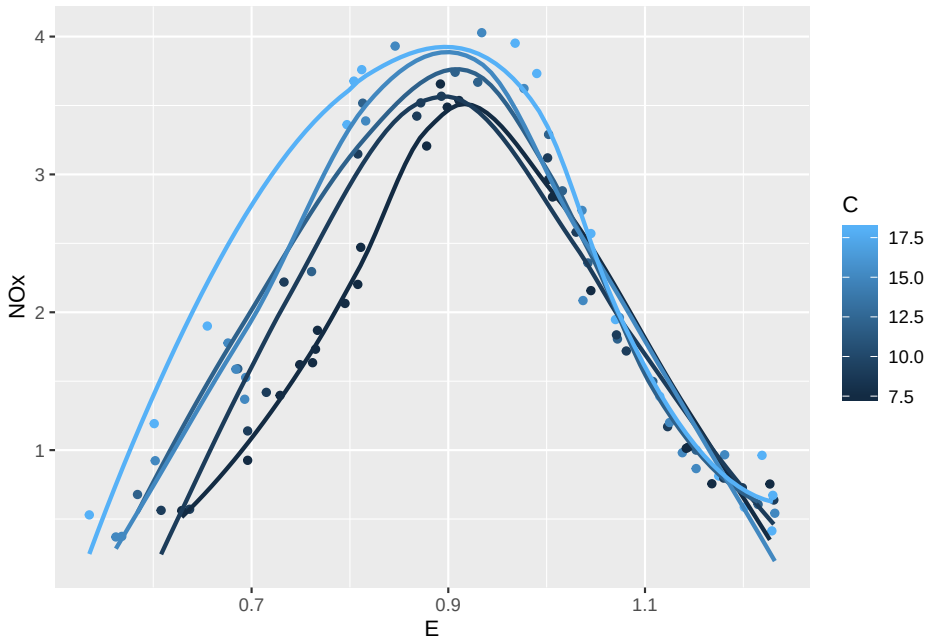
This is very obviously nonlinear, so there's no point in adding an `lm()` fit. Instead, if you don't specify a particular method, `geom_smooth()` will fit a curve:

```
gg + geom_smooth(se = FALSE)
```



The default smoothing method for moderate sample sizes is called **loess**; we won't worry about the details for now and just look at the output. The shape is very similar across all five plots: the trend is increasing up to a peak at an equivalence ratio of about 0.9, then a decline. It's hard to tell the difference in level (for example, which curve peaks the highest) from these graphs. That kind of comparison would be easier if we plotted all five of these curves on one graph. To make the graph clear, we use a *continuous* color scale. Plot each curve in a different color, where the color varies along a gradient with compression ratio. (We turn off the standard errors for clarity.)

```
ggplot(ethanol, aes(x=E, y=NOx, group=C, color=C)) + geom_point() + geom_smooth(se=FALSE)
```

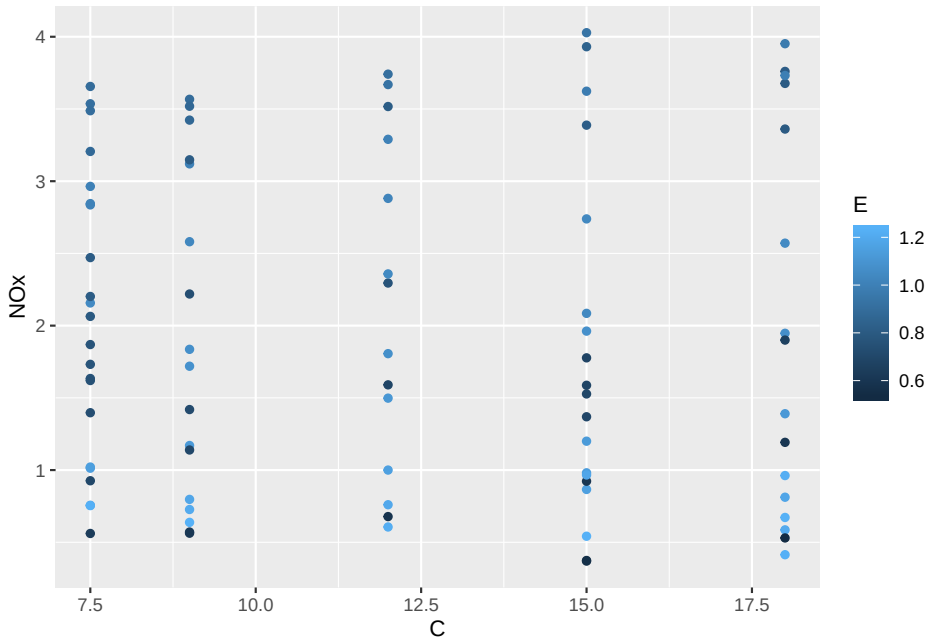


The lower compression ratios are darker. We can now see the highest values of C give the highest peak value of NOx. Note that the curves aren't simply shifted: they're relatively far apart on the left-hand side and relatively close on the right.

5.1.2 Coloring by and conditioning on equivalence ratio

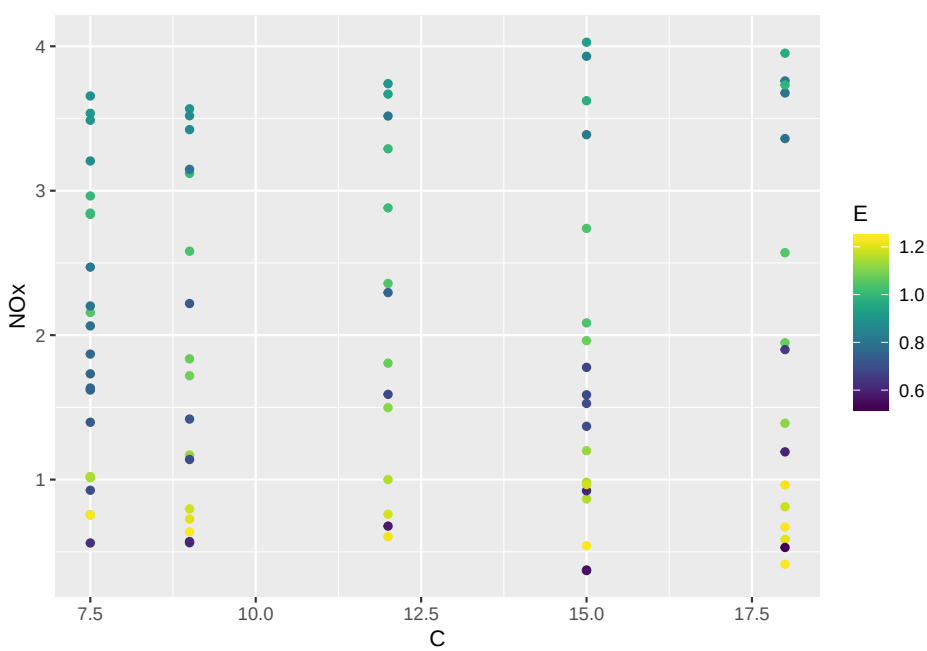
Now let's see how the relationship between C and NOx varies with E. Color the points according to E.

```
ggplot(ethanol, aes(x=C, y=NOx, group=E, color=E)) + geom_point()
```



This is a bit hard to read. A nonlinear relationship gives us an excuse to use a more dramatic color scheme. The `viridis` package is a good choice:

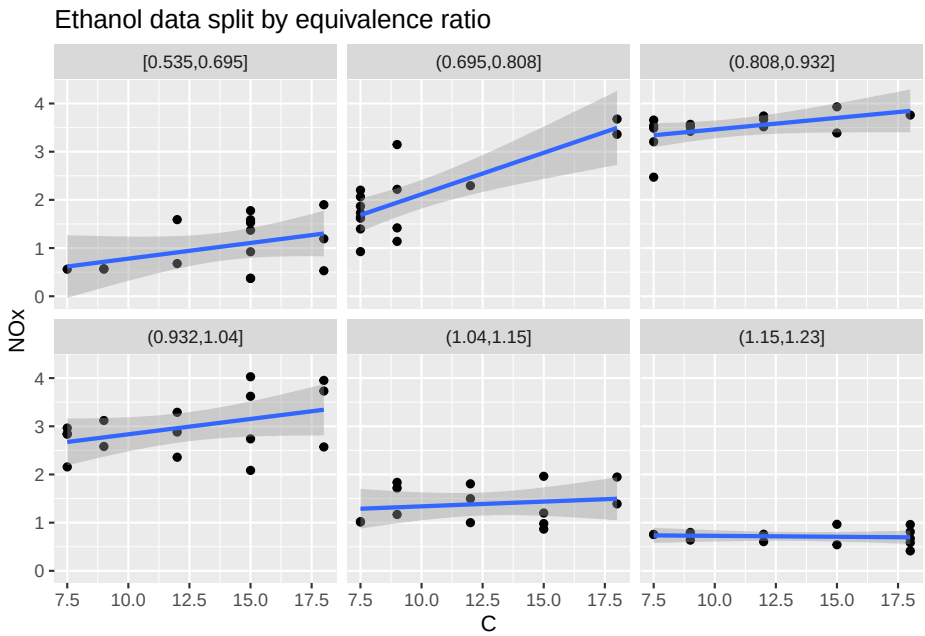
```
library(viridis)
ggplot(ethanol, aes(x=C, y=NOx, group=E, color=E)) + geom_point() + scale_color_viridis()
```



As we saw before, for low E (purple/blue), NOx is low; for middling E (green), NOx is high; and for high E (yellow), NOx is low again. However, it's still hard to see how the relationship with C changes.

With these complex relationships, faceting may be a better choice than color. To facet, we need to "cut" the continuous variable `E` into pieces, and draw a panel for each piece. Let's cut E into six classes, using the `cut_number()` function. By putting this inside `facet_wrap()`, we'll plot six scatterplots of NOx against C. It turns out a linear fit seems adequate:

```
gg = ggplot(ethanol, aes(x=C, y=NOx)) + geom_point() + geom_smooth(method="lm") +
  facet_wrap(~cut_number(E, n=6), ncol=3)
gg + labs(title="Ethanol data split by equivalence ratio")
```



Once again, middling E's give the highest NOx. However, now we can also see that the slope is steepest for a fairly low E, then decreases until the line for the highest E is basically flat.

5.1.3 Fitting and visualizing a model

So far we've found:

- Conditional on equivalence ratio, NOx depends on concentration in an approximately linear way.
- Conditional on concentration, NOx depends on equivalence ratio in a non-monotonic way (i.e. it goes up and then it goes down.)

Conditioning on E gives the simpler structure: then NOx is just a linear function

of C. However, we still need an interaction between C and E.

If you like parametric models, you could try:

```
ethanol.param = lm(NOx ~ (E + I(E^2)) * C, data=ethanol)
```

Instead, we'll pursue a **semiparametric** model. `loess(NOx ~ C * E...)` by default would fit a smooth two-dimensional surface to predict NOx, but wouldn't have the conditional linearity. To achieve this, we use the `parametric` argument to specific a parametric model in C, and the `drop.square` argument to give a linear term in C rather than the default quadratic. After some (actually a lot) of trial and error, we find a span of 1/3 looks okay. Because we have possible outlier problems, we specify `family = "symmetric"` instead of using a least squares-based solution.

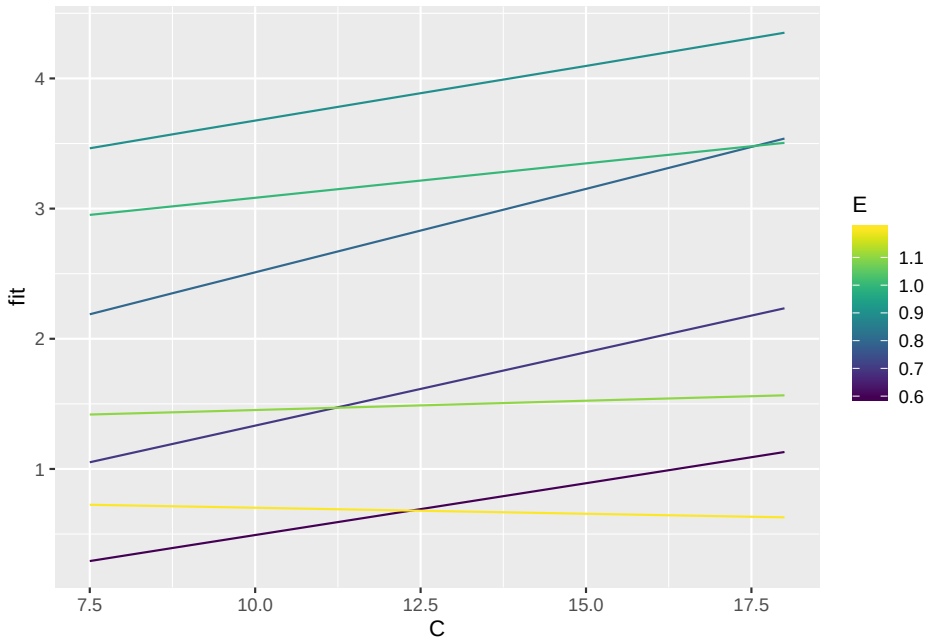
```
ethanol.lo = loess(NOx ~ C * E, data=ethanol, span=1/3, parametric="C", drop.square="C",
  family="symmetric")
```

For display, create a grid of C and E points, then predict the NOx at each point on this grid.

```
ethanol.grid = expand.grid(C=c(7.5,9,12,15,18), E=seq(0.6, 1.2, 0.1))
ethanol.predict = predict(ethanol.lo, newdata=ethanol.grid)
```

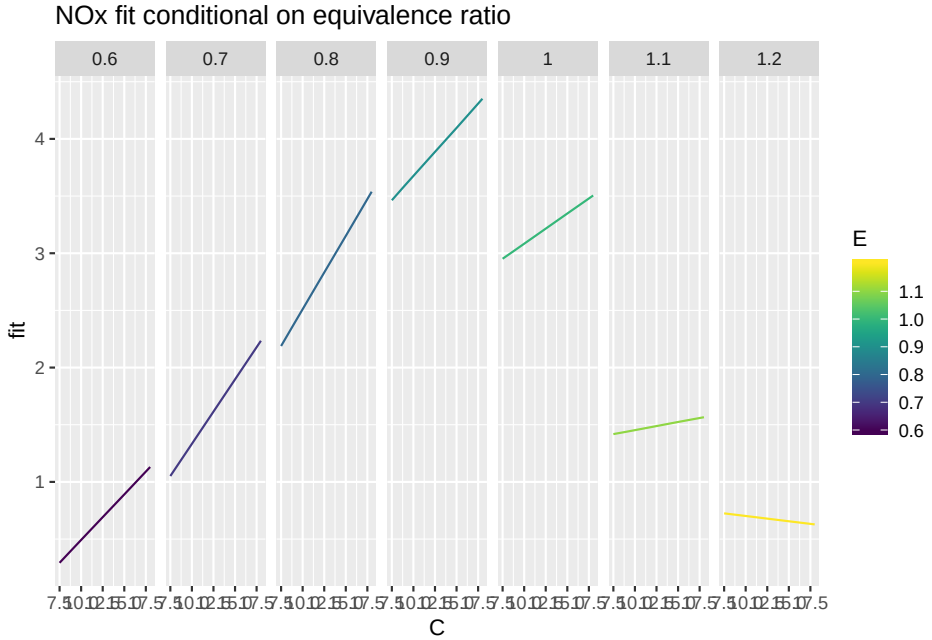
We can plot lines for different values of E on the same graph in different colors, but it gets a bit crowded.

```
ggplot(data.frame(ethanol.grid, fit=as.vector(ethanol.predict)), aes(x=C, y=fit, group=E, color=E
  geom_line() + scale_color_viridis()
```



A faceted plot of the fit faceted by values of E "spreads out" the above graph.

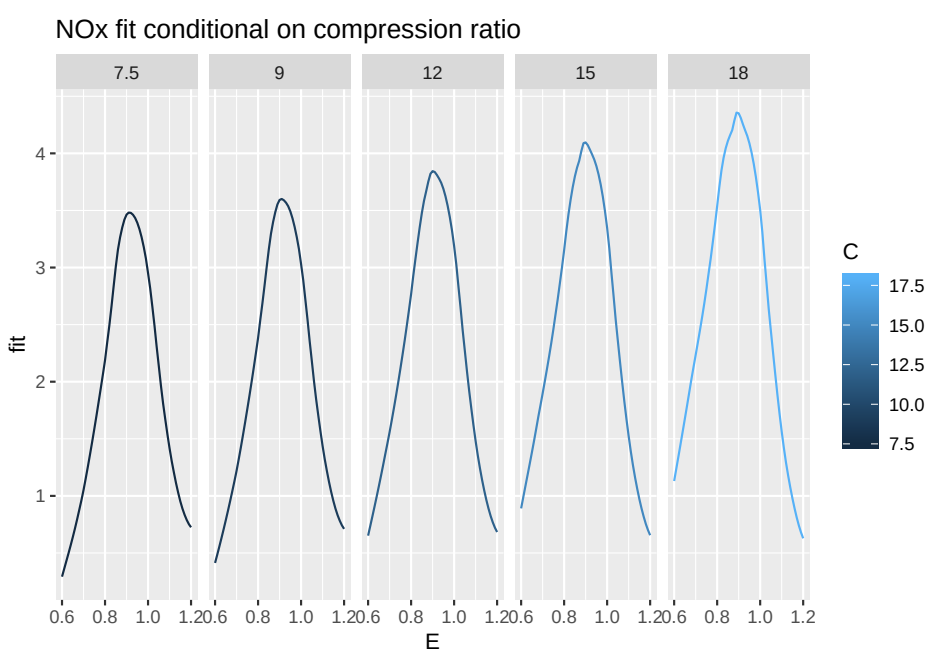
```
ggplot(data.frame(ethanol.grid, fit=as.vector(ethanol.predict)), aes(x=C, y=fit, color=
  geom_line() + facet_grid(~E) + scale_color_viridis() + labs(title="NOx fit conditiona
```



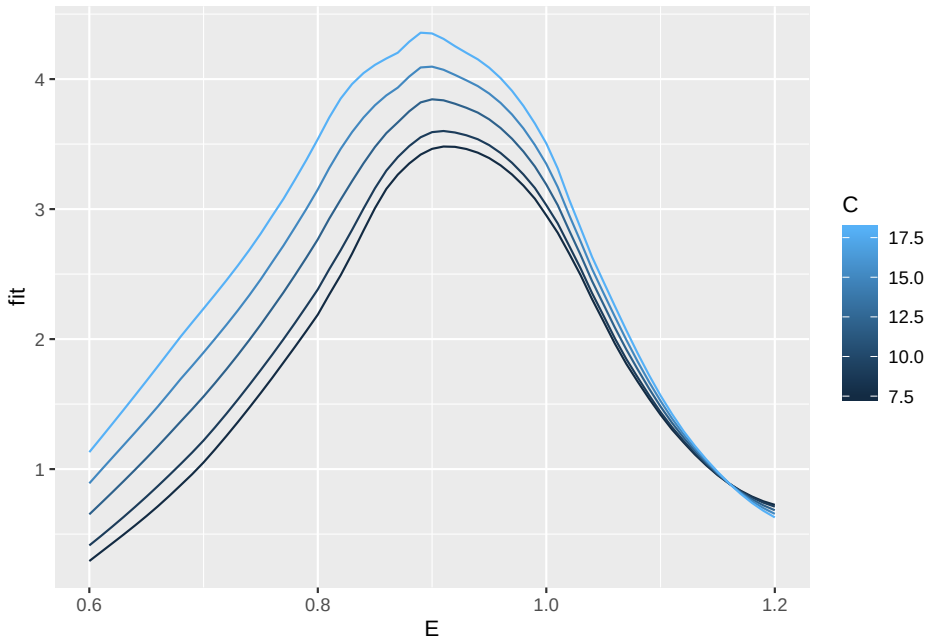
This is much clearer (and makes the graph before this one much easier to understand as well.) In addition to the higher values for E around 0.9, the slope varies with E as well. The steepest slope looks to be at around E = 0.8.

We now draw faceted plots of the fit conditioning on the compression ratio. Because of the nonlinearity, we need a tightly packed grid in E. It makes sense to use the five values of C from the experiment.

```
ethanol.grid2 = expand.grid(C=c(7.5,9,12,15,18), E=seq(0.6, 1.2, 0.01))
ethanol.predict2 = predict(ethanol.lo, newdata=ethanol.grid2)
gg = ggplot(data.frame(ethanol.grid2, fit=as.vector(ethanol.predict2)), aes(x=E, y=fit, color=C))
  geom_line() + facet_grid(~C)
gg + labs(title="NOx fit conditional on compression ratio")
```




```
ggplot(data.frame(ethanol.grid2, fit=as.vector(ethanol.predict2)), aes(x=E, y=fit, group=C, color
  geom_line()
```



The general shape is similar for all five curves. As with the raw data, the curves are separated for low E (with high C giving higher NOx), but come together for high E.

5.1.4 Exploring the residuals

Let's first look at the numerical summary of the model.

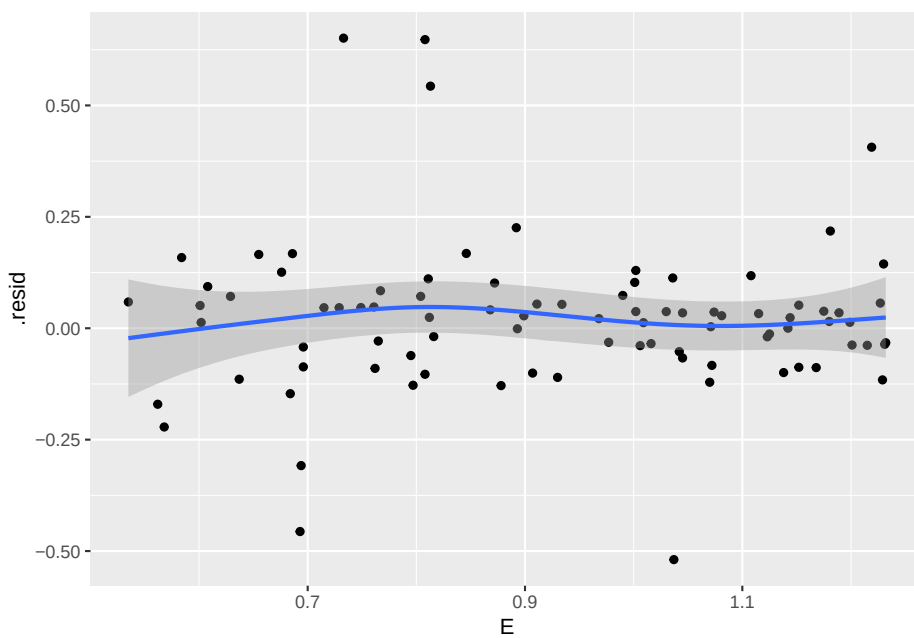
```
summary(ethanol.lo)
```

```
## Call:
## loess(formula = NOx ~ C * E, data = ethanol, span = 1/3,
parametric = "C",
## drop.square = "C", family = "symmetric")
##
## Number of Observations: 88
## Equivalent Number of Parameters: 13.59
## Residual Scale Estimate: 0.1186
## Trace of smoother matrix: 16.72 (exact)
##
## Control settings:
## span : 0.3333333
## degree : 2
## family : symmetric iterations = 4
## surface : interpolate cell = 0.2
## normalize: TRUE
```

```
## parametric: TRUE FALSE
## drop.square: TRUE FALSE
```

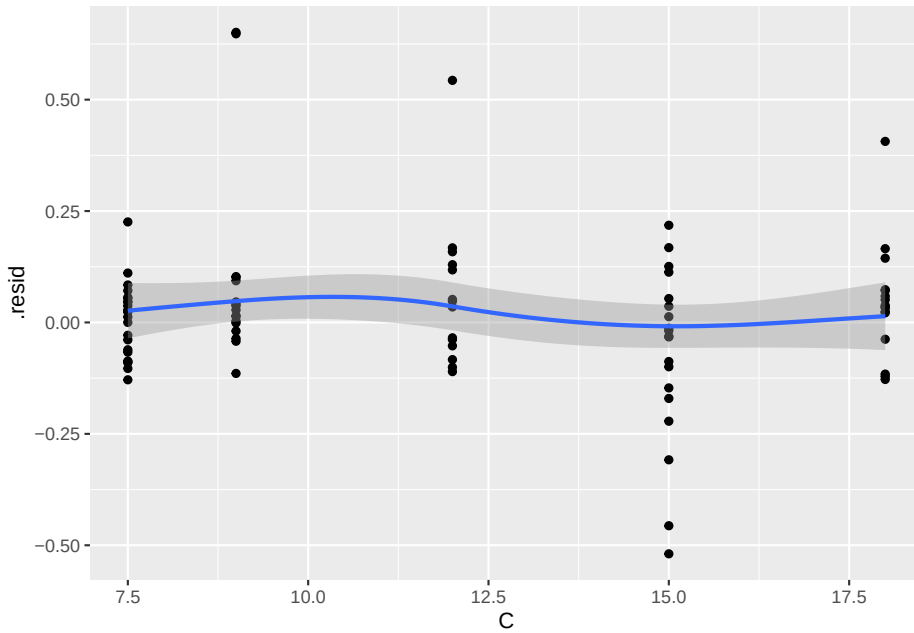
I don't know about you, but that's not very useful to me. Instead, let's construct a data frame with the original variables and the residuals, and get plotting. A good smoother for residual plots (when there's a reasonable amount of data) is a **spline**, fitted via a gam in `geom_smooth()`:

```
library(broom)
ethanol.lo.df = augment(ethanol.lo)
gg = ggplot(ethanol.lo.df, aes(x=E, y=.resid)) + geom_point()
gg + geom_smooth(method = "gam", formula = y ~ s(x))
```



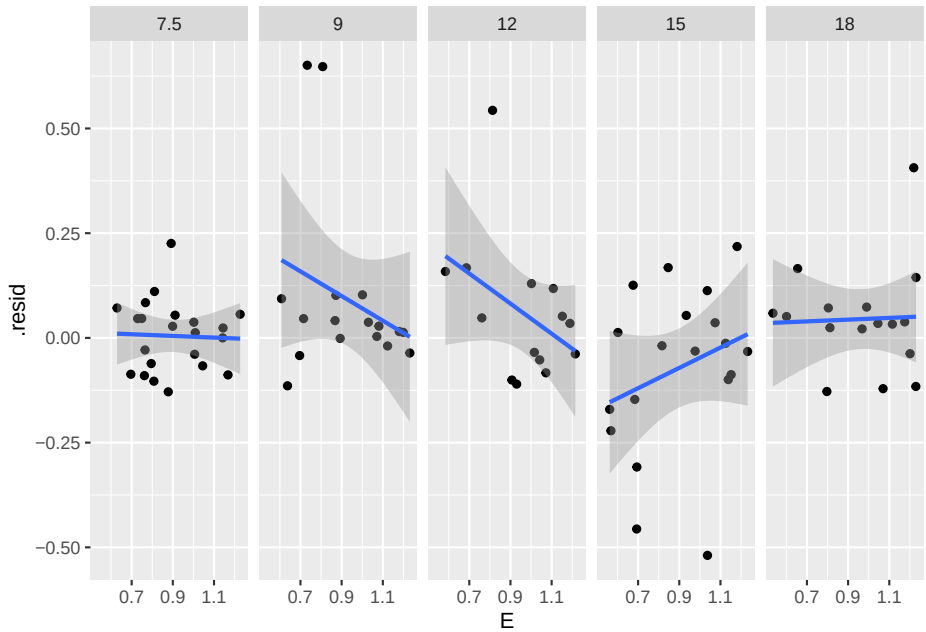
There's no obvious sign of a relationship between the residuals and the equivalence ratio. Now plot the residuals against compression. (`gam` doesn't work here because there are only five different values of C, so fit a loess instead)

```
gg = ggplot(ethanol.lo.df, aes(x=C, y=.resid)) + geom_point()
gg + geom_smooth(method = "loess", method.args = list(degree = 1))
```



The residuals for $C = 9$ are a little concerning: they do seem to be centred a little above zero. While this could be due to nonlinearity, it could also be measurement error or just the one large positive residual. To investigate further, let's condition on C and look at the relationship between the residuals and the equivalence ratio.

```
ggplot(ethanol.lo.df, aes(x=E, y=.resid)) + geom_point() + geom_smooth(method = "lm") +
```

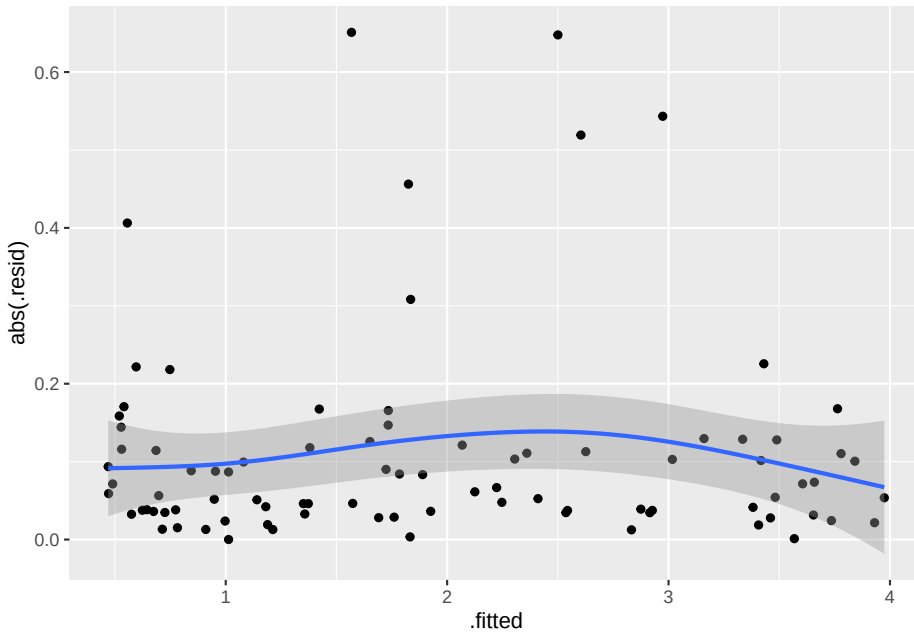


There doesn't seem to be any systematic relationship in the trends of the slopes. The outliers present a minor problem, but if you ignore them there doesn't seem to be too much pattern in the residuals. The fit is probably adequate.

5.1.5 Checking the residuals

We now do the boring stuff. Check for homoskedasticity:

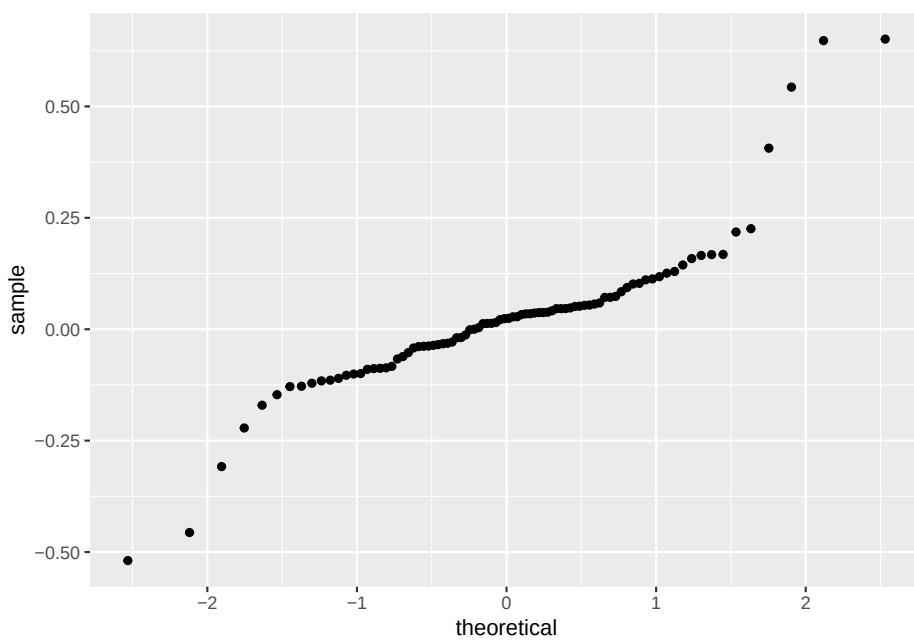
```
ggplot(ethanol.lo.df, aes(x = .fitted, y = abs(.resid))) + geom_point() +
  geom_smooth(method = "gam", formula = y ~ s(x))
```



The trend in this plot is reasonably consistent with a horizontal line. There's negligible evidence of heteroskedasticity.

Next, check normality:

```
ggplot(ethanol.lo.df, aes(sample=.resid)) + stat_qq()
```



The outliers means the residuals aren't normal, so we should hesitate to make probabilistic statements. Oh well.

Finally, did we manage to explain anything?

```
var(ethanol.lo.df$.fitted) / var(ethanol$NOx)
```

```
## [1] 0.9671507
```

```
var(ethanol.lo.df$.fitted) / (var(ethanol.lo.df$.fitted) + var(ethanol.lo.df$.resid))
```

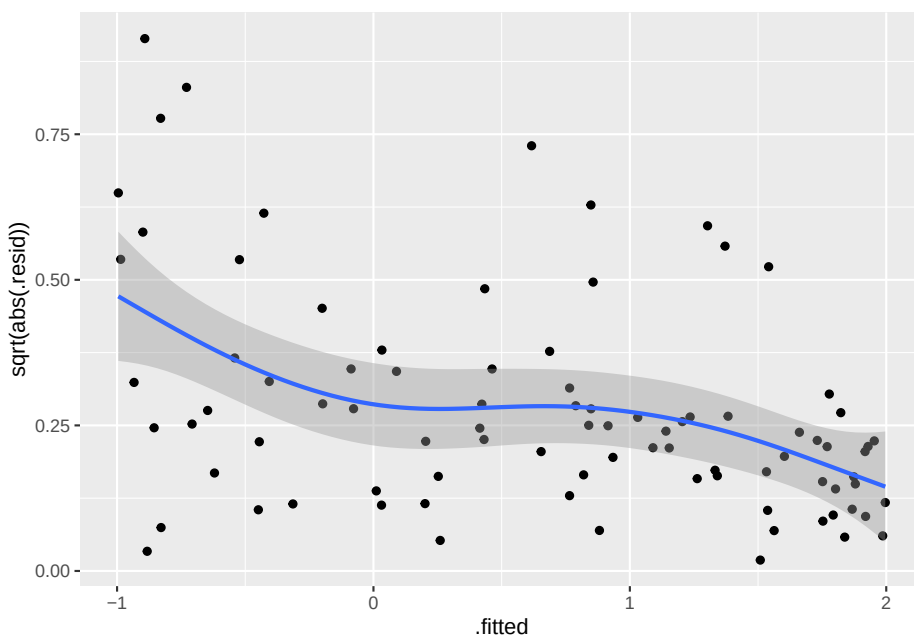
```
## [1] 0.9772901
```

The model captures 97-98% of the variation in the NOx measurements (depending on your definition of R^2.)

5.1.6 Should we have transformed?

We can try out a log transformation and see if it does any better. Let's skip to the residuals and check for homoskedasticity:

```
# Compare log NOx fit
ethanol.log.lo = loess(log2(NOx) ~ C * E, span=1/3, parametric="C", drop.square="C",
  family="symmetric", data=ethanol)
ethanol.log.lo.df = augment(ethanol.log.lo)
ggplot(ethanol.log.lo.df, aes(x=.fitted, y=sqrt(abs(.resid)))) + geom_point() +
  geom_smooth(method = "gam", formula = y ~ s(x))
```



The spread of the residuals decreases as the fitted values increase. The log fit

does worse than the fit on the original scale.

We conclude:

- NOx depends on equivalence ratio in a non-monotonic way.
- Conditional on equivalence ratio, NOx depends on concentration in an approximately linear way.
- The interaction is important: there's no real way to remove it from the data.
- The usual inference based on an assumption of normal errors is inappropriate.
- Transformations don't appear to help and may make things worse.

5.2 Burning rubber

READ: Cleveland pp. 180–187, 200–213.

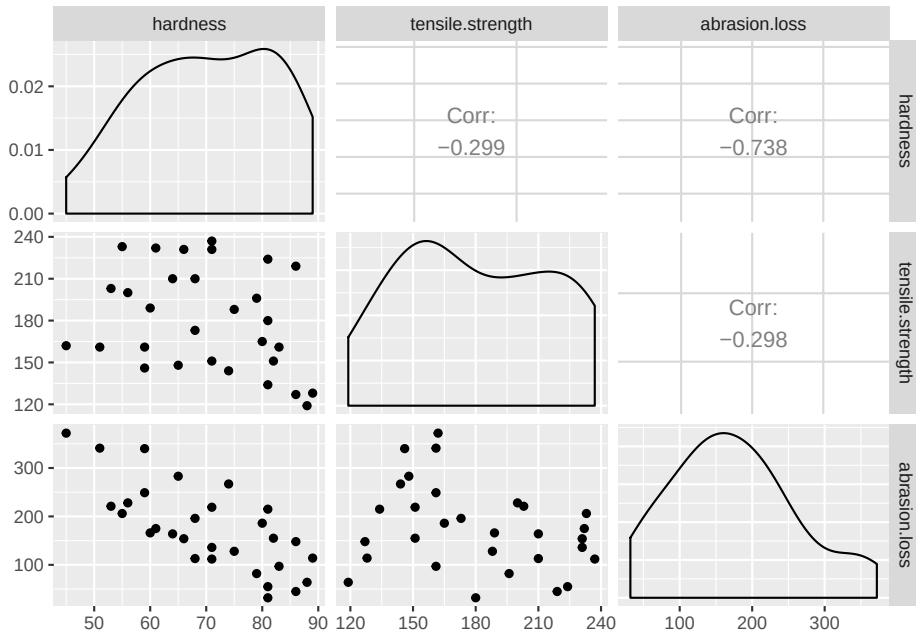
5.2.1 The rubber data

The data frame `rubber` in `lattice.RData` contains three measurements on 30 specimens of tire rubber:

- `hardness`: how much the rubber rebounds after being indented (in Shore degrees.)
- `tensile.strength`: the force per cross-sectional area required to break the rubber (in kg/cm^2.)
- `abrasion.loss`: the amount of material lost to abrasion when rubbing it per unit energy (in grams per hp-hour.) This gives you an idea how fast the tire will wear away when you drive. If we had to choose a "response" variable, it would be this one.

We want to first look at the pairwise relationships between all three variables, so we draw C_2^3 scatterplots. We can again use `ggpairs()`:

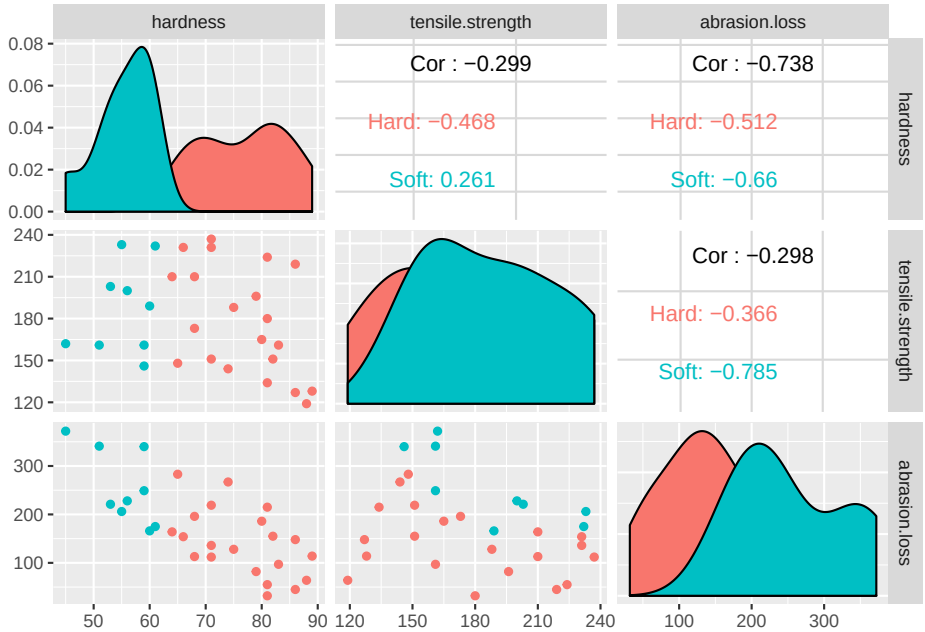
```
load("lattice.RData")
ggpairs(rubber, columns = c("hardness", "tensile.strength", "abrasion.loss"))
```



The diagonal gives density plots. There's no horrible nonlinearity in the data, so the correlations should be good summaries of the strengths of the relationships.

To get at the *trivariate* relationship and not just the bivariate ones, we can use color to indicate the value of one of the three variables. Suppose we count a hardness of over 62 as "hard" and a hardness of 62 or less as "soft". We can create a factor variable that takes these two values, then color hard and soft specimens with different colors.

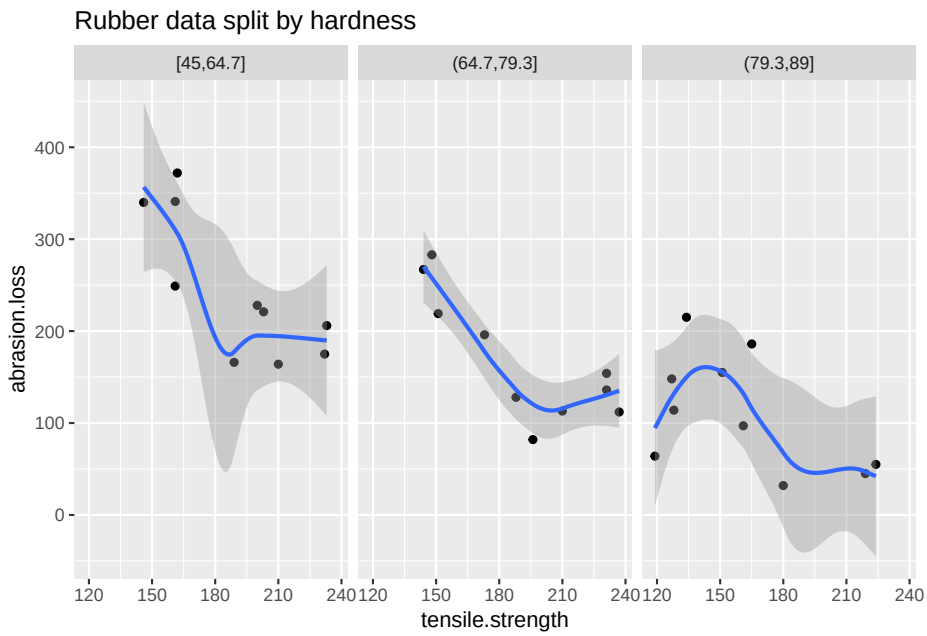
```
hard = rep(NA, nrow(rubber))
hard[rubber$hardness > 62] = "Hard"
hard[rubber$hardness <= 62] = "Soft"
rubber2 = data.frame(rubber, hard)
ggpairs(rubber2, columns = 1:3, aes(colour = hard))
```



The truly trivariate plot is the one in the middle of the bottom row. For a given tensile strength, a soft specimen tends to have more abrasion loss than a hard specimen, which seems to make physical sense.

In the above plot, we colored the points conditional on the binary hardness variable we created. Of course, we could create a hardness variable with three or more levels if we wished. After creating such a categorical variable, we can either color-code or we can draw separate plots for each value of the categorical variable. The latter approach is called a **conditional plot** or **coplot**, and is achieved through faceting. We'll cut the hardness variable into three categories, plot three scatterplots of abrasion loss against tensile strength, and add loess smoothers.

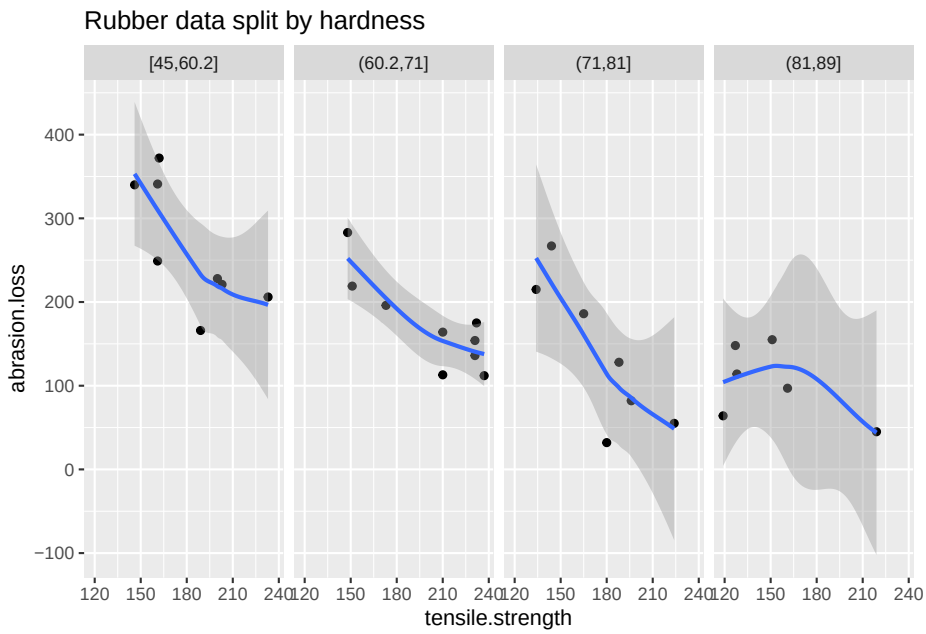
```
ggplot(rubber, aes(x=tensile.strength, y=abrasion.loss)) + geom_point() +
  geom_smooth(method.args=list(degree=1)) + facet_grid(~cut_number(hardness, n=3)) +
  labs(title="Rubber data split by hardness")
```



There's a similar decreasing relationship in each panel, while the level of the curve falls as hardness increases. In the first two plots, the curve seems to flatten out beyond a tensile strength of 180 (in the third it's hard to tell what happens because of lack of data.)

It can be worth trying out multiple cuts and seeing if you get similar results. Try a four categories hardness cut (with a high `span` because of the low number of observations after cutting.)

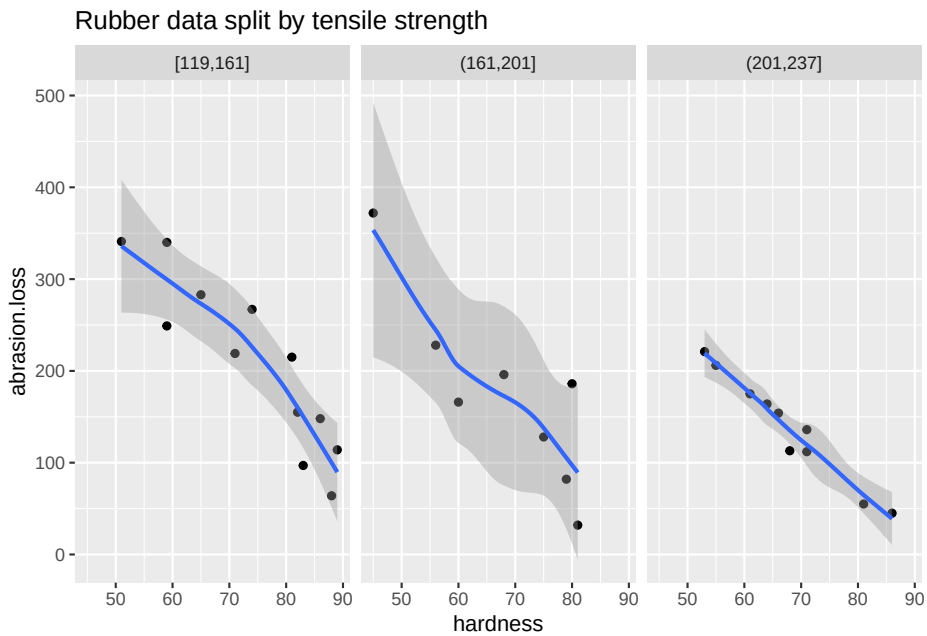
```
ggplot(rubber, aes(x=tensile.strength, y=abrasion.loss)) + geom_point() +
  geom_smooth(span=1, method.args=list(degree=1)) + facet_grid(~cut_number(hardness, n=4)) +
  labs(title="Rubber data split by hardness")
```



Again we see the relationships tend to flatten out somewhere past a tensile strength of 180.

We can also cut across tensile strength, then draw scatterplots of abrasion loss against hardness for each cut. Try three cuts for tensile strength:

```
gg = ggplot(rubber, aes(x=hardness, y=abrasion.loss)) + geom_point()
gg + geom_smooth(method.args=list(degree=1)) + facet_grid(~cut_number(tensile.strength
  labs(title="Rubber data split by tensile strength")
```



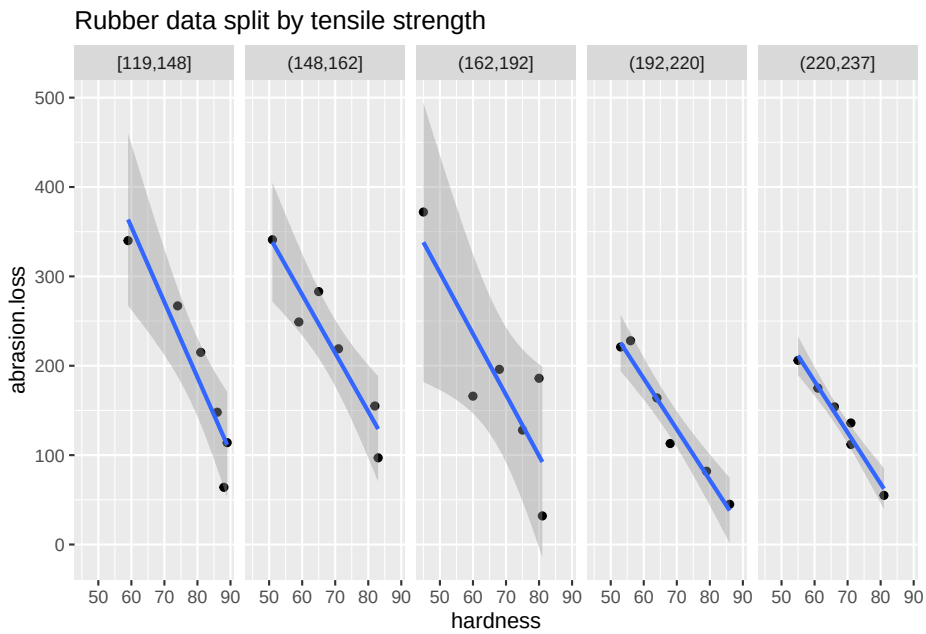
We get an approximately linear relationship in all three plots. The slope is negative and fairly similar in all three.

Try four cuts for tensile strength:

```
gg + geom_smooth(method.args=list(degree=1)) + facet_grid(~cut_number(tensile.strength, n=4)) +
  labs(title="Rubber data split by tensile strength")
```

The differences in level are a bit more apparent now – the two plots for low tensile strength have their lines much higher than for the two plots with low tensile strength. The weird thing is that the relationship isn't strictly decreasing as we go from left to right. We want to investigate further by doing more cuts, but then we start to run out of data. We can alleviate this by just fitting linear models instead of loess curves.

```
gg + geom_smooth(method="lm") + facet_grid(~cut_number(tensile.strength, n=5)) +
  labs(title="Rubber data split by tensile strength")
```



There really does seem to be a critical value for tensile strength at which the relationship changes. This is consistent with what we saw in the coplots conditioning on hardness: there was a flattening beyond a tensile strength of about 180.

This exploration leads to some suggestions for fitting a model with abrasion loss as the response.

- The model might have a critical point for tensile strength, such that the relatioship between hardness and abrasion loss is decreasing below that point and flat above that point. We can either try to carefully locate that point or just eyeball it and say it's around 180.
- The nonmonotonicity in tensile strength could be due to outliers. (*Monotonic* means the trend either always goes up or always goes down. *Nonmonotonic* means the trend is upward in some places and downward in others.) So we should strongly consider using a robust/resistant fitting method.
- There's no obvious need for interaction terms. (If the coplots had lines with very different slopes, we'd have to strongly consider them.)
- The model fitting could get pretty complex, so we should probably stick to linear models (and variations thereof) where possible.

5.2.2 Visualizing a fitted model

Let's try fitting a piecewise function in tensile strength, such that the relationship with abrasion loss is linear below 180 and constant above 180. We still want to fit using `lm()` or `rlm()`, so we rewrite as a linear model by transforming

tensile strength to "tensile strength below 180." You could do this by applying a function:

```
ts.low = function(x){
  return((x - 180) * (x < 180))
}
```

(In fact, we didn't need to do this, because we have a variable called `ts.low` in our `rubber` data frame already.)

Because there appear to be some outlying points in our data set, we prefer to fit using `rlm()` rather than `lm()`.

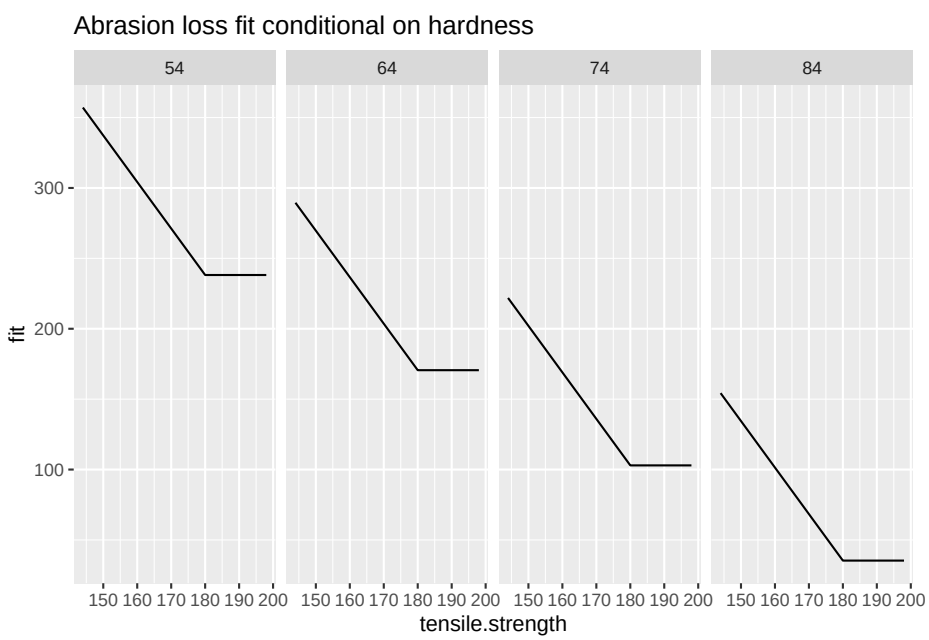
```
library(MASS)
rubber.rlm = rlm(abrasion.loss ~ hardness + ts.low(tensile.strength), data=rubber, psi=
```

To visualize the fitted surface, we want to plot a set of predictions for a grid of different values of hardness and tensile strength.

```
rubber.grid = expand.grid(hardness=c(54,64,74,84), tensile.strength=c(144,162,180,198))
rubber.predict = predict(rubber.rlm, newdata=rubber.grid)
```

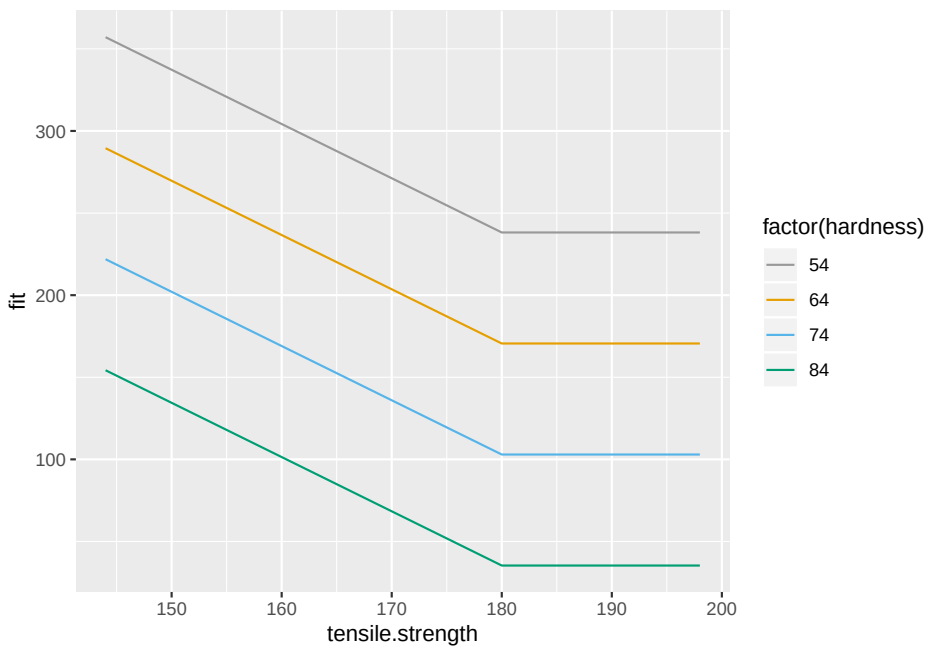
Suppose we want to see how the fit depends on tensile strength, conditioning on different values of hardness. One way to do this is to do coplots:

```
gg = ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict)), aes(x=tensile.stren
  geom_line() + facet_grid(~hardness)
gg + labs(title="Abrasion loss fit conditional on hardness")
```



Alternatively, we can plot all the lines on the same graph, and distinguish between them by color:

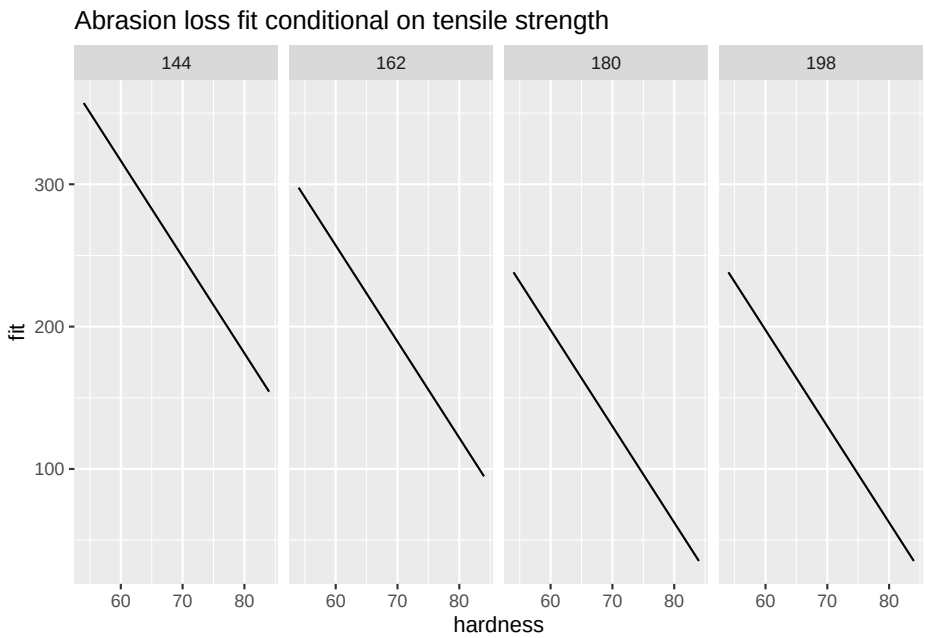
```
gg = ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict)),
  aes(x=tensile.strength, y=fit, group=hardness, color=factor(hardness))) + geom_line()
gg + scale_color_manual(values = cb_palette)
```



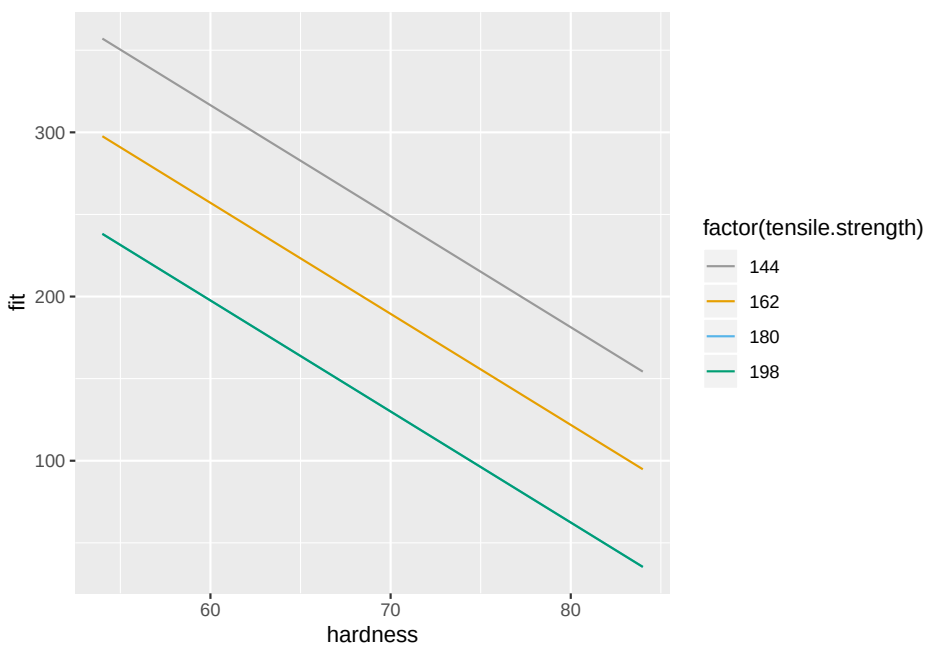
In both plots, we see that the fit becomes horizontal at 180, as we specified. Since there was no interaction in our model, the lines below 180 have the same slope.

We also want to plot the fit as a function of hardness, conditioning on different values of tensile strength. Again, we can use coplots or color.

```
gg = ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict)), aes(x=hardness, y=fit)) +
  geom_line() + facet_grid(~tensile.strength)
gg + labs(title="Abrasion loss fit conditional on tensile strength")
```




```
ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict)),
  aes(x=hardness, y=fit, group=tensile.strength, color=factor(tensile.strength))) +
  geom_line() + scale_color_manual(values = cb_palette)
```



Note that on the second plot, the line for 198 is right on top of the line for 180

– beyond 180, it doesn't make any difference what tensile strength is (according to our model.) Other than the, we just have parallel lines, which is what you should get from a linear model with no interaction.

Of course, you could always just print out the numerical summary:

```
summary(rubber.rlm)
```

```
##
## Call: rlm(formula = abrasion.loss ~ hardness +
ts.low(tensile.strength),
## data = rubber, psi = psi.bisquare)
## Residuals:
## Min 1Q Median 3Q Max
## -145.817 -22.252 -0.332 13.501 74.061
##
## Coefficients:
## Value Std. Error t value
## (Intercept) 603.2798 33.5405 17.9866
## hardness -6.7612 0.4936 -13.6965
## ts.low(tensile.strength) -3.3036 0.3107 -10.6342
##
## Residual standard error: 28.1 on 27 degrees of freedom
```

5.2.3 Exploring the residuals

Let's refit the same model as above, only this time for convenience, we'll use the `ts.low` variable in `rubber`.

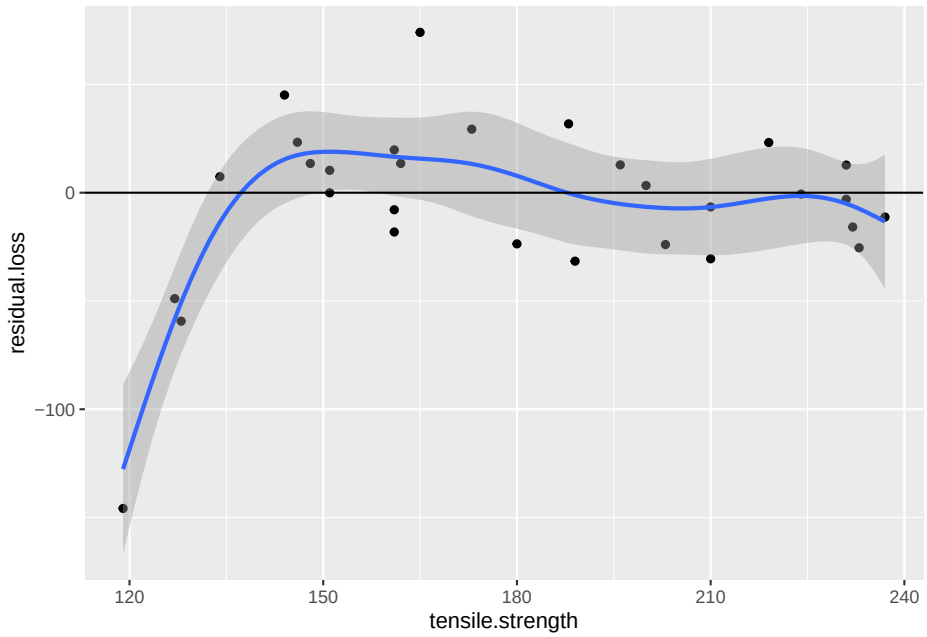
```
rubber.rlm2 = rlm(abrasion.loss ~ hardness + ts.low, data=rubber, psi=psi.bisquare)
```

We now want to collect everything we might want to use in residual plotting – the original variables, cut versions of the variables, as well as the residuals themselves – in one data frame.

```
tensile.strength = rubber$tensile.strength
tensile.cat = cut_number(tensile.strength, n=3)
hardness = rubber$hardness
hard.cat = cut_number(hardness, n=3)
residual.loss = residuals(rubber.rlm2)
rubber.rlm2.df = data.frame(tensile.strength, tensile.cat, hardness, hard.cat, residual.loss)
```

As always, we examine the residuals to look for weaknesses in our fit. First, plot the residuals as a function of tensile strength, then add a gam or loess smoother.

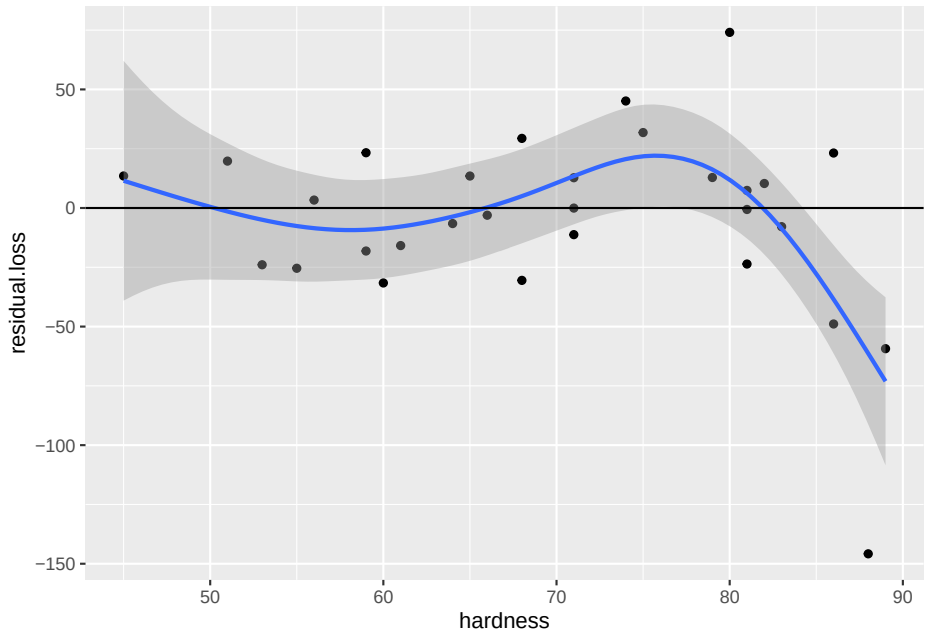
```
ggplot(rubber.rlm2.df, aes(x=tensile.strength, y=residual.loss)) + geom_point() +
  geom_smooth(method = "gam", formula = y ~ s(x)) + geom_abline(slope=0)
```



Those outliers on the left are pretty big… Putting those aside, the rest of the residuals seem randomly scattered about the zero line.

We repeat this with hardness as the explanatory variable:

```
ggplot(rubber.rlm2.df, aes(x=hardness, y=residual.loss)) + geom_point() +
  geom_smooth(method = "gam", formula = y ~ s(x)) + geom_abline(slope=0)
```



This time the three problematic points have shifted to the right. Once again, the rest of the observations look like random noise. So far, the fit seems to be a good guess despite the handful of outliers.

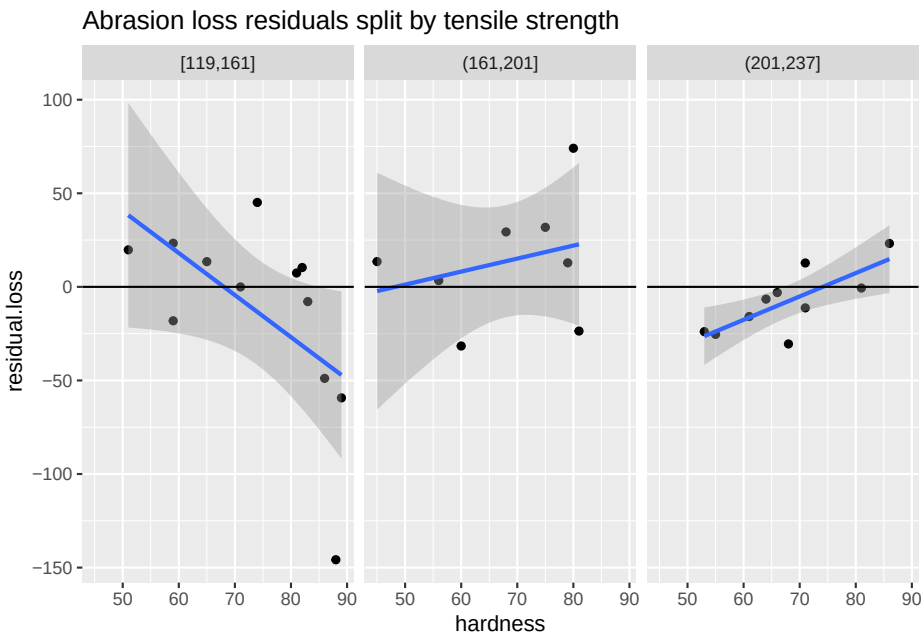
We also want to look at coplots of the residuals. We break up the data into three categories of hardness, then for each category, plot the residuals against tensile strength, adding a linear smooth.

```
gg = ggplot(rubber.rlm2.df, aes(x=tensile.strength, y=residual.loss)) + geom_point() +
  geom_smooth(method = "lm") + geom_abline(slope=0) + facet_grid(~hard.cat)
gg + labs(title="Abrasion loss residuals split by hardness")
```

This looks less good. The residual coplots for soft and medium-hardness both looks like they slope downward somewhat. The residual coplot for the hardest specimens is bent out of shape by the three outliers on the left.

Now look at coplots that condition on tensile strength.

```
gg = ggplot(rubber.rlm2.df, aes(x=hardness, y=residual.loss)) + geom_point() +
  geom_smooth(method = "lm") + geom_abline(slope=0) + facet_grid(~tensile.cat)
gg + labs(title="Abrasion loss residuals split by tensile strength")
```



We get a similar problem in the opposite direction. The low-strength graph (left) shows the high-hardness outliers distorting the picture. In the other two panels, the trend is upward, and the slope doesn't seem to be the same in the two plots.

There are enough problems that we should try to improve our model.

5.2.4 Improving the fit

The residual plots identified two major problems:

- There are three outlying points with low tensile strength, high hardness, and low abrasion loss that don't seem to fit the same pattern as the rest of the data.
- The residual coplots had different (non-zero) slopes, suggesting that we need an interaction term after all.

The interaction is easy to deal with. The main subjective decision we need to make is whether or not to drop the three outliers. I would lean toward yes, because they really are a long way from the rest of the data. However, dropping them means we should *not* use our model to say anything about rubber specimens with low tensile strength and high hardness (e.g. beyond the observation with ts 151 and hardness 82.)

We fit the model `rubber.rlm3` with an interaction to the data set with the outliers dropped, then throw everything (including fitted values and fitted value with mean removed) into a data frame `rubber.rlm3.df`.

```
rubber.no.outliers = rubber[tensile.strength>130,]
rubber.rlm3 = rlm(abrasion.loss ~ hardness * ts.low, data = rubber.no.outliers, psi=psi
tensile.strength = rubber.no.outliers$tensile.strength
tensile.cat = cut_number(rubber.no.outliers$tensile.strength, n=3)
hardness = rubber.no.outliers$hardness
hard.cat = cut_number(rubber.no.outliers$hardness, n=3)
fitted.loss = fitted.values(rubber.rlm3)
fit.demeaned = fitted.values(rubber.rlm3) - mean(fitted.values(rubber.rlm3))
residual.loss = residuals(rubber.rlm3)
rubber.rlm3.df = data.frame(
  tensile.strength,
  tensile.cat,
  hardness,
  hard.cat,
  fitted.loss,
  fit.demeaned,
  residual.loss)
```

Let's try drawing residual coplots again. First, condition on hardness:

```
gg = ggplot(rubber.rlm3.df, aes(x=tensile.strength, y=residual.loss)) + geom_point() +
  geom_smooth(method = "lm") + geom_abline(slope=0) + facet_grid(~hard.cat)
gg + labs(title="Abrasion loss residuals split by hardness")
```

There's no longer any systematic problem. The small slopes in the added smooths could easily be random noise.

Now condition on tensile strength.

```
gg = ggplot(rubber.rlm3.df, aes(x=hardness, y=residual.loss)) + geom_point() +
  geom_smooth(method = "lm") + geom_abline(slope=0) + facet_grid(~tensile.cat)
gg + labs(title="Abrasion loss residuals split by tensile strength")
```

The middle plot has an upward-sloping trend, but this just seems like luck based on the way tensile strength happened to be cut. (We could test out different cuts, but this seems like overkill.) It seems like our model fits the data well.

Now what does the model actually tell us? We could look at the numerical summary:

```
summary(rubber.rlm3)
```

```
##
## Call: rlm(formula = abrasion.loss ~ hardness * ts.low, data =
rubber.no.outliers,
## psi = psi.bisquare)
## Residuals:
## Min 1Q Median 3Q Max
## -30.6476 -8.0941 -0.8085 11.1000 66.8164
##
## Coefficients:
## Value Std. Error t value
## (Intercept) 531.5812 35.2843 15.0656
## hardness -5.7893 0.5081 -11.3942
```

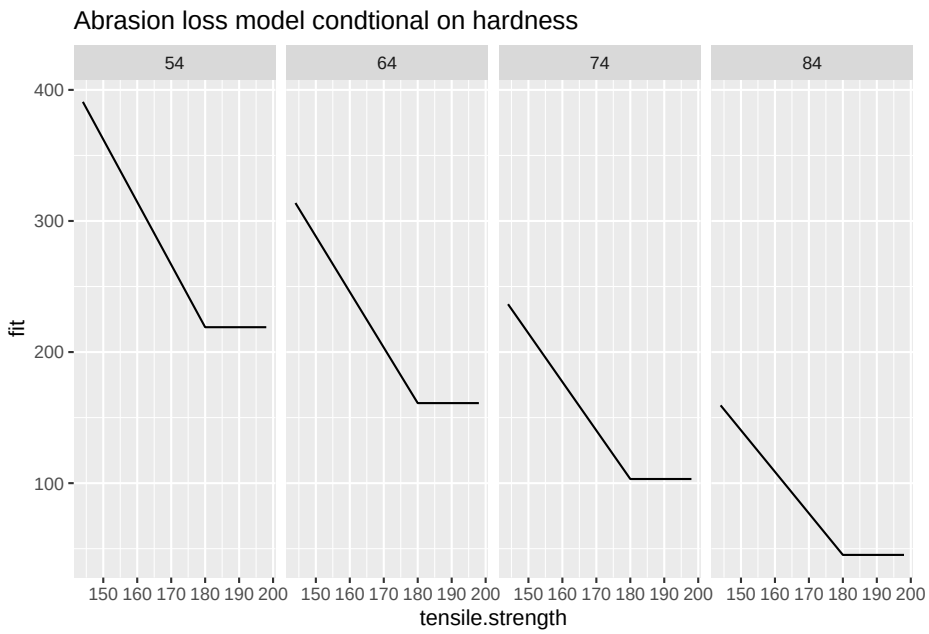
```
## ts.low -7.6652 2.0844 -3.6775
## hardness:ts.low 0.0535 0.0290 1.8457
##
## Residual standard error: 16.16 on 23 degrees of freedom
```

However, it's hard to understand what the interaction term means just from the numbers. Let's do plots then. Firstly, make predictions on a grid:

```
rubber.grid = data.frame(rubber.grid, ts.low = ts.low(rubber.grid$tensile.strength))
rubber.predict3 = predict(rubber.rlm3, newdata=rubber.grid)
```

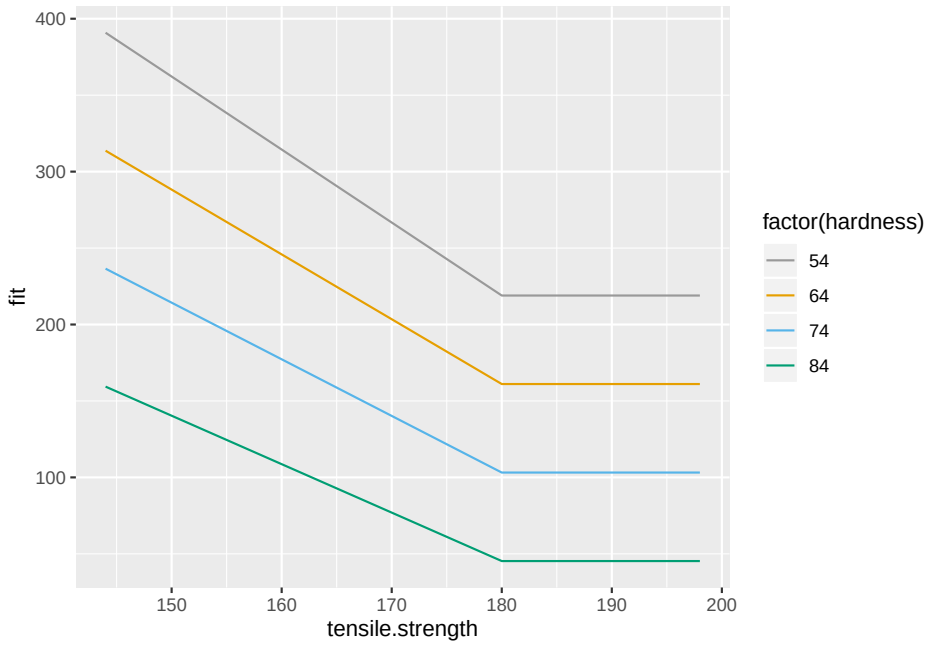
Next, coplots of the model, conditioning on hardness…

```
gg = ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict3)), aes(x=tensile.str
  geom_line() + facet_grid(~hardness)
gg + labs(title="Abrasion loss model condtional on hardness")
```



and/or plot the fits for different hardnesses in different colors.

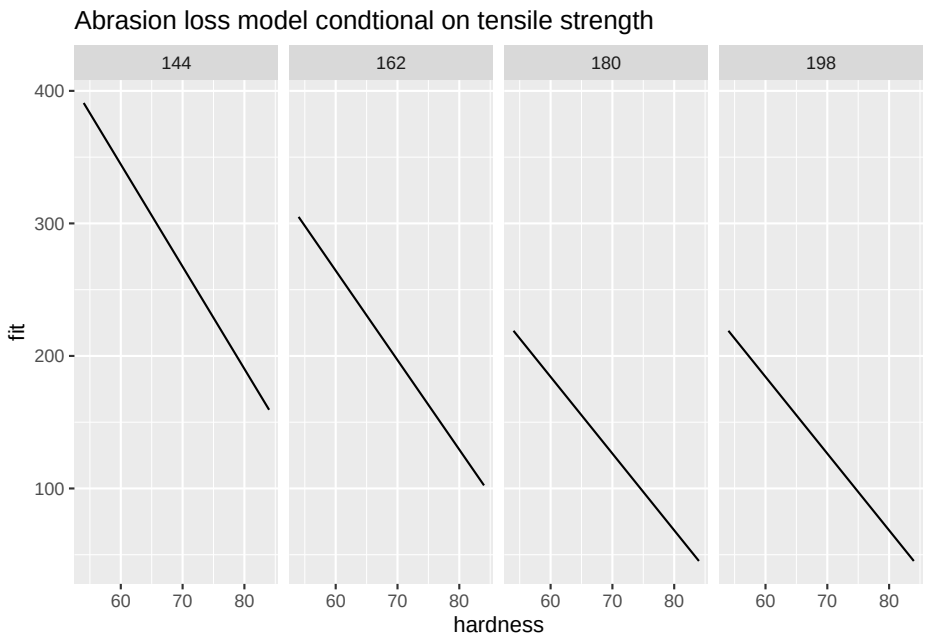
```
gg = ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict3)),
  aes(x=tensile.strength, y=fit, group=hardness, color=factor(hardness))) + geom_line(
gg + scale_color_manual(values = cb_palette)
```



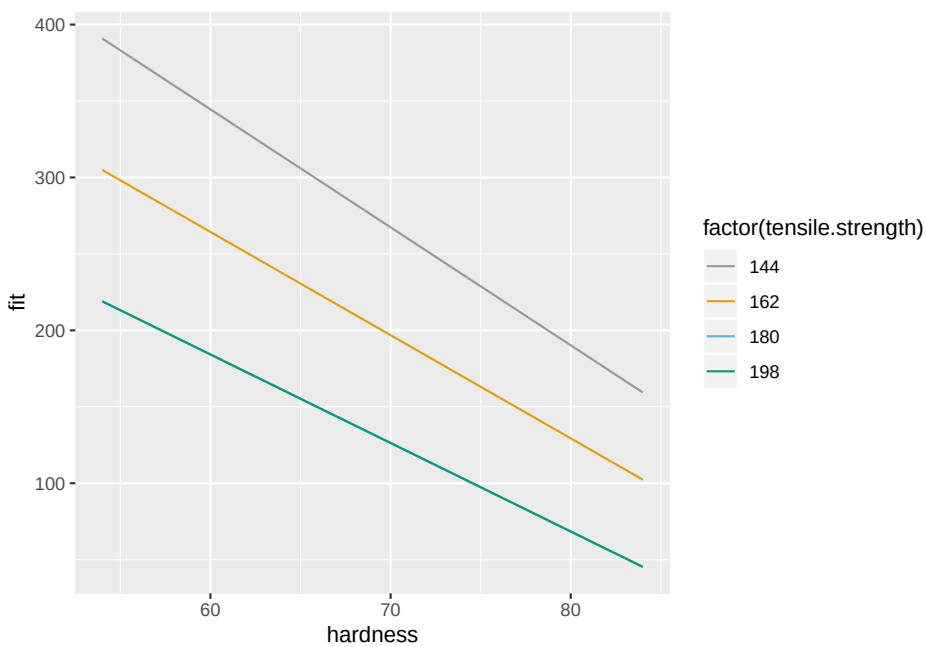
We're still fitting the same kind of function: linear below 180, constant above 180. However, the slopes are now slightly different for different values of hardness – the left-hand lines aren't parallel. The slope gets more shallow as hardness increases.

Now condition on tensile strength. Here are coplots and colored fits:

```
gg = ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict3)), aes(x=hardness, y=fit)) +
  geom_line() + facet_grid(~tensile.strength)
gg + labs(title="Abrasion loss model condtional on tensile strength")
```




```
ggplot(data.frame(rubber.grid, fit=as.vector(rubber.predict3)),
  aes(x=hardness, y=fit, group=tensile.strength, color=factor(tensile.strength))) +
  geom_line() + scale_color_manual(values = cb_palette)
```



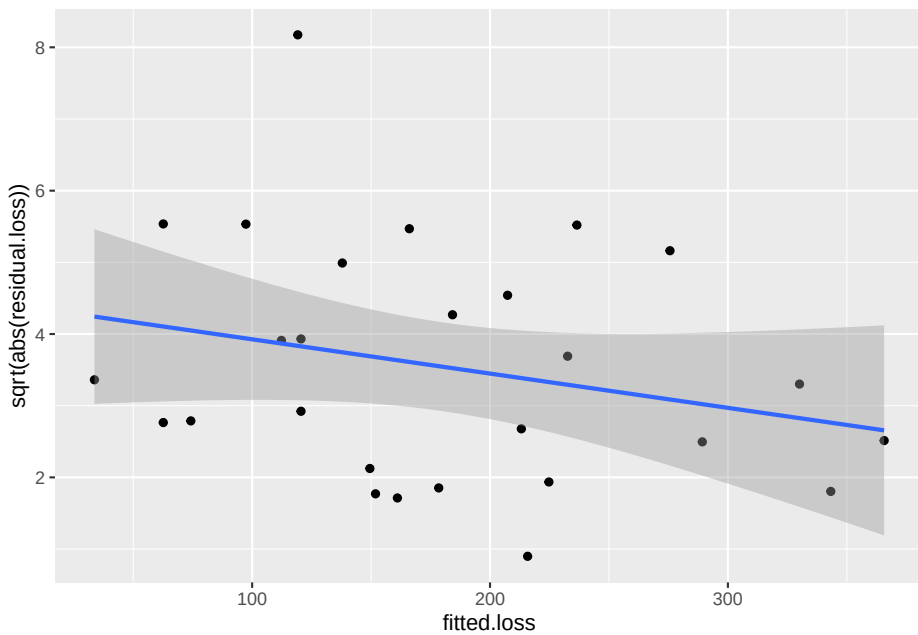
For tensile strength below 180, the slopes of the linear fits become slightly

shallower as tensile strength increases. Once tensile strength increases below 180, then by model assumption it makes no difference to either the slope or level of the fit.

5.2.5 Checking the fit

After all the fudging we did, it seems unlikely the residuals will turn nicely. Nevertheless, we should check and hope. First, draw a spread-location plot of the residuals, with the x-axis giving the fitted values and the y-axis giving the absolute residuals.

```
ggplot(rubber.rlm3.df, aes(x=fitted.loss, y=sqrt(abs(residual.loss)))) + geom_point() +
  geom_smooth(method = "gam", formula = y ~ s(x))
```



If the errors are homoskedastic, the level of the residuals under this transformation should be approximately constant. We see here that the level dropping a little going from left to right, but this might just be random, and even if it wasn't it would be unlikely that going to the effort of fixing this would be worthwhile.

Next, we want to know whether our model acutally explains much. Calculate R^2:

```
var(rubber.rlm3.df$fitted.loss) / var(rubber.no.outliers$abrasion.loss)
```

```
## [1] 0.9788679
```

The model captures 98% of the variance of the abrasion loss data after outliers

have been removed. (Of course, capturing 98% of the variance without having to remove outliers would've been much better.)

It's not really worth drawing a normal QQ plot of the residuals here, because we've already dropped outliers and seen so many imperfections that doing anything probabilistic would be misleading.

In summary, exploration has allowed us to see features of the data that aren't immediately obvious without visualization:

- The decreasing relationship between tensile strength and abrasion loss dies off for a tensile strength above about 180.
- The interaction between hardness and tensile strength is weak, but it's there.
- There are outliers with low tensile strength, high hardness, and low abrasion loss. They're different enough that they arguably should be separated from the rest of the data.

5.3 Using weights: Immigrants and income

Can the income of children of immigrants be explained by the income of the father's country and by education? This example comes from David Shor from Civis Analytics. He posted his code here:

https://gist.github.com/davidshor/3eb4c47b4e2e0d9bfbad2c656a0b58cf

It's of interest if you want to know how to use SQL statements within R. We'll just start from the data frame created on line 79. The file `immigrants.txt` contains the following variables:

- `recode`: father's country of origin
- `count`: number of individuals whose father had that country of origin
- `has_bachelor_degree` as a proportion
- `personal_income`
- `gdp_per_capita_ppp` of the father's country

```
immigrants = read.table("immigrants.txt", header = TRUE)
summary(immigrants)
```

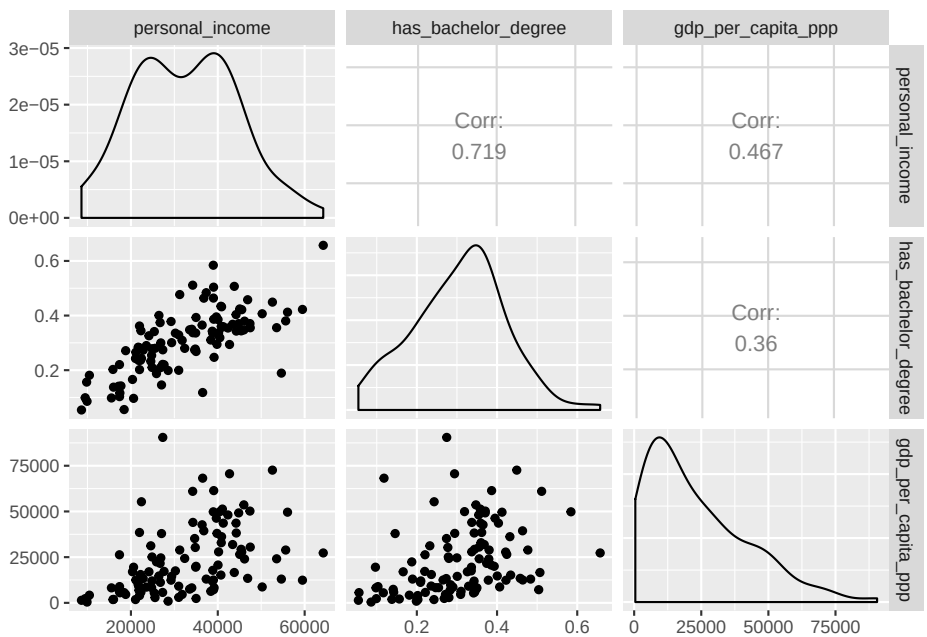
```
## recode count has_bachelor_degree
## AFGHANISTAN : 1 Min.  : 31.0 Min.  :0.05425
## ALBANIA : 1 1st Qu.: 101.5 1st Qu.:0.22964
## AMERICAN SAMOA : 1 Median : 226.5 Median :0.32249
## ANTIGUA AND BARBUDA: 1 Mean : 969.4 Mean :0.30594
## ARGENTINA : 1 3rd Qu.: 579.2 3rd Qu.:0.37520
## ARMENIA : 1 Max.  :34989.0 Max.  :0.65723
## (Other) :102
## personal_income gdp_per_capita_ppp
## Min.  : 8618 Min.  : 434
```

```
## 1st Qu.:22728 1st Qu.: 8200
## Median :32953 Median :17100
## Mean :32523 Mean :23018
## 3rd Qu.:40746 3rd Qu.:33475
## Max.  :64284 Max.  :90500
##
```

5.3.1 Exploring the data

Let's first do a standard pairs plot of the data.

```
library(GGally)
ggpairs(immigrants, columns = c("personal_income", "has_bachelor_degree",  "gdp_per_capita_ppp"))
```



GDP is strongly right-skewed, but personal income isn't really, and it would be weird to take the log of GDP but not of income. We'll start out without transformations but might change our minds.

The bigger issue is that the counts are very different between countries. You might argue that if you're fundamentally interested in the *countries*, you don't need weights, but it seems weird to count 31 children of Singaporean fathers and 35,000 children of Mexican fathers equally. We should check if weighting actually makes a difference:

```
mean(immigrants$personal_income)
```

```
## [1] 32523.03
```

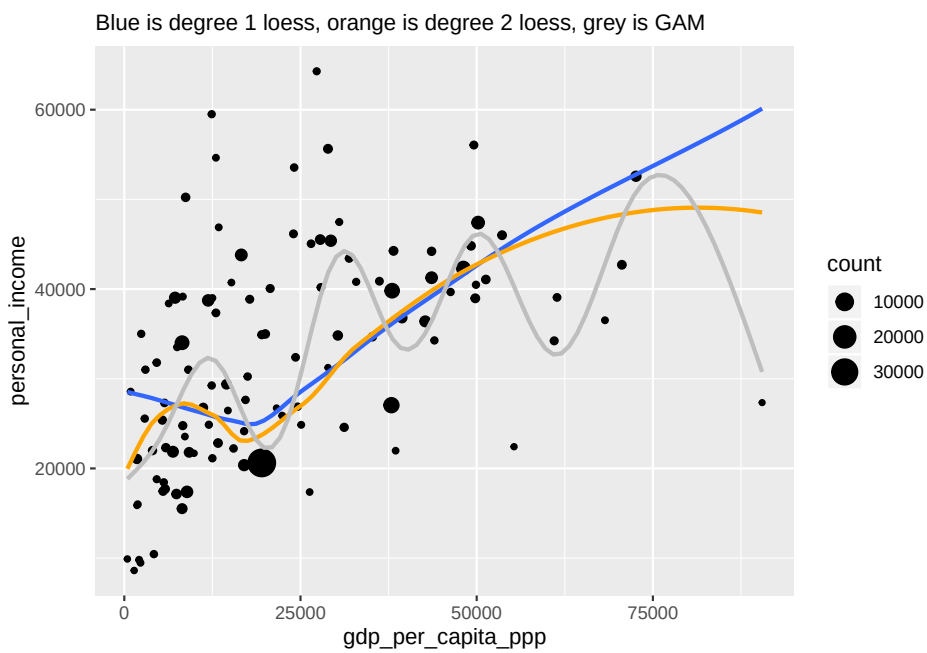
```
weighted.mean(immigrants$personal_income, immigrants$count)
```

```
## [1] 30148.3
```

This is a big enough difference that it will matter whether or not we weight. Let's say we decide that we will.

We now draw scatterplots such that (i) the area of the point is proportional to the count, and (ii) the smoother finds a weighted fit. The `size` and `weight` aesthetics take care of these respectively. We'll try degree 1 and 2 loess, as well as a GAM.

```
gg = ggplot(immigrants, aes(x = gdp_per_capita_ppp, y = personal_income)) +
  geom_point(aes(size = count)) +
  geom_smooth(aes(weight = count), method.args = list(degree = 1), se = FALSE) +
  geom_smooth(aes(weight = count), color = 'orange', se = FALSE) +
  geom_smooth(method = 'gam', formula = y ~ s(x), aes(weight = count),
              color = 'grey', se = FALSE)
gg + labs(subtitle = "Blue is degree 1 loess, orange is degree 2 loess, grey is GAM")
```

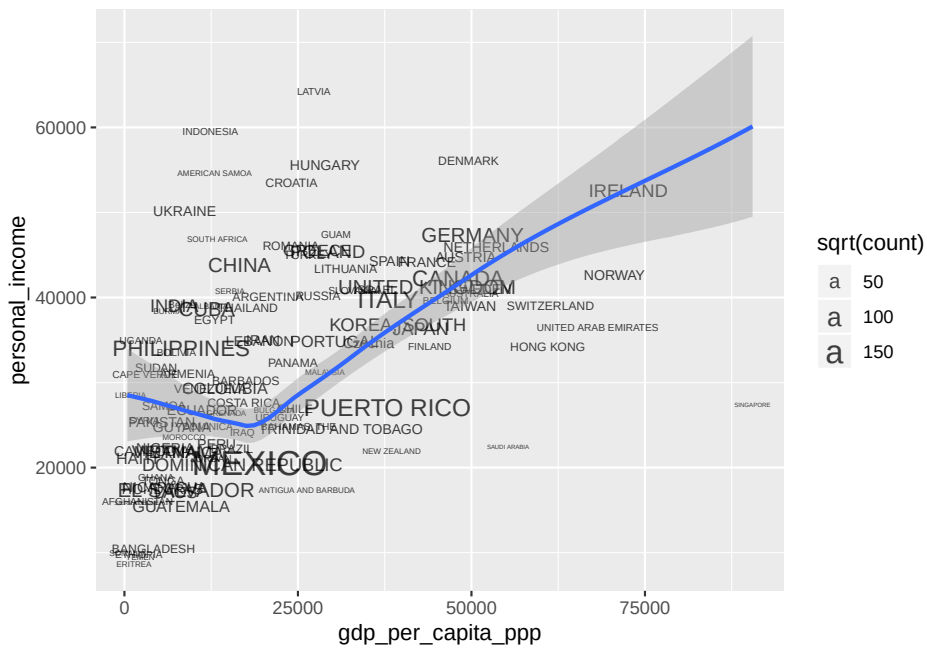


The "tick" shape of the degree 1 loess is awkward, because there isn't really any reason to suspect an underlying nonmonotonic relationship, and it's not clear that additional smoothing will help. The default degree 2 loess is worse and the gam is nonsense (GAM often doesn't handle weights well), so we won't pursue it.

If which country is at which point is of high interest, we can text labels instead.

Resizing the labels is controversial (as the text area will no longer be proportional to the count) but it does emphasize the countries with more observations, which is what you want.

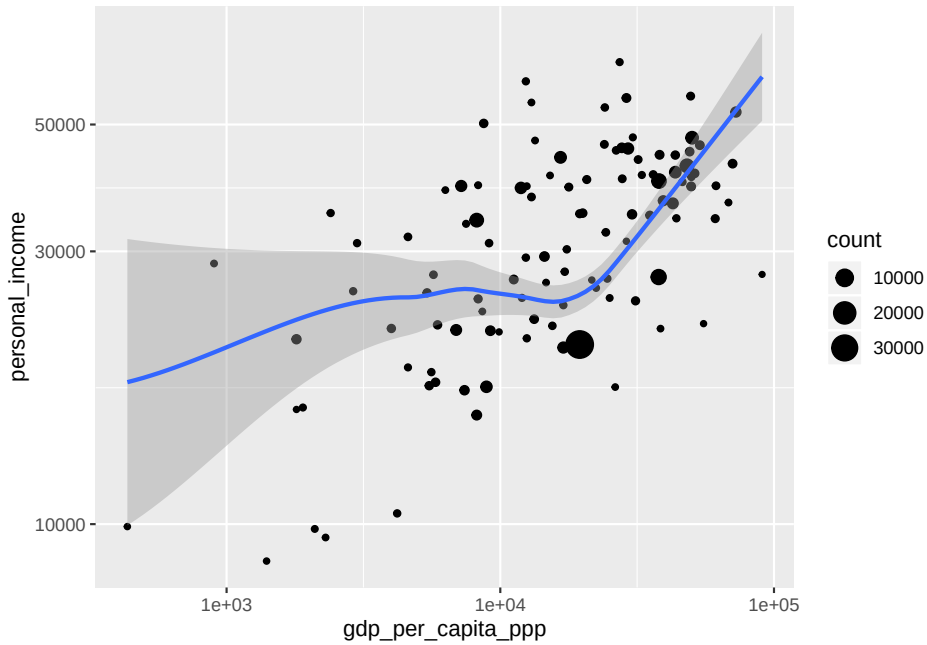
```
gg = ggplot(immigrants, aes(x = gdp_per_capita_ppp, y = personal_income)) +
  geom_text(aes(label = recode, size = sqrt(count)), alpha = 0.75)
gg + geom_smooth(aes(weight = count), method.args = list(degree = 1))
```



The "tick" shape is mostly or entirely due to Mexico. In the context of U.S. immigration, you might declare Mexico a special case and drop it. But it's not exactly optimal to drop a third of your data from your analysis.

Perhaps a log-log helps?

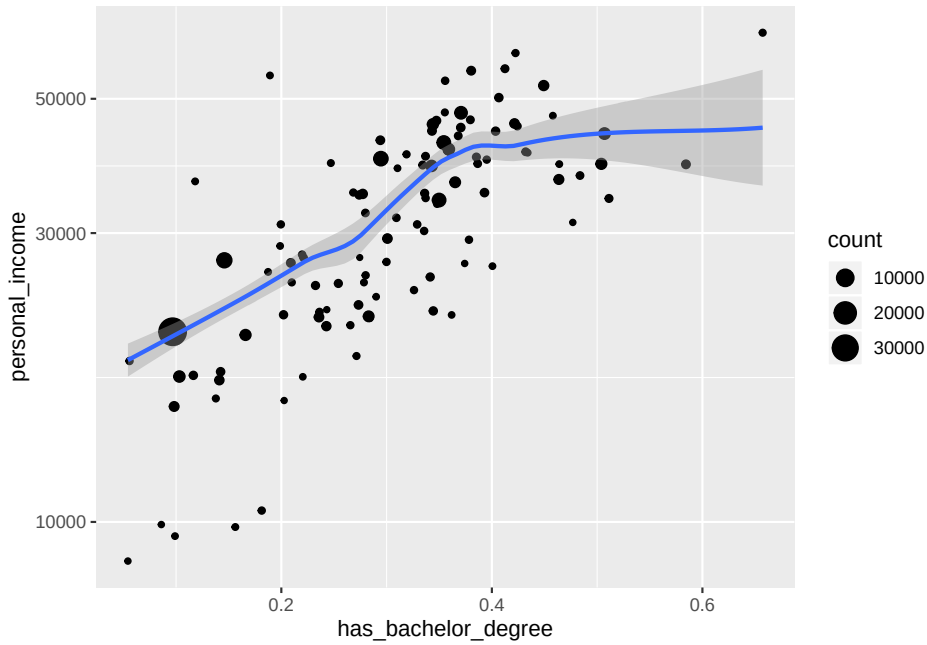
```
ggplot(immigrants, aes(x = gdp_per_capita_ppp, y = personal_income)) +
  geom_point(aes(size = count)) + scale_x_log10() + scale_y_log10() +
  geom_smooth(aes(weight = count), method.args = list(degree = 1))
```



It does a little bit: the amount of nonmonotonicity is now pretty small.

Now plot log income against education:

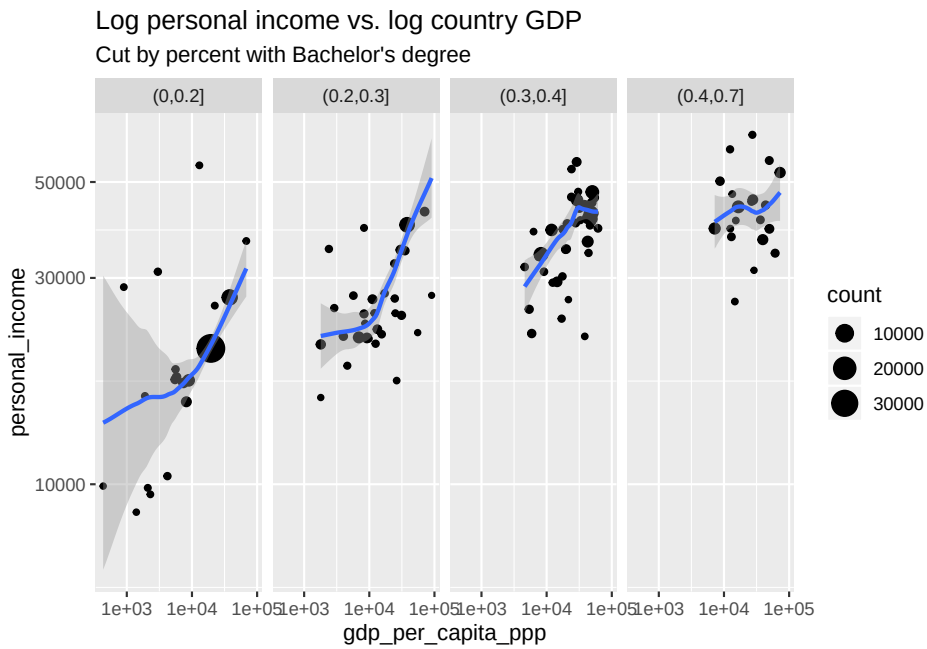
```
ggplot(immigrants, aes(x = has_bachelor_degree, y = personal_income)) +
  geom_point(aes(size = count)) + scale_y_log10() +
  geom_smooth(aes(weight = count), method.args = list(degree = 1))
```



The curve flattens out when the percent with a bachelor's gets beyond about 38%.

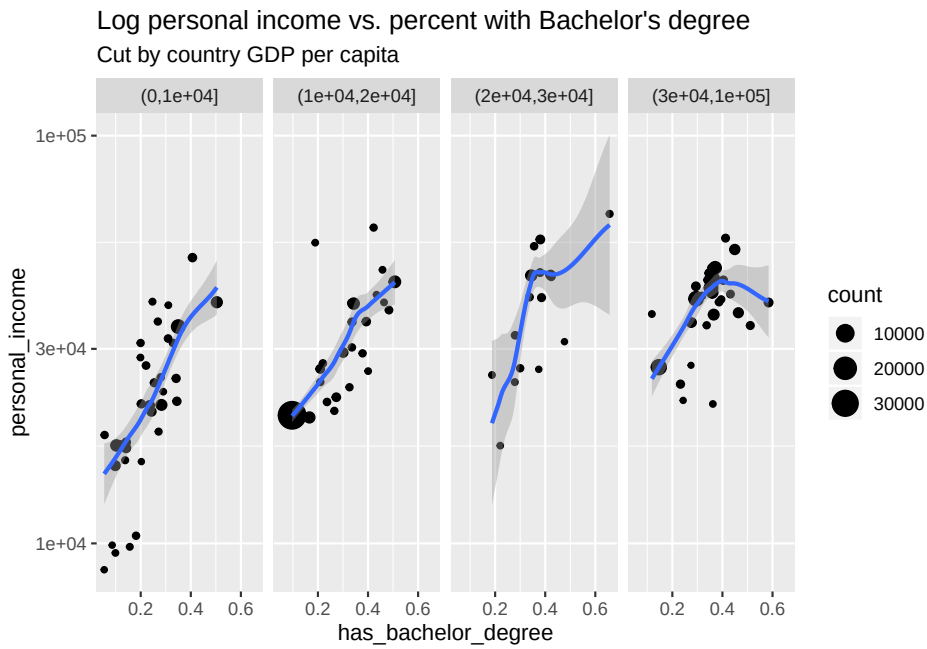
Now we draw faceted plots. For interpretability, we can choose our own cut points, roughly corresponding to quartiles of the variables, heavily rounded.

```
gg = ggplot(immigrants, aes(x = gdp_per_capita_ppp, y = personal_income)) +
  geom_point(aes(size = count)) + scale_x_log10() + scale_y_log10() +
  geom_smooth(aes(weight = count), method.args = list(degree = 1)) +
  facet_grid(~cut(has_bachelor_degree, c(0, 0.2, 0.3, 0.4, 0.7)))
gg + ggtitle("Log personal income vs. log country GDP") +
  labs(subtitle = "Cut by percent with Bachelor's degree")
```



Whatever's going on here, it doesn't appear to be a simple additive shift. So fitting a model with an interaction will be safer.

```
gg = ggplot(immigrants, aes(x = has_bachelor_degree, y = personal_income)) +
  geom_point(aes(size = count)) + scale_y_log10() +
  geom_smooth(aes(weight = count), method.args = list(degree = 1)) +
  facet_grid(~cut(gdp_per_capita_ppp, c(0, 10000, 20000, 30000, 100000)))
gg + ggtitle("Log personal income vs. percent with Bachelor's degree") +
  labs(subtitle = "Cut by country GDP per capita")
```



Note that there's a big region of the predictor space where we don't have any data (low GDP, high education.) So we shouldn't attempt to draw conclusions about this region.

5.3.2 Modeling the data

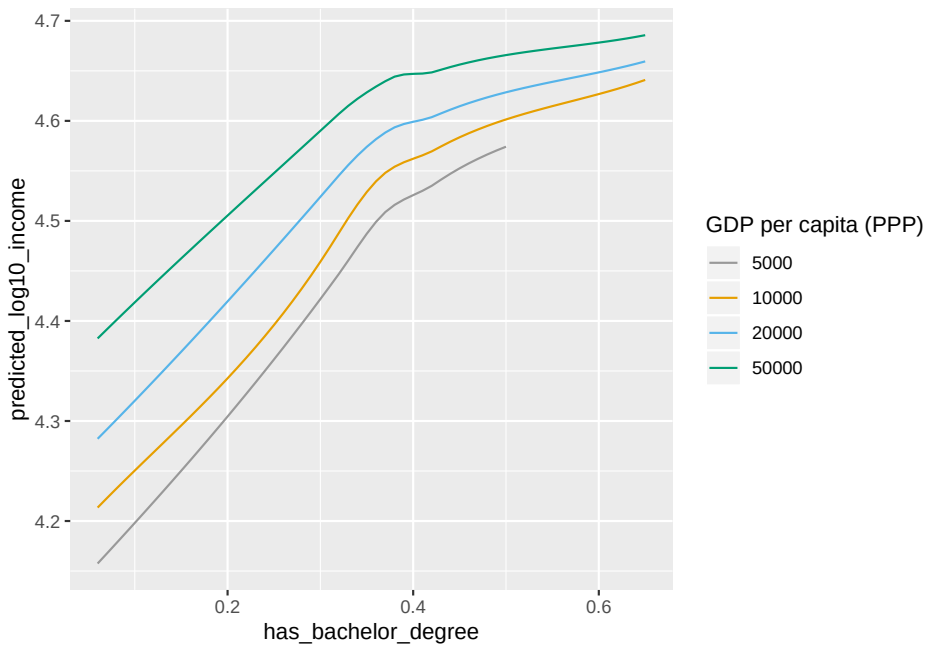
We fit a weighted degree 1 loess, which necessarily involves an interaction. (If we wanted a nonparametric model without an interaction we'd fit a **generalized additive model (GAM)** instead.)

```
immigrants.lo = loess(log10(personal_income) ~ log10(gdp_per_capita_ppp) * has_bachelor_degree,
  weights = count, degree = 1, data = immigrants)
```

We set up a grid to do prediction. We'll first plot the fit against education and condition on GDP. That requires a grid that's dense in education but sparse in GDP. We also note we have little data for very high education and low GDP, so we remove predictions in that range (let's say lower GDP and higher education than India.)

```
newdata1 = expand.grid(has_bachelor_degree = seq(0.06, 0.65, 0.01),
  gdp_per_capita_ppp = c(5000, 10000, 20000, 50000))
immigrants.pred1 = predict(immigrants.lo, newdata = newdata1)
immigrants.pred1 = data.frame(newdata1, predicted_log10_income = as.vector(immigrants.pred1))
immigrants.pred1 = immigrants.pred1[immigrants.pred1$has_bachelor_degree <= 0.5 |
  immigrants.pred1$gdp_per_capita_ppp >= 7200,]
ggplot(immigrants.pred1, aes(x = has_bachelor_degree, y = predicted_log10_income,
```

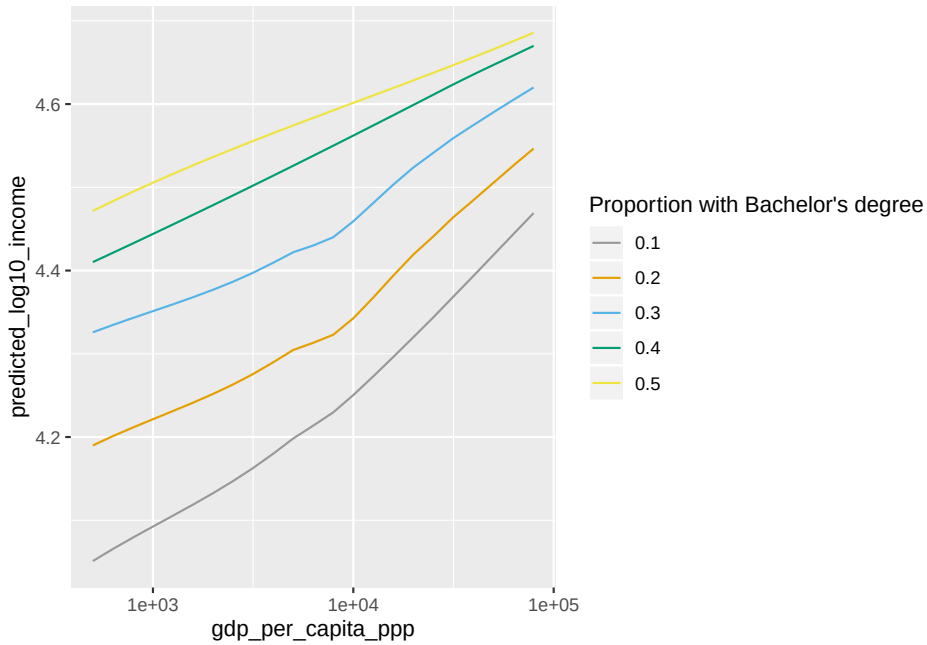
```
  group = gdp_per_capita_ppp, color = factor(gdp_per_capita_ppp))) + geom_line() +
  scale_color_manual(values = cb_palette) + labs(color = "GDP per capita (PPP)")
```



The curves get a little closer as the percent with a bachelor's increases, suggesting that in some sense as education goes up, what country your father is from matters a bit less to your (log) income.

To plot our fit with education as our main x-variable, we use a grid dense in education and sparse in GDP:

```
newdata2 = expand.grid(has_bachelor_degree = seq(0.1, 0.5, 0.1),
  gdp_per_capita_ppp = 10 ^ seq(2.7, 4.9, 0.1))
immigrants.pred2 = predict(immigrants.lo, newdata = newdata2)
immigrants.pred2 = data.frame(newdata2, predicted_log10_income = as.vector(immigrants.p
gg = ggplot(immigrants.pred2, aes(x = gdp_per_capita_ppp, y = predicted_log10_income,
  group = has_bachelor_degree, color = factor(has_bachelor_degree))) +
  scale_x_log10() + geom_line() + scale_color_manual(values = cb_palette)
gg + labs(color = "Proportion with Bachelor's degree")
```

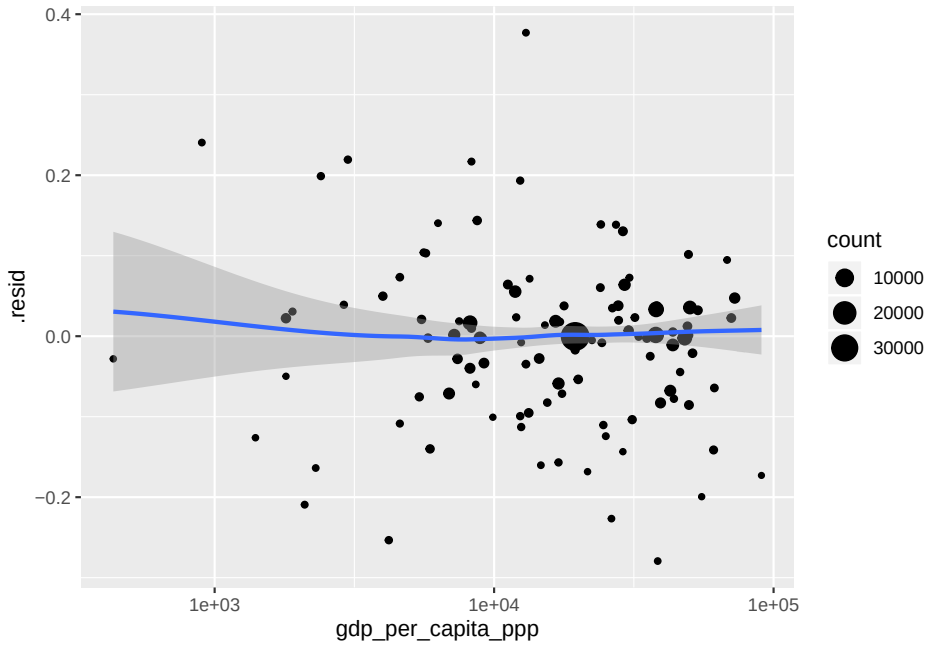


Again the curves converge toward the right hand side.

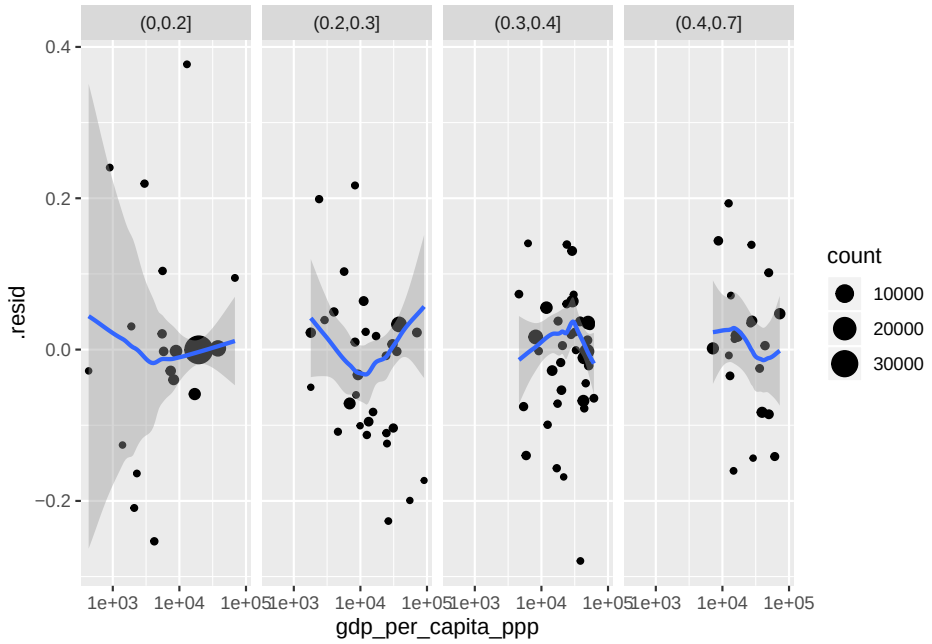
5.3.3 Residuals

Fits on our residual plots should again be weighted.

```
immigrants.lo.df = data.frame(immigrants, .resid = residuals(immigrants.lo))
gg = ggplot(immigrants.lo.df, aes(x = gdp_per_capita_ppp, y = .resid))
gg = gg + geom_point(aes(size = count)) + scale_x_log10()
gg = gg + geom_smooth(aes(weight = count), method.args = list(degree = 1))
gg
```



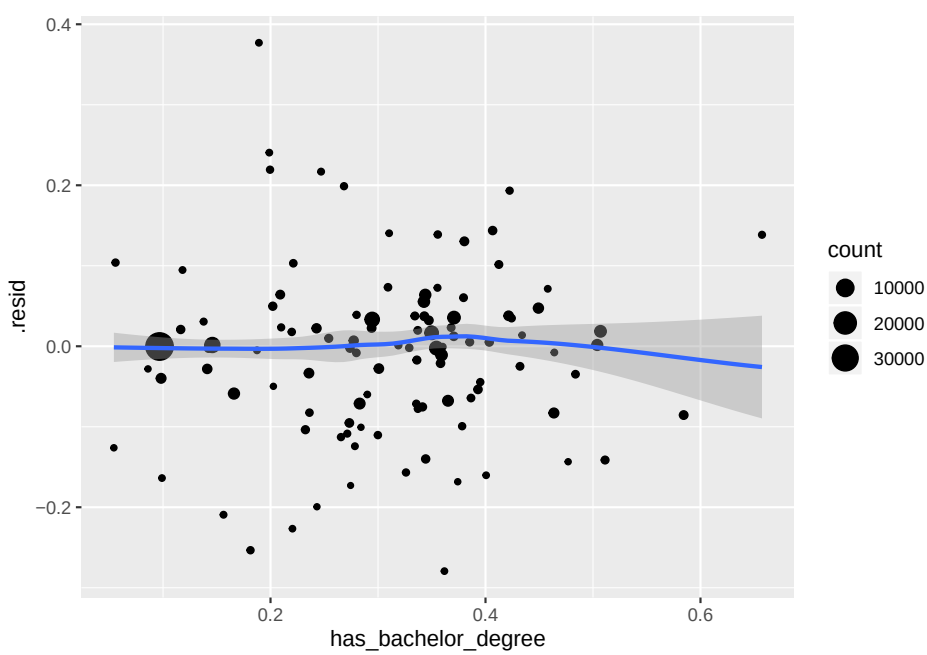

```
gg + facet_grid(~cut(has_bachelor_degree, c(0, 0.2, 0.3, 0.4, 0.7)))
```



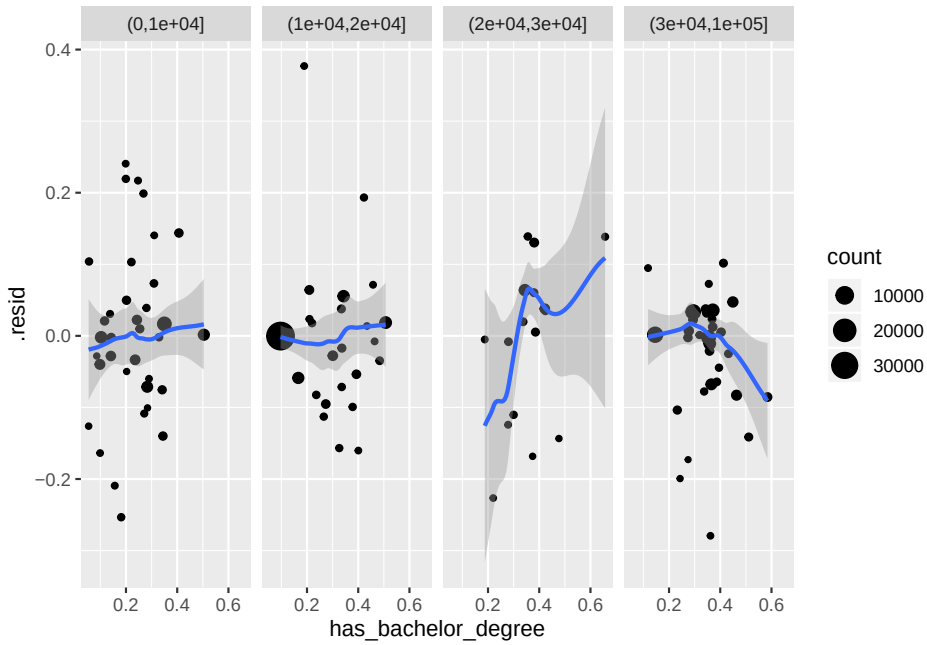
The conditional residual plots for the midrange education levels wiggle around a bit more than I would like. That suggests decreasing the span of the loess below the default of 0.75. However, decreasing it too much might make weird

stuff happen in the fit (e.g. strong non-monotonicity.)

```
gg = ggplot(immigrants.lo.df, aes(x = has_bachelor_degree, y = .resid)) +
  geom_point(aes(size = count)) +
  geom_smooth(aes(weight = count), method.args = list(degree = 1))
gg
```




```
gg + facet_grid(~cut(immigrants$gdp_per_capita_ppp, c(0, 10000, 20000, 30000, 100000)))
```



You can try adding a `span = ...` argument into the call to the loess fit, knitting this Markdown, and seeing what difference this makes.

Since we're fitting a complex model with weird data, it doesn't seem especially informative to check homoskedasticity and normality. We should check how much variance we've captured:

```
var(fitted.values(immigrants.lo))
```

```
## [1] 0.02346432
```

```
var(residuals(immigrants.lo))
```

```
## [1] 0.01127028
```

Our model has captured about two-thirds of the variation in log income (but note this is after aggregation; we would have captured much, much less than two-thirds of the variation in log income among individuals.)

5.4 Contour plots

READ: Cleveland pp. 228–248.

5.4.1 Velocities of the NGC 7531 galaxy

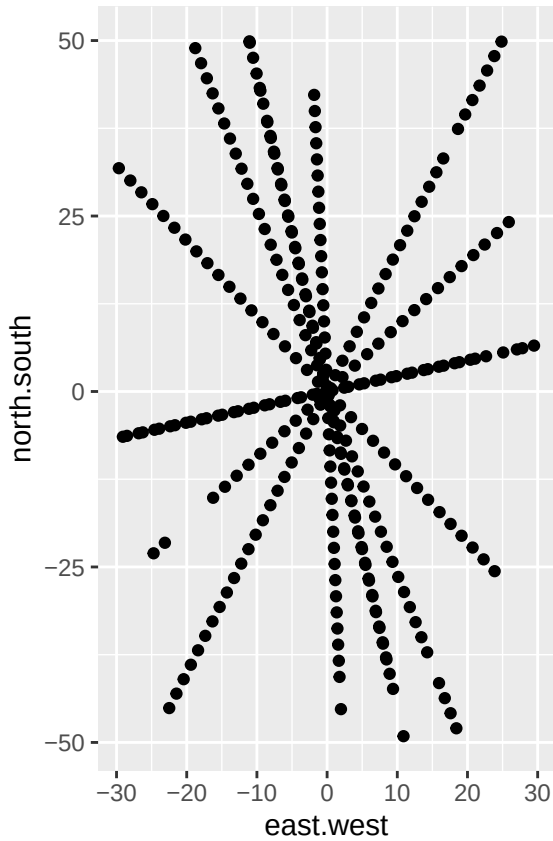
The data frame `galaxy` contains 323 measurements of velocity for locations in galaxy NGC 7531.

```r
load("lattice.RData")
summary(galaxy)
```

```
## east.west north.south angle radial.position
## Min.   :-29.66693 Min.   :-49.108 Min.   : 12.50 Min.   :-52.4000
## 1st Qu.: -7.91688 1st Qu.:-13.554 1st Qu.: 63.50 1st
Qu.:-21.3500
## Median : -0.06493 Median : 0.671 Median : 92.50 Median :
-0.8000
## Mean : -0.33237 Mean : 1.521 Mean : 80.89 Mean : -0.8427
## 3rd Qu.: 6.95053 3rd Qu.: 18.014 3rd Qu.:102.50 3rd Qu.:
19.6500
## Max.   : 29.48414 Max.   : 49.889 Max.   :133.00 Max.   : 55.7000
## velocity
## Min.   :1409
## 1st Qu.:1523
## Median :1586
## Mean :1594
## 3rd Qu.:1669
## Max.   :1775
```

`velocity` (in km/second) is the response variable, and the locations along `east.west` and `north.south` coordinates are the main explanatory variables. (North and west are positive – if you lie on your back with your head pointing north, then west is to your right.) First, we simply look where the measurements were taken:

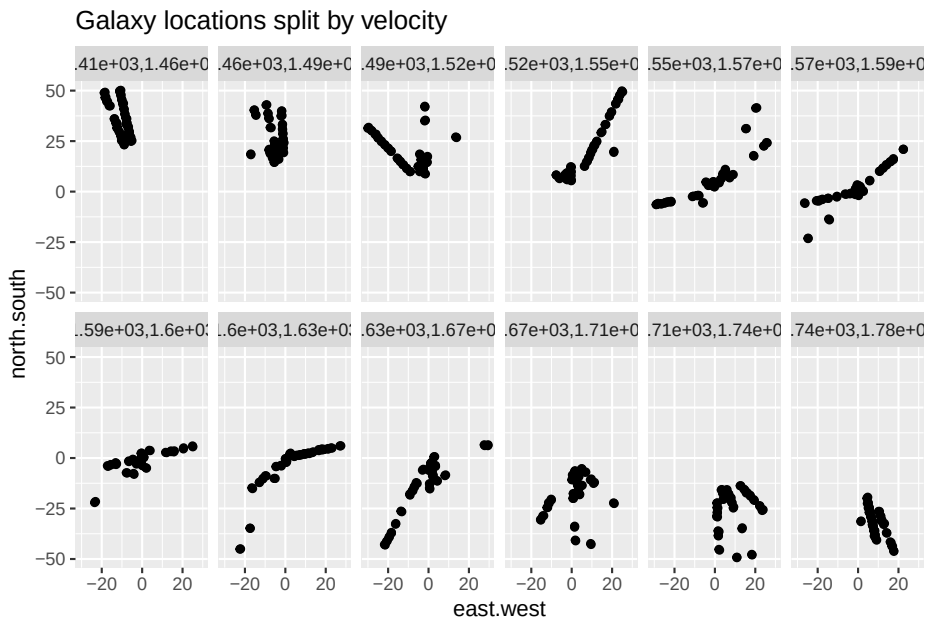
```r
ggplot(galaxy, aes(x=east.west, y=north.south)) + geom_point() + coord_fixed()
```



We see the measurements were taken along seven different lines ("slits".) The variables `radial.position` and `angle` give the locations in polar coordinates.

How does velocity vary with location? We first facet by the `velocity` variable.

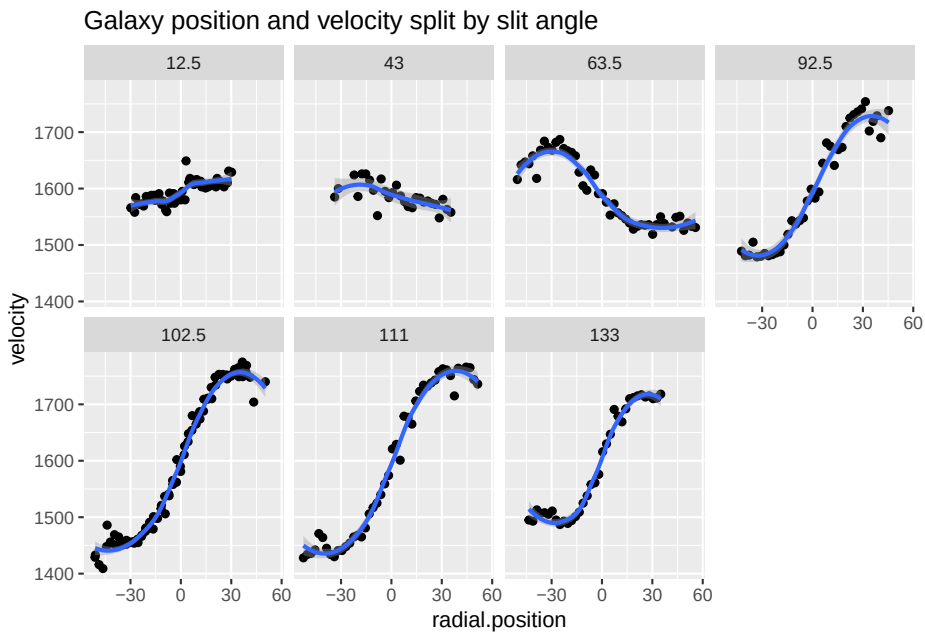
```
ggplot(galaxy, aes(x=east.west, y=north.south)) + geom_point() + geom_jitter(width=0.5
  facet_wrap(~cut_number(velocity, n=12), ncol=6) + coord_fixed() +
  labs(title="Galaxy locations split by velocity")
```



The slowest locations are in the top left panel – these are all in the northeast. The fastest locations in the bottom right panel – these are in the southwest.

We can also look at the velocities as a function of `radial.position`, faceted by `angle`.

```
ggplot(galaxy, aes(x=radial.position, y=velocity)) +
  geom_point() + geom_smooth() + facet_wrap(~angle, ncol=4) +
  labs(title="Galaxy position and velocity split by slit angle")
```



For each angle, we see a nonmonotonic relationship. The magnitude of the variation in velocities varies a lot – it's big at 92.5 to 111 degrees (NNE to SSW), and small for 12.5 and 43 degrees (WNW to ESE.)

5.4.2 Modeling galaxy velocities

We fit a full loess model with interaction. We add the argument `normalize=FALSE`, which is appropriate when the two explanatory variables are on the same scale and we don't want to standardize them. By trial and error, a smoothing parameter `span = 0.25` seems about right.

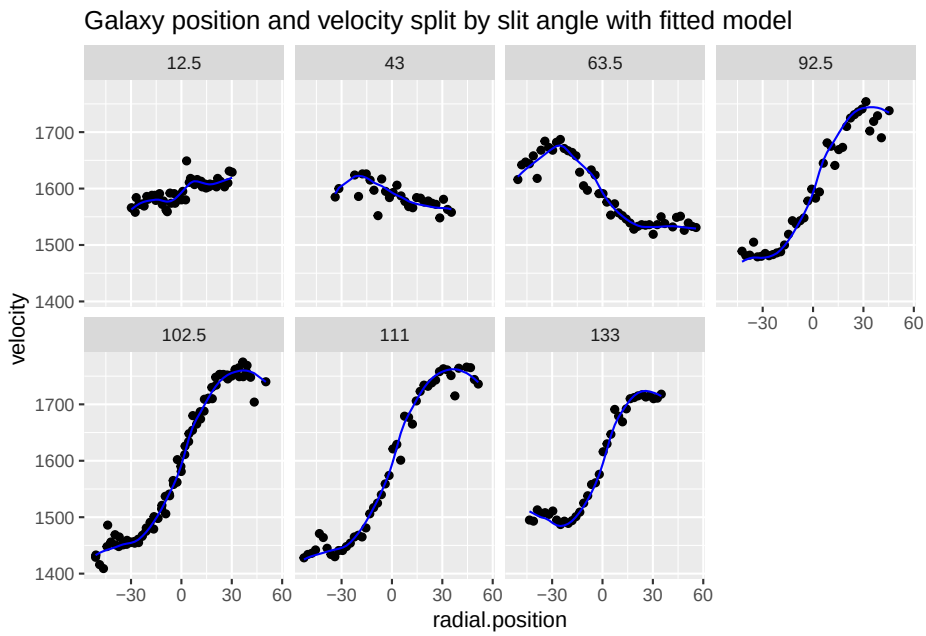
```
galaxy.lo = loess(velocity ~ east.west * north.south, data = galaxy, span = 0.25,
  family="symmetric", normalize=FALSE)
```

We create a data frame with the original variables as well as the fitted values and residuals:

```
galaxy.lo.df = data.frame(galaxy,
  .fitted = fitted.values(galaxy.lo),
  .resid = residuals(galaxy.lo))
```

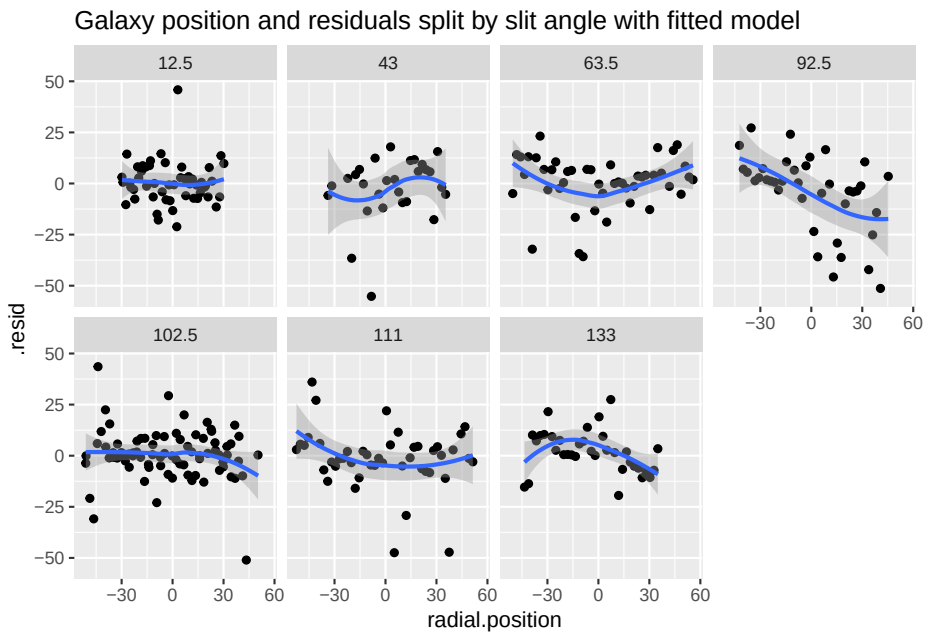
Now redraw the above plot faceted by angle with the fit added.

```
ggplot(galaxy.lo.df, aes(x=radial.position, y=velocity)) + geom_point() +
  geom_line(aes(x=radial.position, y=.fitted), color="blue") +
  facet_wrap(~angle, ncol=4) +
  labs(title="Galaxy position and velocity split by slit angle with fitted model")
```



The model provides the right general shape to the data in polar coordinates.
We take a closer look at the residuals:

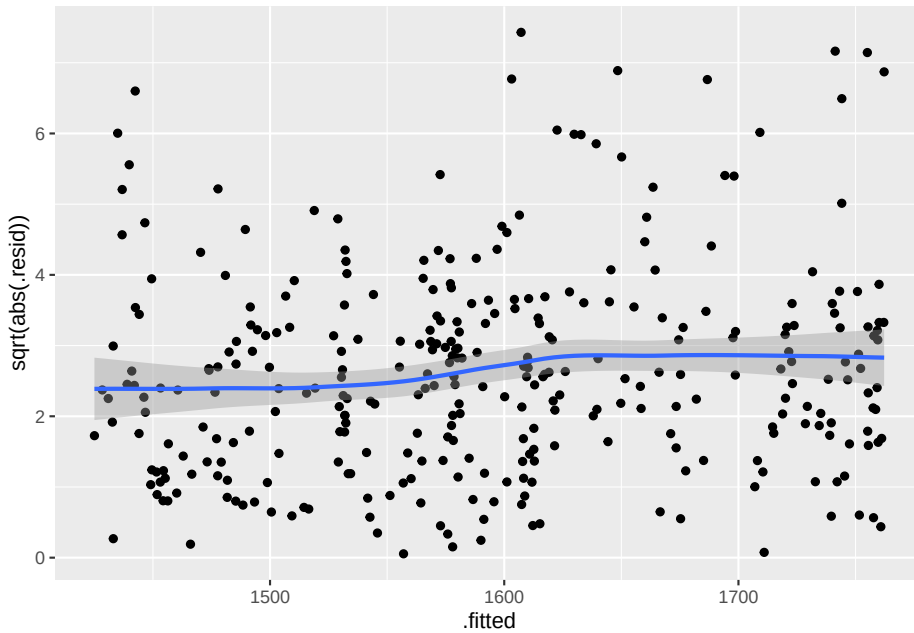
```
gg = ggplot(galaxy.lo.df, aes(x=radial.position, y=.resid)) + geom_point() +
  geom_smooth(span=1) + facet_wrap(~angle, ncol=4)
gg + labs(title="Galaxy position and residuals split by slit angle with fitted model")
```



The 92.5 degrees plot is somewhat bad – it slopes down and it appears heteroskedastic. There's a bit more slope and/or curvature in the others than we would expect under random error as well. Nevertheless, we'll keep our model for simplicity.

We now do our usual residual diagnostics. The spread-location plot of transformed residuals:

```
ggplot(galaxy.lo.df, aes(x=.fitted, y=sqrt(abs(.resid)))) + geom_point() +
  geom_smooth(method.args=list(degree=1))
```



The magnitude of the residuals may increase slightly with fitted value, but it's not so bad that it would demand a transformation. It is so bad that we can't do probabilistic stuff, so we omit the normal QQ plot.

Finally, how much of the variance does our model capture?

```
var(galaxy.lo.df$.fitted) / var(galaxy$velocity)
```

```
## [1] 1.043778
```

Our model captures 104% of the of the variance of the galaxy velocities. Wait, what?

With complex models, the variance of the fitted values can sometimes exceed the variance of the response. If this bothers you, an alternative way of defining a (pseudo) R^2 is

$$\frac{\text{variance of fitted values}}{\text{variance of fitted values} + \text{variance of residuals}}$$

Under this definition, our R^2 becomes

```
var(galaxy.lo.df$.fitted) / (var(galaxy.lo.df$.fitted) + var(galaxy.lo.df$.resid))
```

```
## [1] 0.980679
```

Some authors prefer using the squared correlation between the fitted values and the response:

```
cor(galaxy.lo.df$.fitted, galaxy$velocity)^2
```

```
## [1] 0.9804274
```

The difference is academic, though. Our model captures the vast majority of the variation in the galaxy velocities.

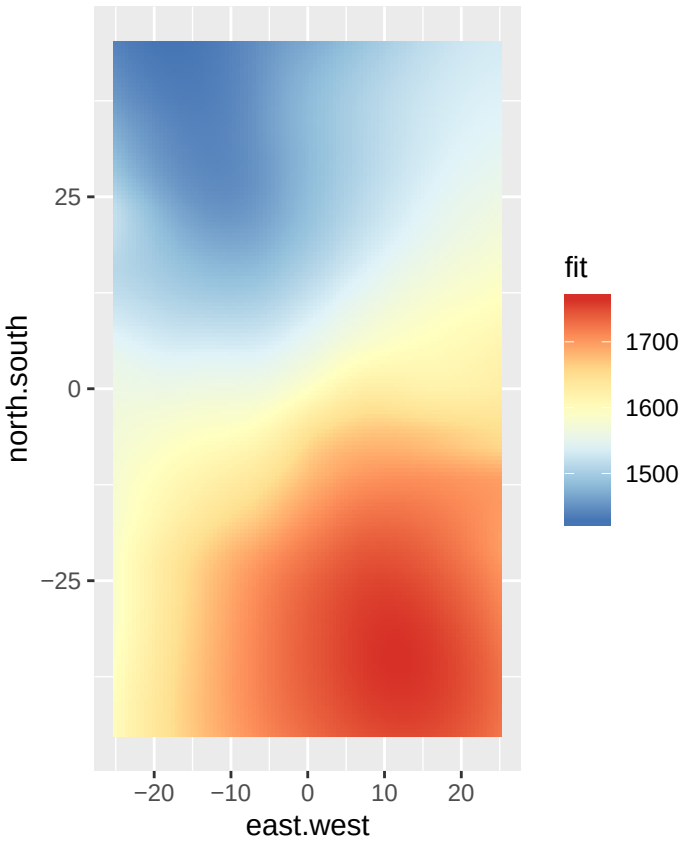
5.4.3 Visualizing the fit: geom_raster and geom_contour

To prepare for plotting, let's make predictions on a rectangular grid, and put the results in a data frame, `galaxy.plot.df`.

```
galaxy.grid = expand.grid(east.west=seq(-25,25,0.5), north.south=seq(-45,45,0.5))
galaxy.predict = predict(galaxy.lo, newdata=galaxy.grid)
galaxy.plot.df = data.frame(galaxy.grid, fit=as.vector(galaxy.predict))
```

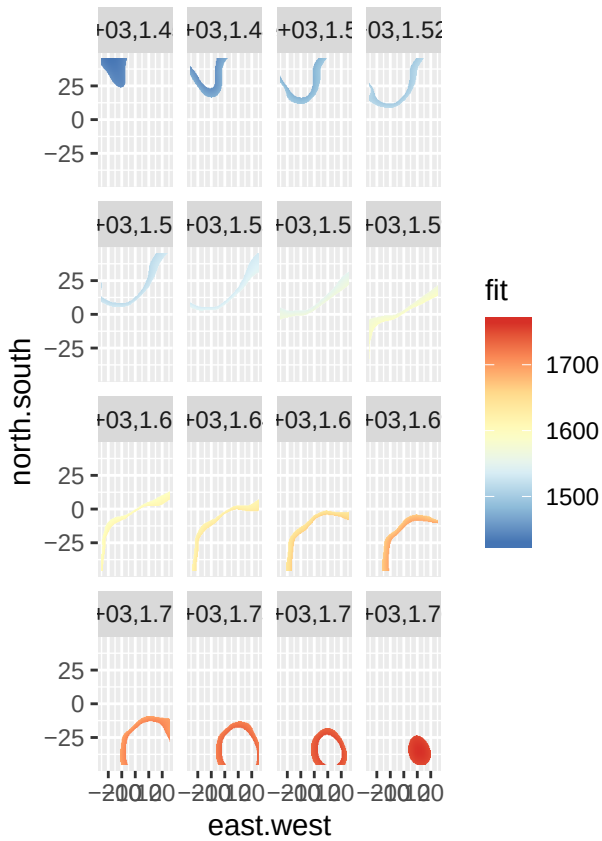
Because of the interaction in the loess, the techinques we've used previously that condition on explanatories aren't going to be sufficient. One option is to use `geom_raster()` to indicate the third variable using color. (`geom_tile()` can be used instead for non-rectangular data.)

```
ggplot(galaxy.plot.df, aes(x=east.west, y=north.south, fill=fit)) + geom_raster() + co
  scale_fill_distiller(palette="RdYlBu")
```



It's often quite hard to find a color palette that clearly displays all the features you want to make apparently. One solution is to facet on the fitted value. We cut the above plot into a large number of pieces by fitted value, then draw a grid of plots.

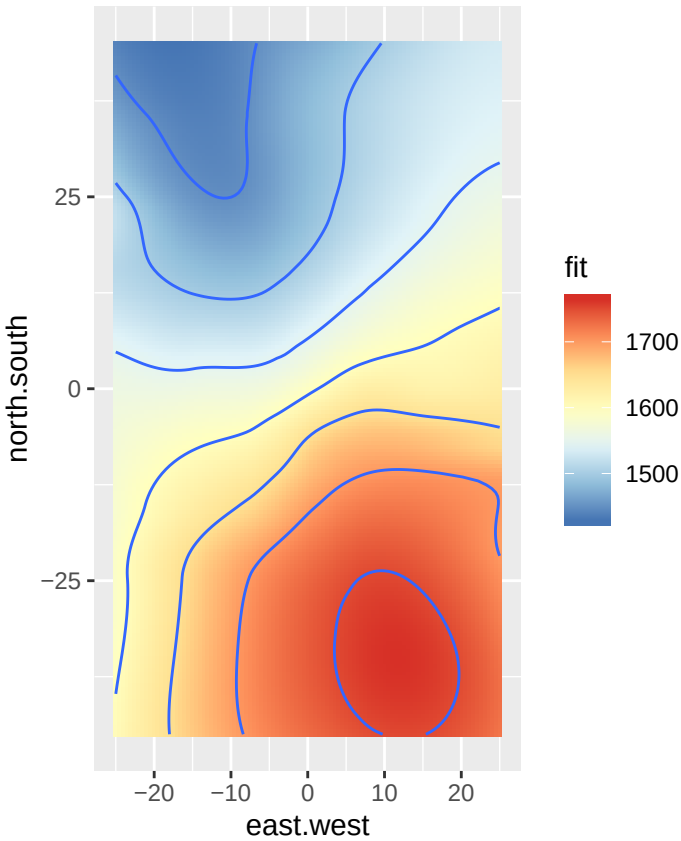
```
ggplot(galaxy.plot.df, aes(x=east.west, y=north.south, fill=fit)) + geom_raster() + coord_fixed()
  scale_fill_distiller(palette="RdYlBu") + facet_wrap(~cut_number(fit, n=16), ncol=4)
```



The basic pattern is clear: the fitted velocities are small in the northeast and generally increase as you move toward the southwest, though the details are complicated.

Alternatively, we can make the shapes clear on the original graph by adding **contours**. A contour is a curve that joins together points that have the same value of the z-variable. `geom_contour()` gives the basic contour plot:

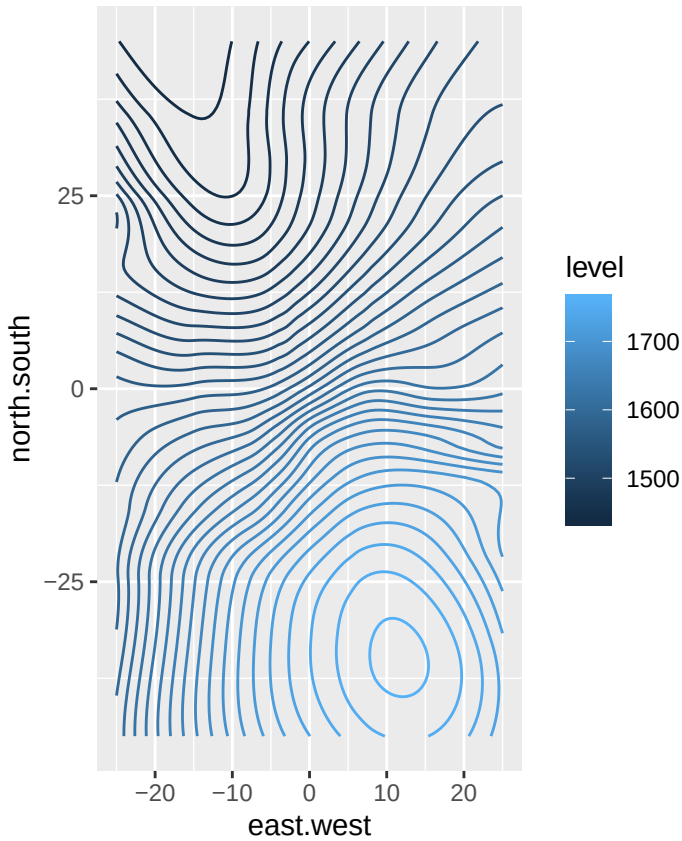
```
ggplot(galaxy.plot.df, aes(x=east.west, y=north.south, z=fit)) + geom_raster(aes(fill =
  coord_fixed() + scale_fill_distiller(palette="RdYlBu") + geom_contour()
```



We can see, for example, a circle of points in the southwest all have the same fitted value. The coloring makes it clear this is a high value (velocity over 1700 km/second.) The northeast has the lowest fitted values.

A final choice is to skip the raster part and go straight to the contours.

```
ggplot(data.frame(galaxy.grid, fit=as.vector(galaxy.predict)),
  aes(x=east.west, y=north.south, z=fit)) +
  geom_contour(binwidth=10, aes(color=..level..)) + coord_fixed()
```

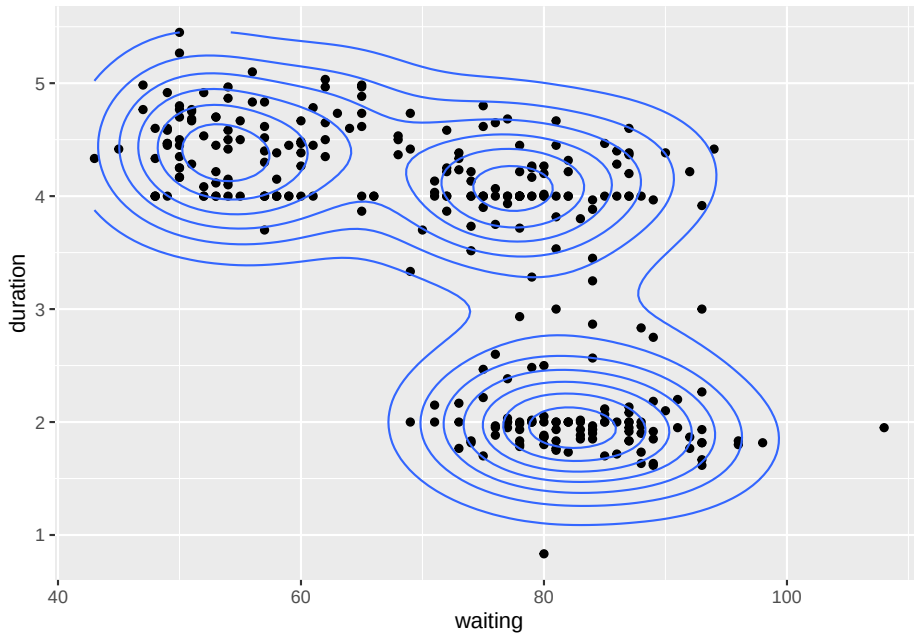


If relying on contours alone, they need to be fairly dense to get the details of the distribution across. Note that labeling the contours by putting numbers directly on the graph is very hard to do within `ggplot` (if you must, use `contourplot` in the `lattice` library instead.)

5.4.4 Contour plots for bivariate densities

The other major use for contour plots is to display density estimates for bivariate data. The ggplot function `stat_density_2d` does bivariate density estimation. We apply it to the data set `geyser`, which contains bivariate data on the waiting time until eruption and the duration of the subsequent eruption (both in minutes) for 299 successive eruptions of Old Faithful in 1985:

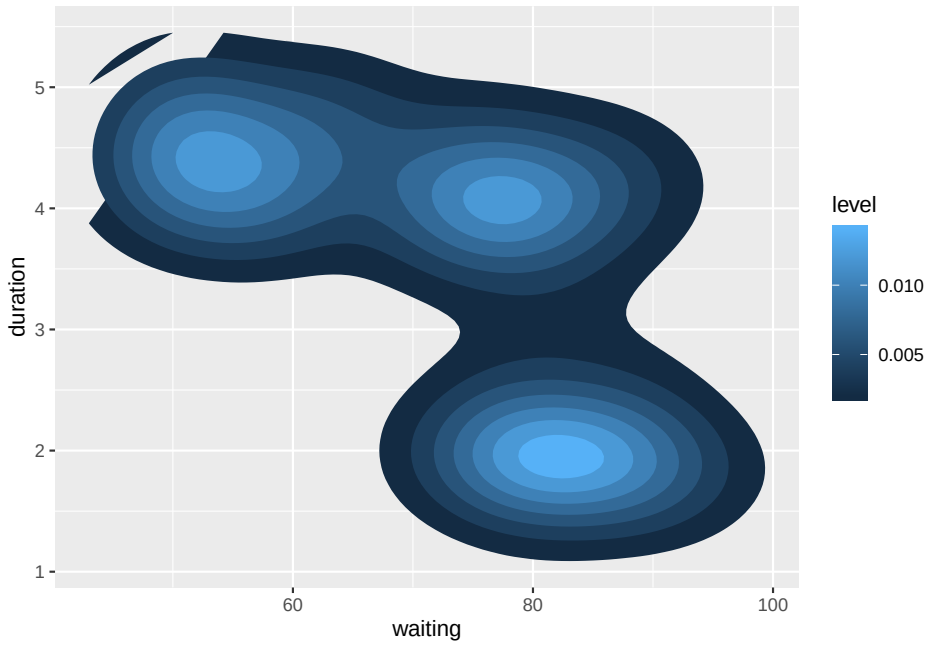
```
library(MASS)
ggplot(geyser, aes(x=waiting, y=duration)) + geom_point() + stat_density_2d()
```



The data appears trimodal: we get short wait/long eruption, long wait/long eruption, and long wait/short eruption. We don't get short wait/short eruption.

We could also look at this by coloring in between the contours to get a solid surface:

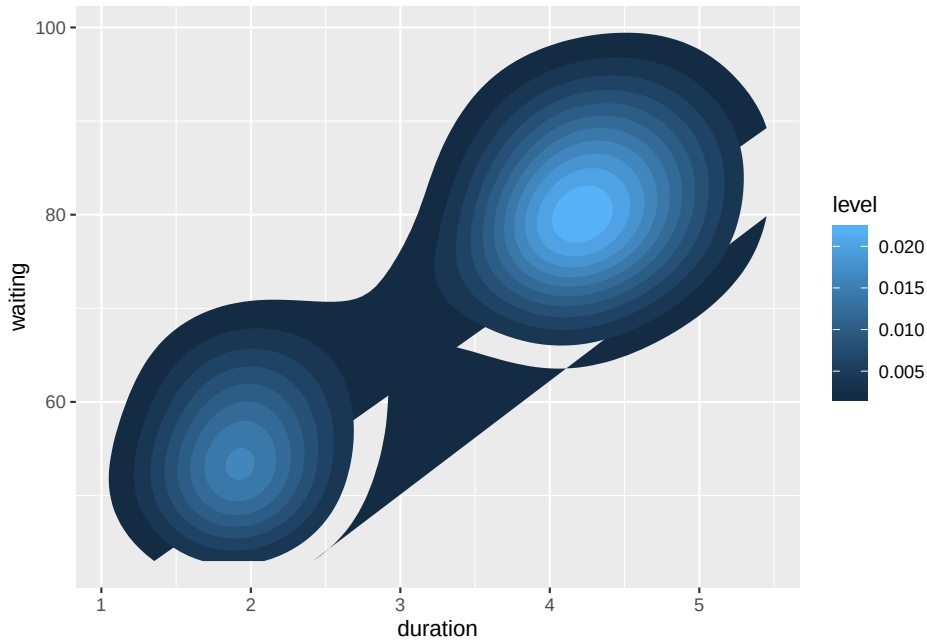
```
ggplot(geyser, aes(x=waiting, y=duration)) + stat_density_2d(aes(fill=..level..), geom="polygon")
```



The general message is the same, but now we have quantitative information on the density. The highest peak is for long wait/long eruption, at around 0.015 (per minute per minute.) You can explore the 2D density further by studying things like the conditional distribution of eruption length given waiting time, etc.

If you're sitting around at Yellowstone, you might be more interested in the joint density of the eruption duration and the waiting time to the *next* eruption.

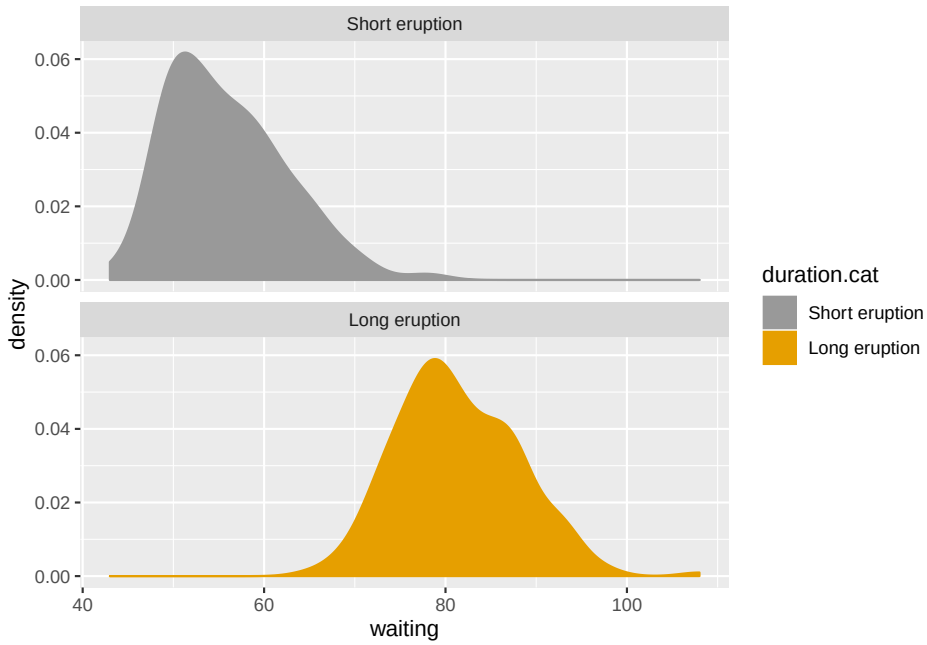
```
n = nrow(geyser)
next.geyser = data.frame(duration = geyser$duration[-n], waiting = geyser$waiting[-1])
ggplot(next.geyser, aes(x=duration, y=waiting)) + stat_density_2d(aes(fill=..level..),
```



This time the data looks bimodal: short eruption followed by short wait, or long eruption followed by long wait. The long/long peak is substantially higher.

Finally, we want to study the **conditional** distributions: given an eruption duration, how long will we have to wait for the next eruption? For simplicity, we only split `duration` into two categories.

```
duration.cat = rep(NA, nrow(next.geyser))
duration.cat[next.geyser$duration<=3] = "Short eruption"
duration.cat[next.geyser$duration>3] = "Long eruption"
duration.cat = factor(duration.cat, levels=c("Short eruption", "Long eruption"))
ggplot(data.frame(next.geyser, duration.cat), aes(x=waiting, fill=duration.cat, color=duration.ca
  stat_density() + facet_wrap(~duration.cat, ncol=1) +
  scale_color_manual(values = cb_palette) + scale_fill_manual(values = cb_palette)
```



The modal waiting time after a short eruption is a bit over 50 minutes, while the modal waiting time after a long eruption is a bit under 80 minutes. Note that this graph does not attempt to show that long eruptions are more common than short eruptions.

5.5 Truly 3D plots: Wireframes

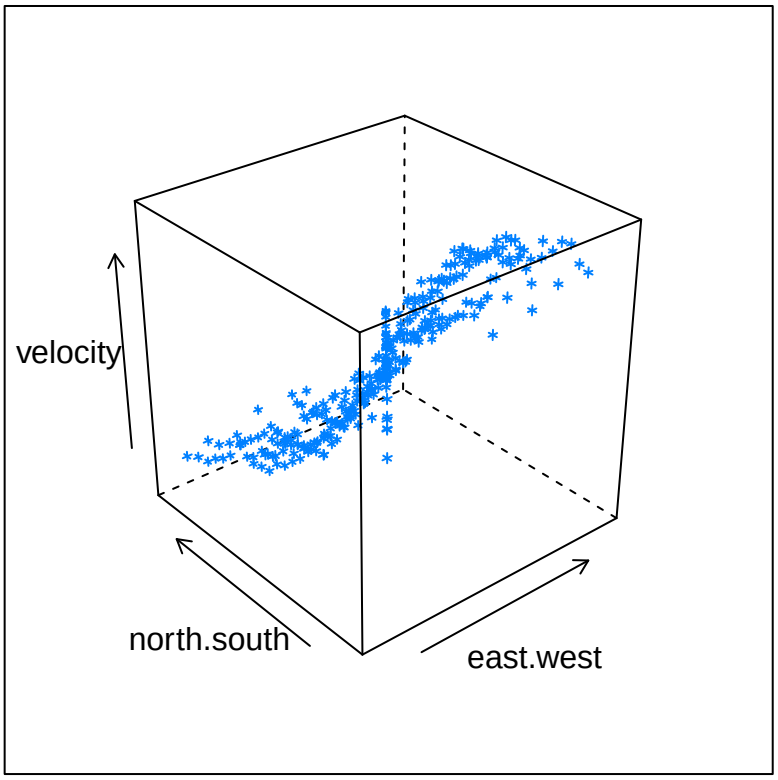
READ: Cleveland pp. 249–267.

It's the lecturer's opinion that 3D plots are kind of overrated, but some people like them so here they are. `ggplot()` doesn't really do 3D plots, so we go to the `lattice` library for this section.

5.5.1 Galaxy data

Let's return to the loess fit to the galaxy data from last time. Recall that north and west are positive; putting north on top, the slow parts of the galaxy were in the top left (northeast) and the fast parts of the galaxy were in the bottom right (southwest.) Firstly, we can use the `cloud()` function to plot the raw data in three dimensions.

```
load("lattice.RData")
cloud(velocity ~ east.west * north.south, data=galaxy)
```



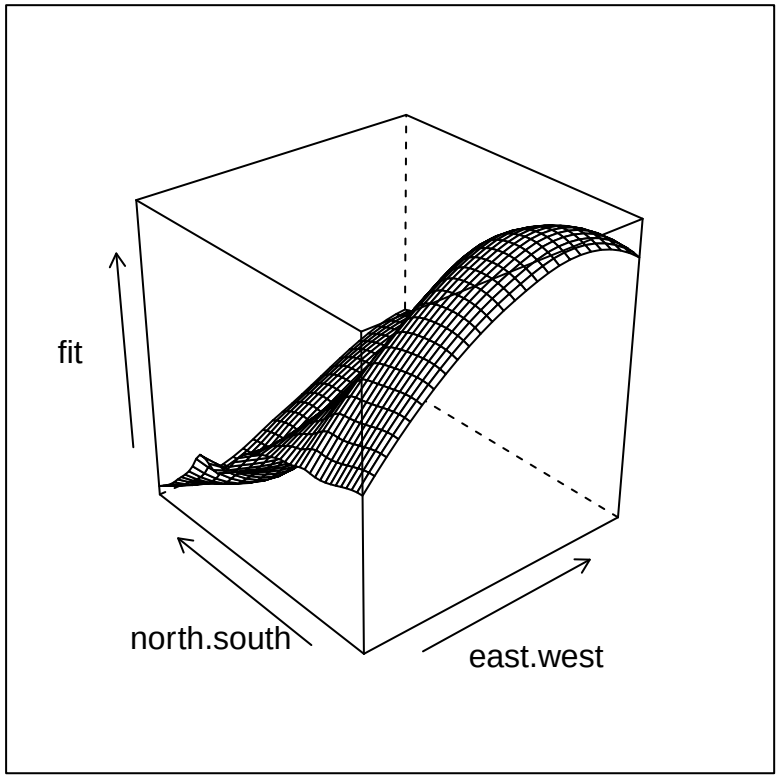
The arrows point toward more positive values of the variables: north, west, and high velocity respectively. (If you wanted numeric scales on the axes instead of just arrows, you could use `scatterplot3d()` in the library of the same name, but I find it near-impossible to accurately read numbers off 3D plots so I don't bother including them.) Now that we know what to look for, we see the data is consistent with low velocities in the northeast and high velocities in the southwest. This would be possible but somewhat hard to see if we didn't know what we were looking for.

Now re-fit the loess model we chose last time, and make predictions on a grid.

```
galaxy.lo = loess(velocity ~ east.west * north.south, data = galaxy, span = 0.25,
  family="symmetric", normalize=FALSE)
galaxy.wf.grid = expand.grid(east.west=seq(-25,25,2), north.south=seq(-45,45,2))
galaxy.wf.predict = predict(galaxy.lo, newdata=galaxy.wf.grid)
galaxy.wf.df = data.frame(galaxy.wf.grid, fit=as.vector(galaxy.wf.predict))
```

To draw a truly 3D plot, we use `wireframe()`.

```
wireframe(fit ~ east.west * north.south, data=galaxy.wf.df)
```



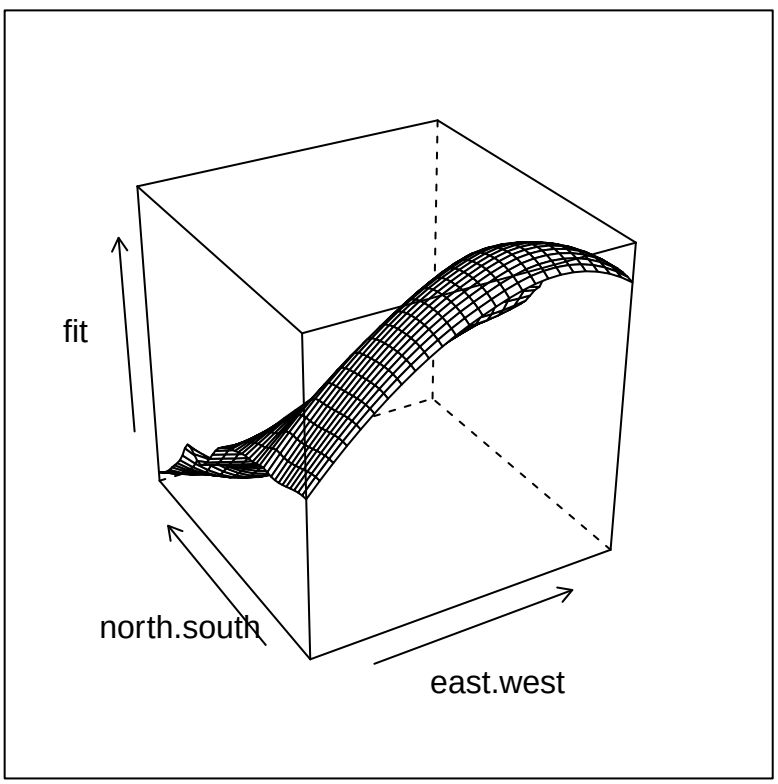
This is quite a bit clearer than the `cloud()` plot. In particular, the curvature of the fitted surface is apparent. The estimated velocity goes down a bit in the extreme southwest corner.

With a wireframe plot, it's rare that one angle lets you see all relevant details of the fit. If you're just exploring the data on your own, the `rotate.wireframe()` function in the `TeachingDemos` library can be fun to play with (if a bit buggy.)

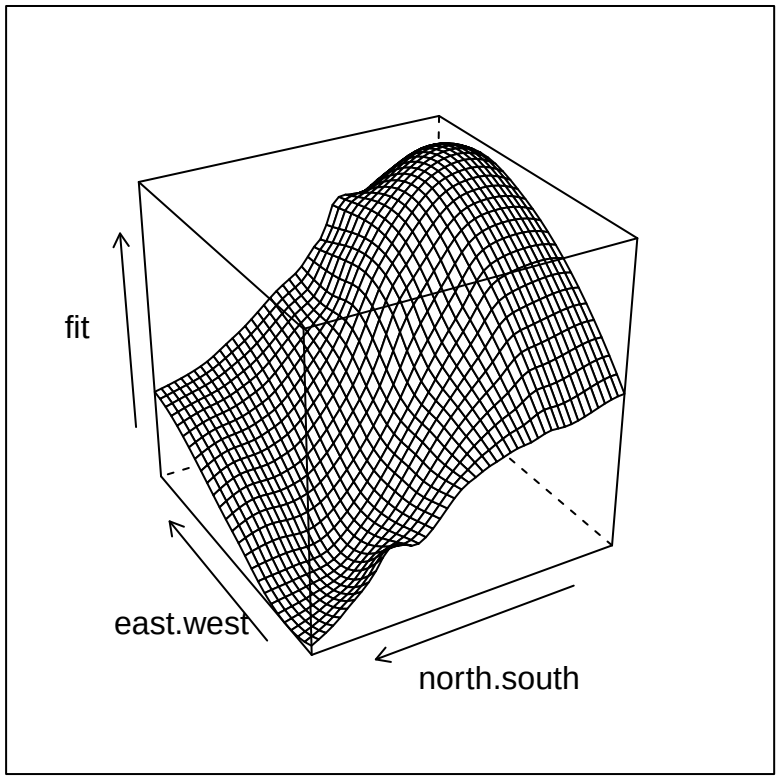
```
# install.packages("TeachingDemos")
library(TeachingDemos)
rotate.wireframe(fit ~ east.west * north.south, data=galaxy.wf.df)
```

Of course, this doesn't fly if you're trying to prepare a document. You could build a Shiny app but that probably isn't going to be worth the effort. Instead, pick a few different angles, and show the wireframe from those angles.

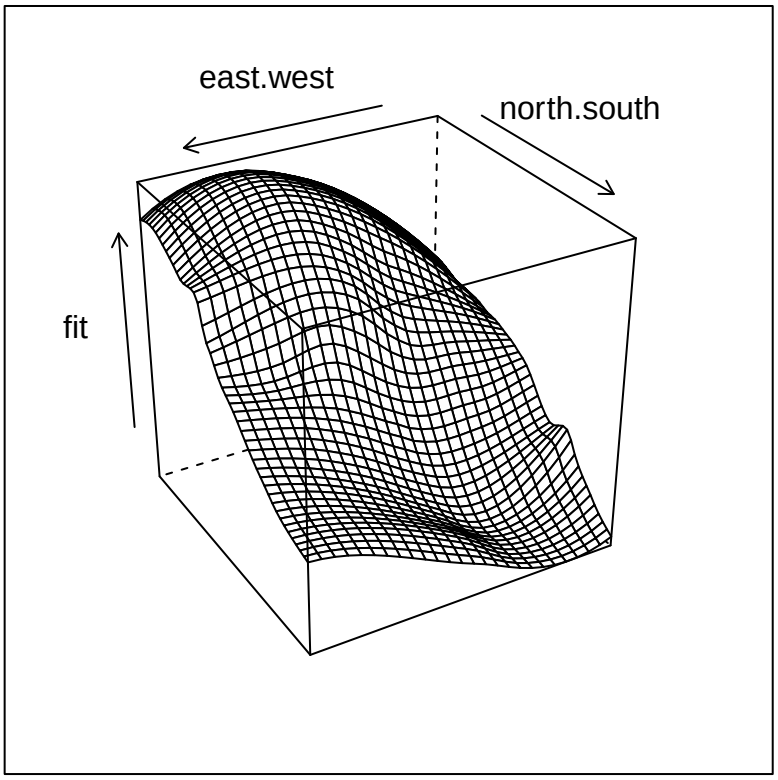
```
wireframe(fit ~ east.west * north.south, data=galaxy.wf.df, screen = list(z=30, x=-60, y=0))
```



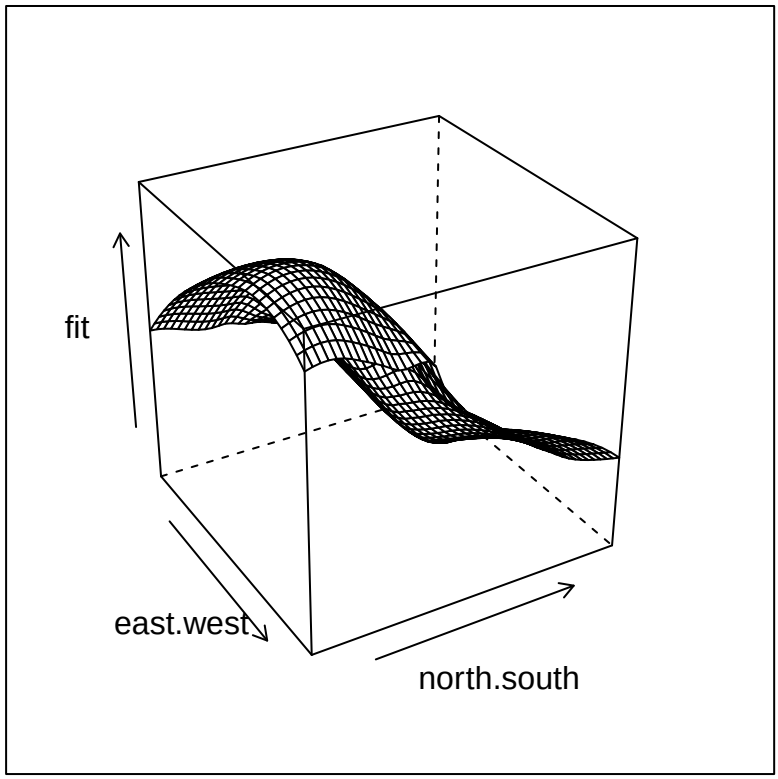

```
wireframe(fit ~ east.west * north.south, data=galaxy.wf.df, screen = list(z=120, x=-60, y=0))
```




```
wireframe(fit ~ east.west * north.south, data=galaxy.wf.df, screen = list(z=210, x=-60
```



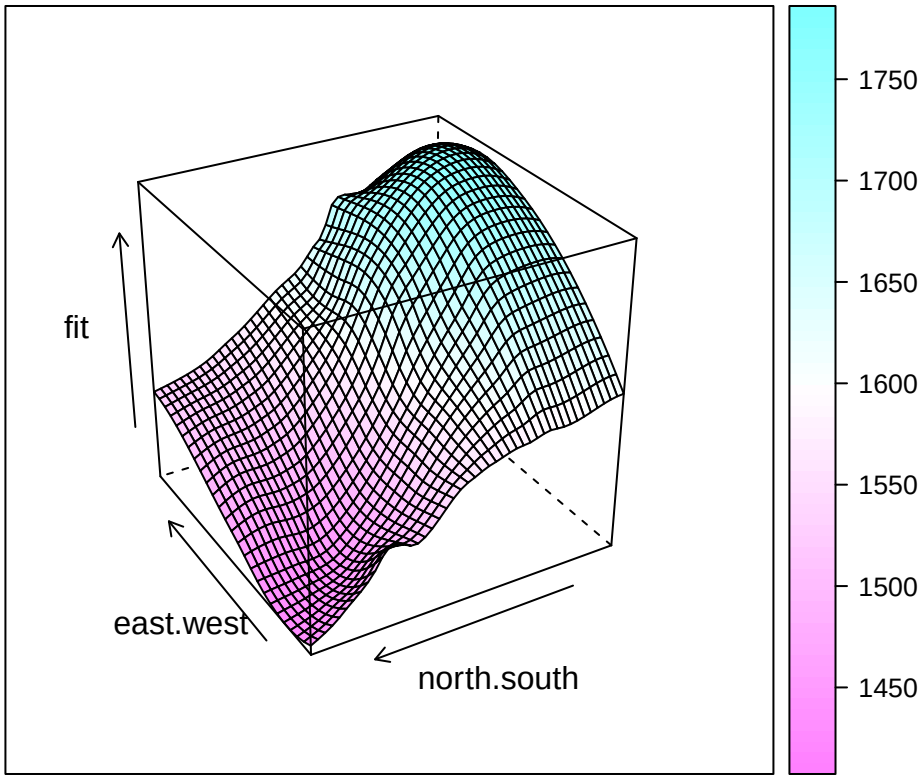

```
wireframe(fit ~ east.west * north.south, data=galaxy.wf.df, screen = list(z=300, x=-60, y=0))
```



Here we kept the x and y angles fixed, while rotating z by 90 degrees each time. The effect is to "spin" the surface around while keeping the "camera" fixed.

Finally, we can "fill in" in the wireframe using the `drape` argument.

```
wireframe(fit ~ east.west * north.south, data = galaxy.wf.df,
          screen = list(z=120, x=-60, y=0), drape=TRUE)
```

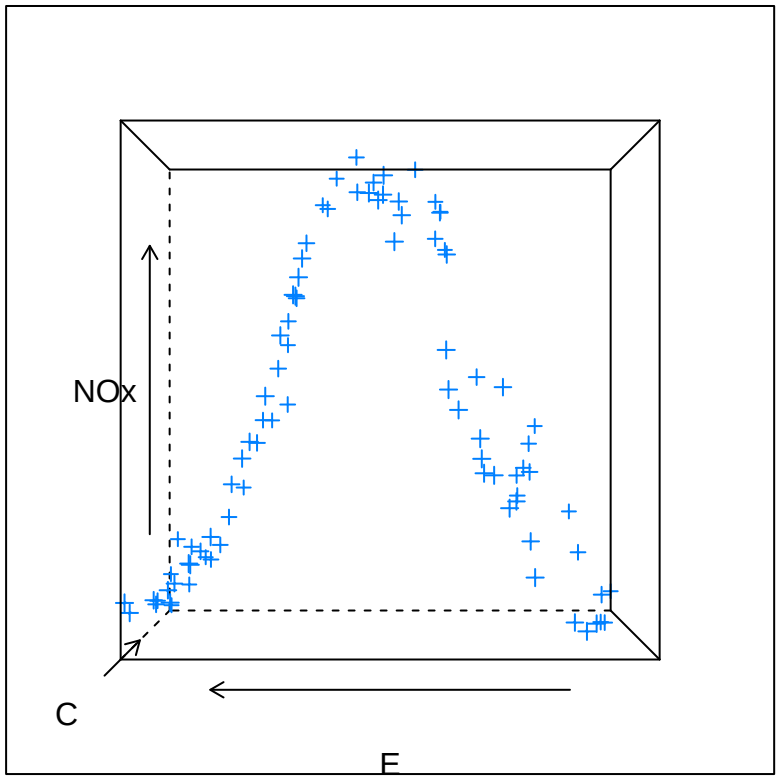


You can play around with the color scheme using the `col.regions` argument. All this is getting dangerously close to chartjunk territory, though.

5.5.2 Ethanol data

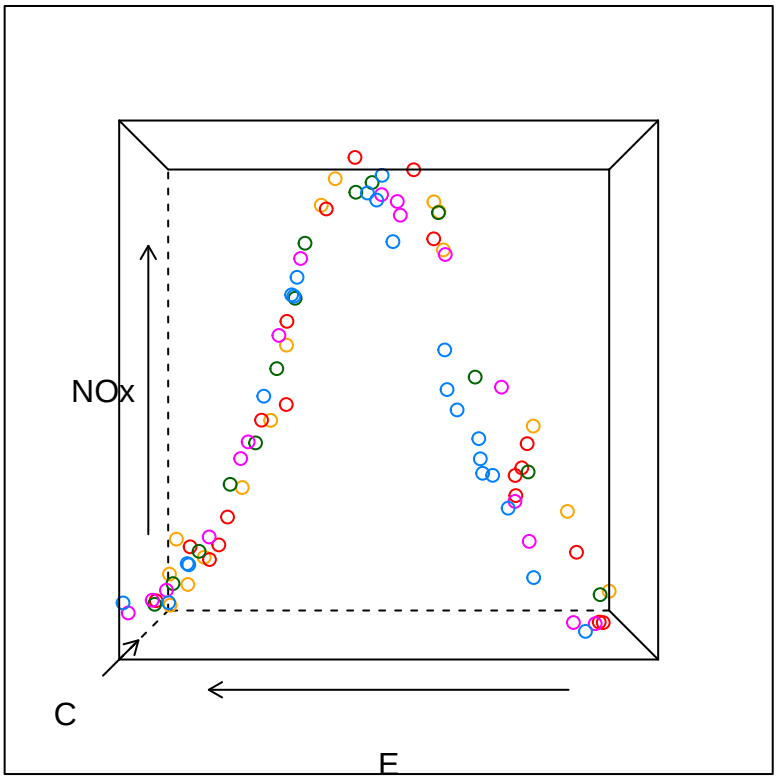
We now return to the ethanol engine data. Recall that the curves relating NOX to equivalence ratio (E) were close in shape but not identical for all five values of compression ratio (C). If we cloud plot the data and look at it "front-on," i.e. with C going into the screen, we get something that's almost a 2D scatterplot of NOx against E.

```
cloud(NOx ~ C * E, data=ethanol, screen = list(z=90, x=-90, y=0))
```



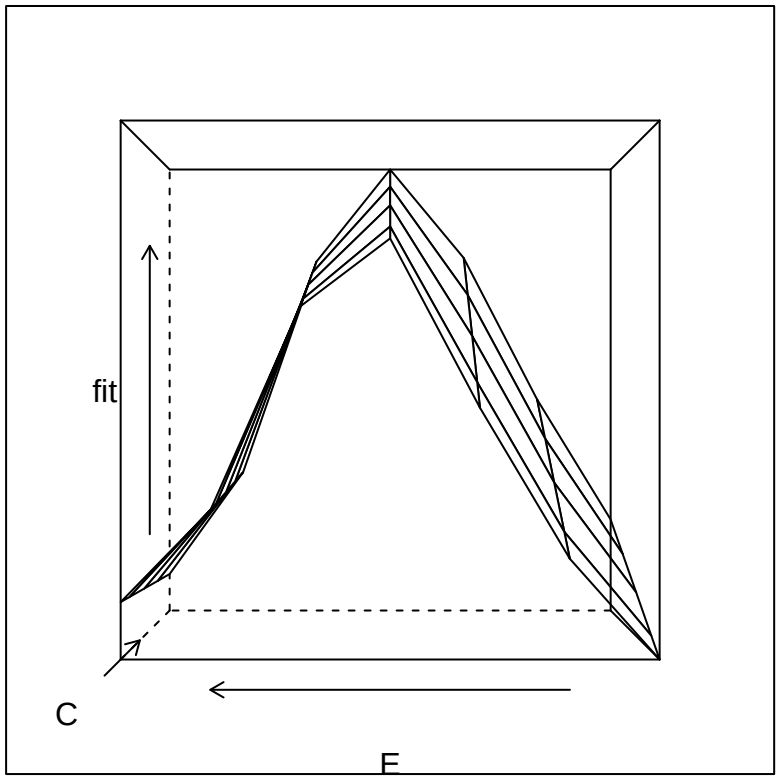
We can try coloring the plot by levels of C:

```
cloud(NOx ~ C * E, data=ethanol, screen = list(z=90, x=-90, y=0), groups=C)
```



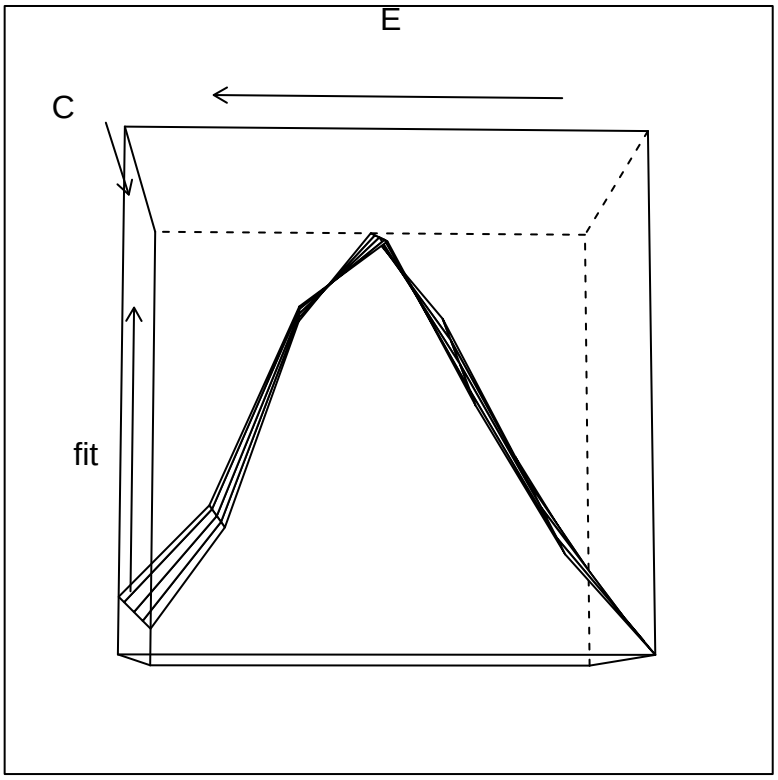
Now we can see the blue points (for example) are lower than the others on the right hand side, while they're similar to the other colors on the left. (Again, this is easier to see in retrospect after having studied the data.) We could try to make this plot look nicer by fiddling with the color scheme or adding a legend, but we'll be better off instead looking at the loess surface we fitted to the data. Re-fit the model and look at it head-on:

```
ethanol.lo = loess(NOx ~ C * E, data=ethanol, span=1/3, parametric="C",
                   drop.square="C", family="symmetric")
ethanol.grid = expand.grid(C=c(7.5,9,12,15,18), E=seq(0.6, 1.2, 0.1))
ethanol.predict = predict(ethanol.lo, newdata=ethanol.grid)
ethanol.df = data.frame(ethanol.grid, fit=as.vector(ethanol.predict))
wireframe(fit ~ C * E, data=ethanol.df, screen=list(z=90, x=-90, y=0))
```



We see that the height of the surface (the fitted value) gets taller as C increases. Note, however, that changing the angles by just a few degrees make this seem to disappear.

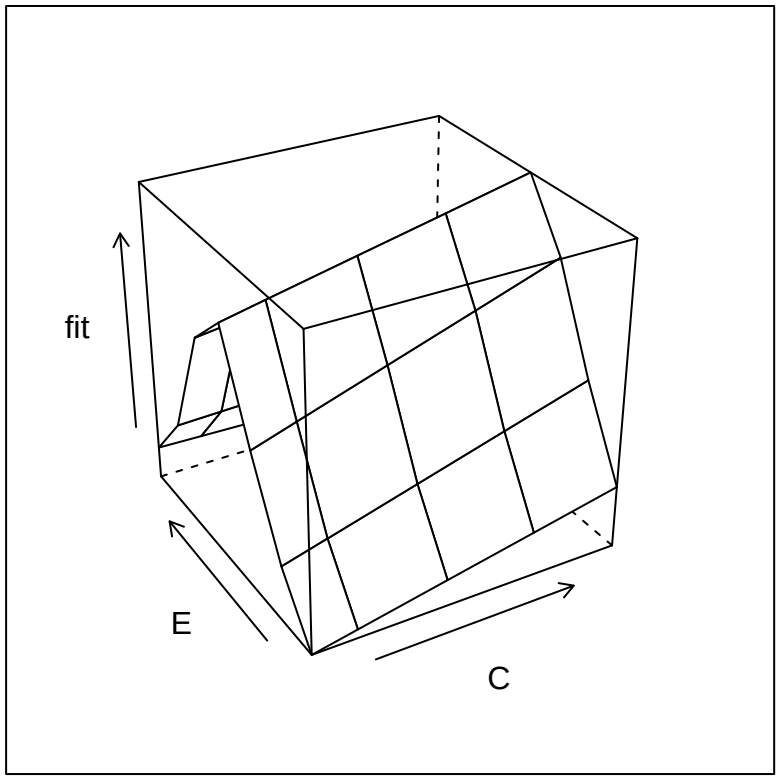
```
wireframe(fit ~ C * E, data=ethanol.df, screen=list(z=92, x=-97, y=0))
```



The issue is *foreshortening* – things that are further away look smaller. To better understand foreshortening, look at a bunch of Italian Renaissance paintings. For our purposes, it's enough to remember that the choice of angle is important, so make sure to try out a few.

For this fit, looking somewhat from the side makes the change in height clearer:

```
wireframe(fit ~ C * E, data=ethanol.df, screen=list(z=30, x=-60, y=0))
```

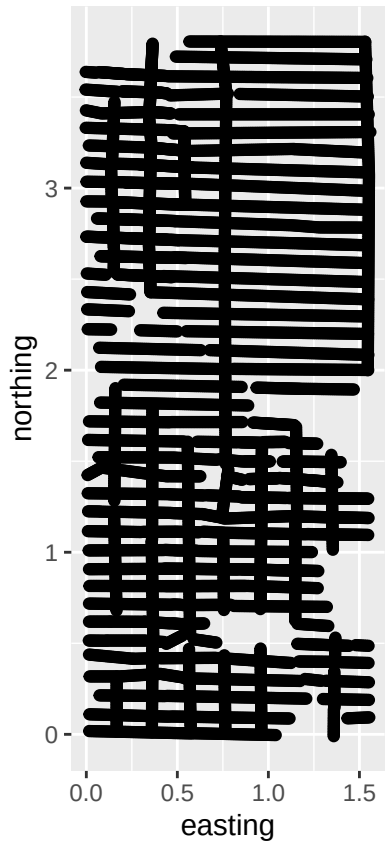


Now it's obvious that higher C generally means somewhat higher NOx. Again, you need more than one angle to see all of what's going on.

5.5.3 Soil

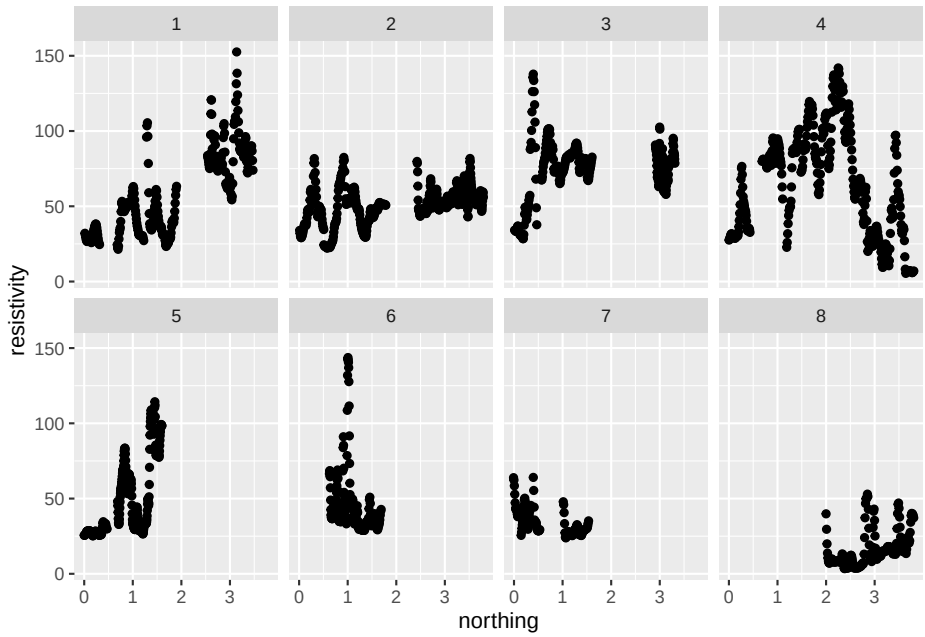
The Cleveland data set `soil` contains measurements on resistivity (in ohm cm) in a field in Western Australia. The locations are given by "easting" and "northing" coordinates, which just measure distance from an origin in kilometers.

```
ggplot(soil, aes(x=easting, y=northing)) + geom_point() + coord_fixed()
```



We see the locations occur along a number of "tracks", which are recorded in the `track` variable. Some tracks are north-south while others are east-west; this is recorded in the variable `is.ns`. To see how the resistivity varies by coordinate, we'll first subset by track direction, then facet by track.

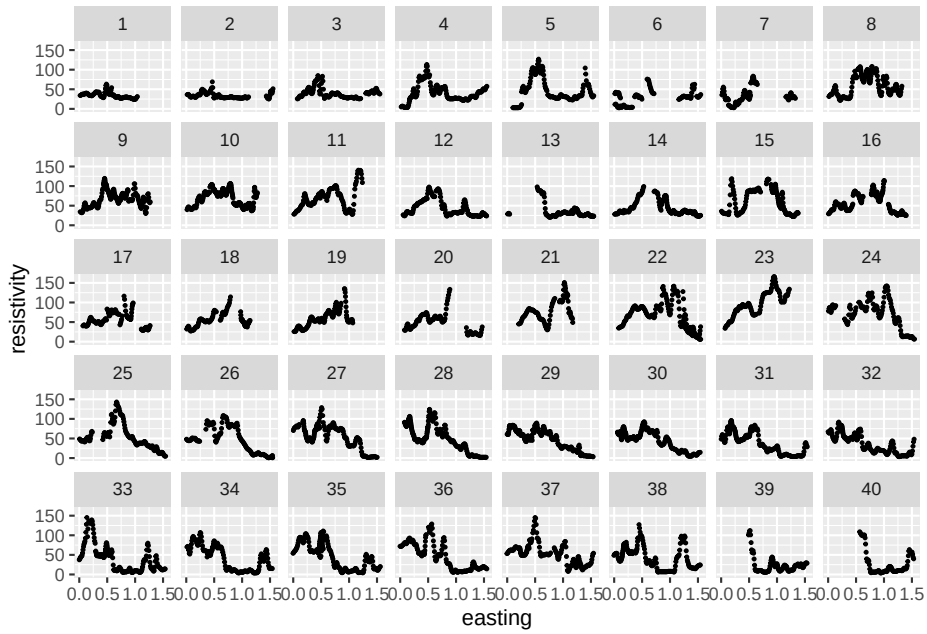
```
ggplot(subset(soil, is.ns == TRUE), aes(x=northing, y=resistivity)) + geom_point() +
  facet_wrap(~track, ncol=4)
```



Looking at the north-south tracks, the patterns aren't very consistent. There are a bunch of spikes occurring at seemingly random locations.

Now try the east-west tracks:

```
ggplot(subset(soil, is.ns == FALSE), aes(x=easting, y=resistivity)) + geom_point(size=0
  facet_wrap(~track, ncol=8)
```



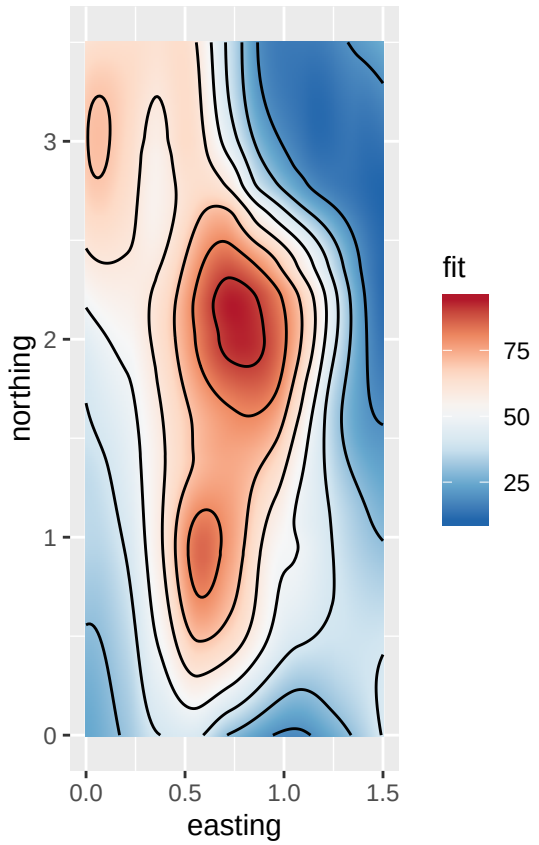
We see that there's usually a downward trend in resistivity as the easting coordinate increases. However, this doesn't show up in all the plots.

We fit a loess model to the data, predicting resistivity from the easting and northing coordinates (with an interaction.)

```
soil.lo = loess(resistivity ~ easting * northing, span = 0.25, data=soil)
```

Now predict on a grid and plot the fit using color and contours.

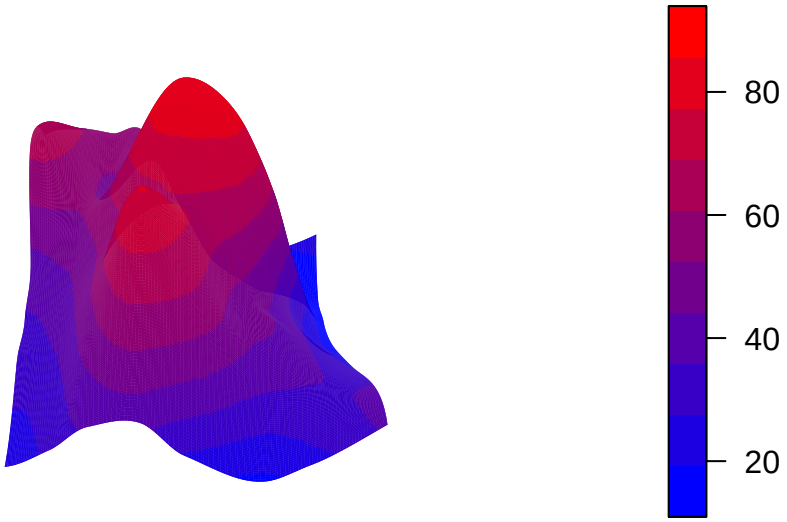
```
soil.grid = expand.grid(easting = seq(0, 1.5, 0.01), northing = seq(0, 3.5, 0.01))
soil.predict = predict(soil.lo, newdata=soil.grid)
soil.df = data.frame(soil.grid, fit=as.vector(soil.predict))
ggplot(soil.df, aes(x=easting, y=northing, z=fit, fill=fit)) + geom_raster() +
  geom_contour(binwidth=10, color="black") +
  scale_fill_distiller(palette="RdBu") + coord_fixed()
```



There's a clear peak around $(0.75, 2.1)$, along with a smaller peak near $(0.6, 0.9)$.

The complexity of the surface means that a wireframe plot isn't well-suited to displaying the fit. You can use `surf3D()` in the `plot3D` library:

```
# install.packages("plot3D")
library(plot3D)
east.grid = seq(0, 1.5, 0.01)
north.grid = seq(0, 3.5, 0.01)
mesh.grid = mesh(east.grid, north.grid)
fit.grid = matrix(soil.predict, nrow=length(east.grid))
surf3D(mesh.grid$x, mesh.grid$y, fit.grid, theta=0, col = ramp.col(col = c("blue", "red
```



It's unclear whether this is of any real data analytic value. It looks cool, I guess!

5.5.4 Things I should add

GAMs/thin-plate splines for local effects?

Chapter 6

Hypervariate data

6.1 Four variables: Modeling ozone

READ: Cleveland pp. 272–292.

Load the usual:

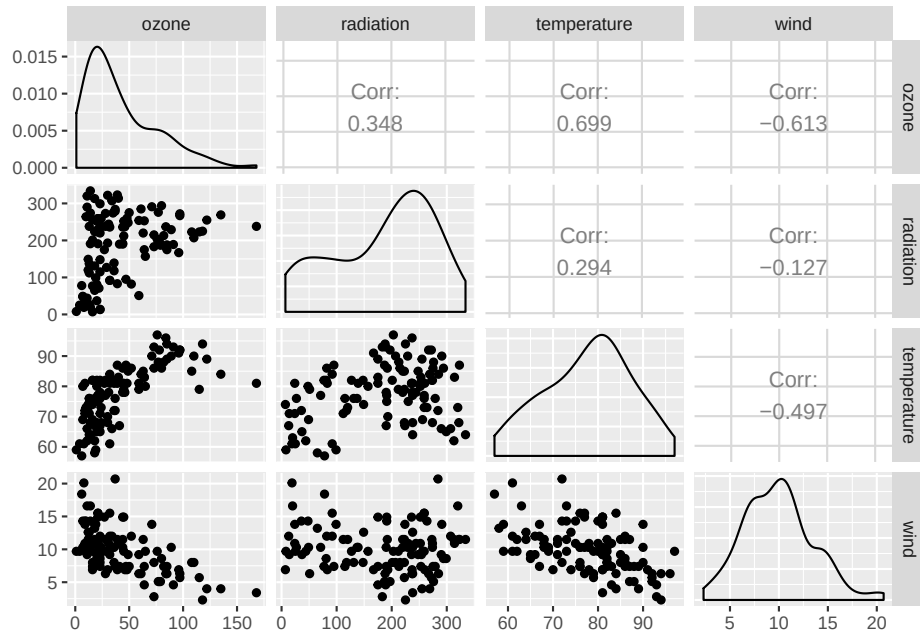
```
load("lattice.RData")
library(tidyverse)
library(broom)
library(GGally)
cb_palette = c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC7
```

The `environmental` data set contains four variables measured in New York on 111 days from May to September 1973:

- `ozone` in parts per billion;
- `radiation` in Langleys;
- `temperature` in degree Fahrenheit;
- `wind` in miles per hour.

We previously looked at the bivariate distribution between temperature and wind speed. Now we'll look at radiation, temperature, and wind as predictors of ozone.

```
ggpairs(environmental)
```




```r
summary(environmental)
```

```
## ozone radiation temperature wind
## Min.  : 1.0 Min.  : 7.0 Min.  :57.00 Min.  : 2.300
## 1st Qu.: 18.0 1st Qu.:113.5 1st Qu.:71.00 1st Qu.: 7.400
## Median : 31.0 Median :207.0 Median :79.00 Median : 9.700
## Mean : 42.1 Mean :184.8 Mean :77.79 Mean : 9.939
## 3rd Qu.: 62.0 3rd Qu.:255.5 3rd Qu.:84.50 3rd Qu.:11.500
## Max.  :168.0 Max.  :334.0 Max.  :97.00 Max.  :20.700
```

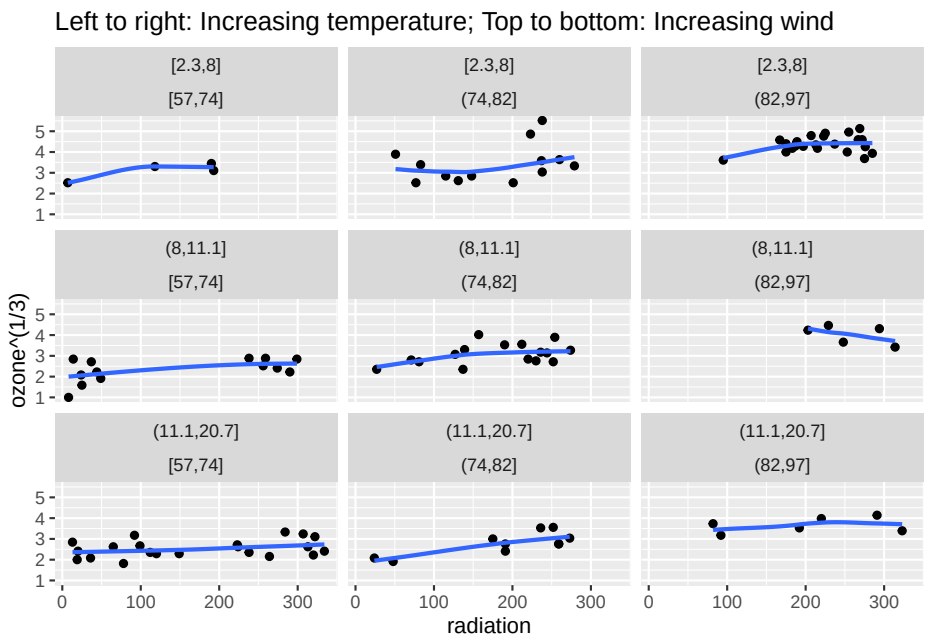
The paired plots showed that ozone is related to all three of the other variables. Since it doesn't look like any of the other variables are strongly collinear, we keep all three predictors. The density plot of ozone shows strong right skew, so some transformation will probably be beneficial. After some experimentation (not shown here), Cleveland's proposal of a cube root transformation seems reasonable, if somewhat hard to interpret.

6.1.1 Faceting the data

To consider all four variables at once, we now have to facet two ways. That is, pick one predictor variable (here, radiation), then cut the other two predictors into categories and create a grid of faceted plots. In the grid below, left to right gives low to high temperature, while top to bottom gives low to high wind.

```r
ggplot(environmental, aes(x = radiation, y = ozone^(1/3))) + geom_point() +
  geom_smooth(span = 1, method.args = list(degree = 1,
    family = "symmetric"), se = FALSE) +
```

```
  facet_wrap(~cut_number(wind, n = 3) + cut_number(temperature, n = 3)) +
  ggtitle("Left to right: Increasing temperature; Top to bottom: Increasing wind")
```



This is getting complicated, so it would be nice to give names to the categories to keep track of what's going on. Create some variables:

```
CubeRootOzone = environmental$ozone^(1/3)
radiation = environmental$radiation
temperature = environmental$temperature
wind = environmental$wind
```

Create a categorical version of `radiation`:

```
radiation.cat = cut_number(radiation, n = 3)
levels(radiation.cat) = c("Low radiation", "Medium radiation", "High radiation")
```

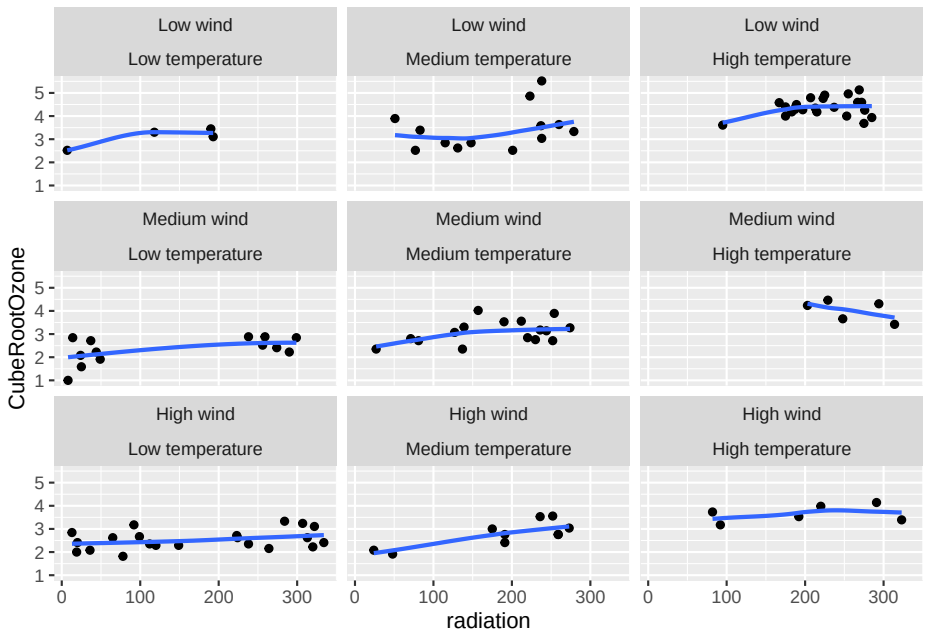
Same thing for `temperature` and `wind`:

```
temperature.cat = cut_number(temperature, n = 3)
levels(temperature.cat) = c("Low temperature", "Medium temperature", "High temperature")
wind.cat = cut_number(wind, n = 3)
levels(wind.cat) = c("Low wind", "Medium wind", "High wind")
```

Put everything into a data frame and facet two ways:

```
environmental.cat = data.frame(environmental, CubeRootOzone, radiation.cat, temperature.cat, wind
ggplot(environmental.cat, aes(x=radiation, y=CubeRootOzone)) + geom_point() +
  geom_smooth(span = 1, method.args = list(degree = 1, family = "symmetric"),
```

```
    se = FALSE) + facet_wrap(~wind.cat + temperature.cat)
```

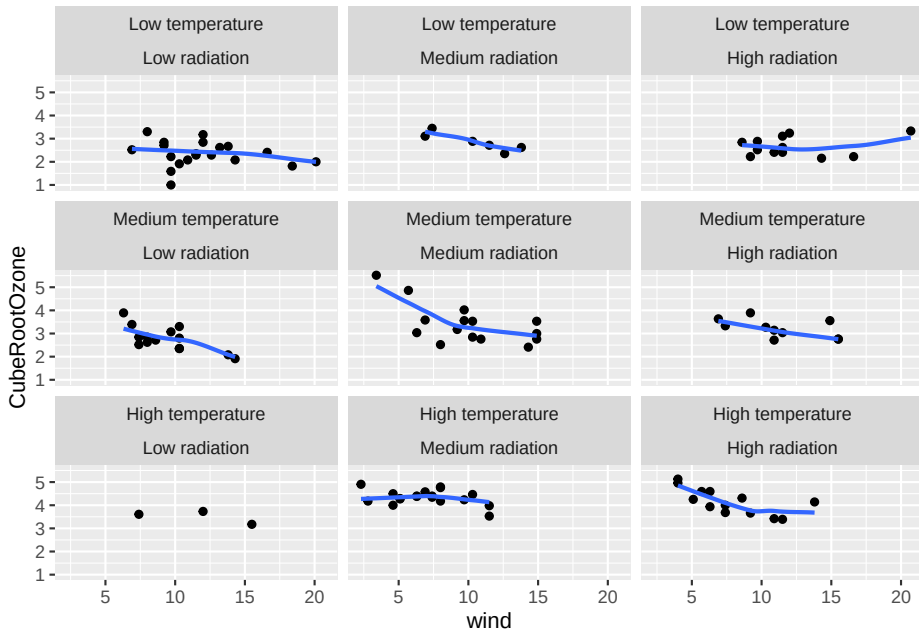


Ozone tends to increase with radiation. The levels of the fits, however, also change from top to bottom and from left to right, so the other variables will help predict as well.

Now make wind the main x-variable and facet on temperature and radiation.

```
ggplot(environmental.cat, aes(x = wind, y = CubeRootOzone)) + geom_point() +
  geom_smooth(span = 1, method.args = list(degree = 1, family = "symmetric"),
    se = FALSE) + facet_wrap(~temperature.cat + radiation.cat)
```

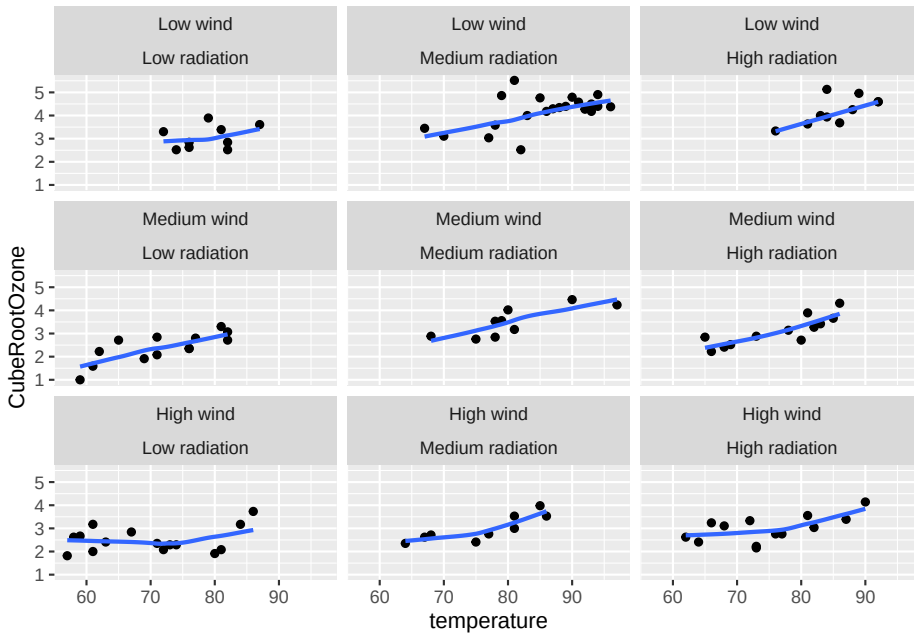
```
## Warning: Computation failed in `stat_smooth()`:
## NA/NaN/Inf in foreign function call (arg 1)
```



For once the error message is important. The high temperature, low radiation plot only has three observations. We make a mental note to *not* use the model for high temperature and low radiation situations later on because of the sparsity of data. Other than that, we see the relationship between ozone and wind is generally negative, though the slopes vary a lot.

Finally, make temperature the main x-variable and facet on wind and radiation.

```
ggplot(environmental.cat, aes(x = temperature, y = CubeRootOzone)) +
  geom_point() + geom_smooth(span = 1, method.args = list(degree = 1,
    family = "symmetric"), se = FALSE) + facet_wrap(~wind.cat + radiation.cat)
```



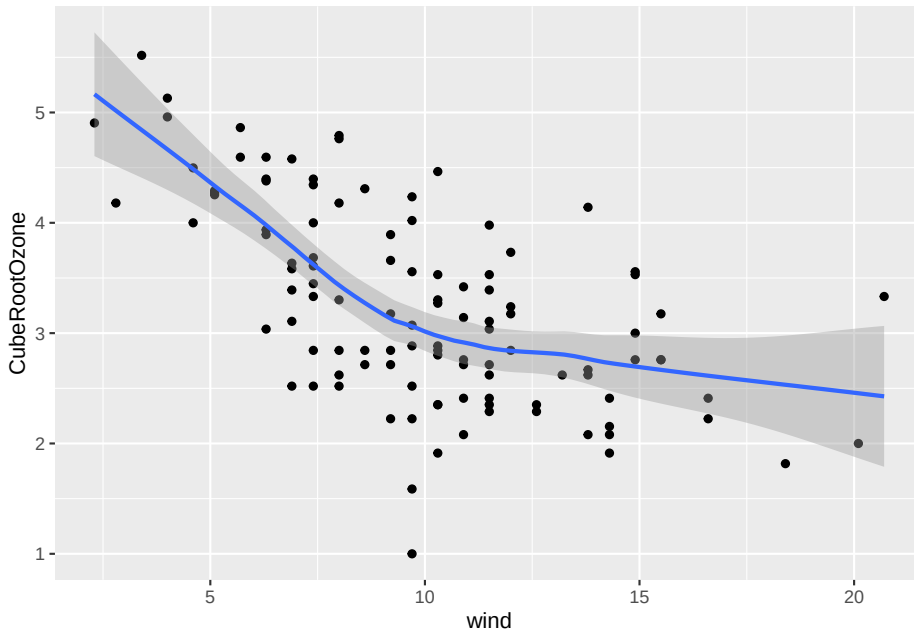
The slopes are pretty positive and consistent this time. Higher temperatures are associated with higher ozone.

6.1.2 Where is there no data?

Whenever the number of explanatory variables starts to get high, the **curse of dimensionality** becomes a concern: there may be large regions of predictor space where there's little or no data.

We're used to this idea with one predictor:

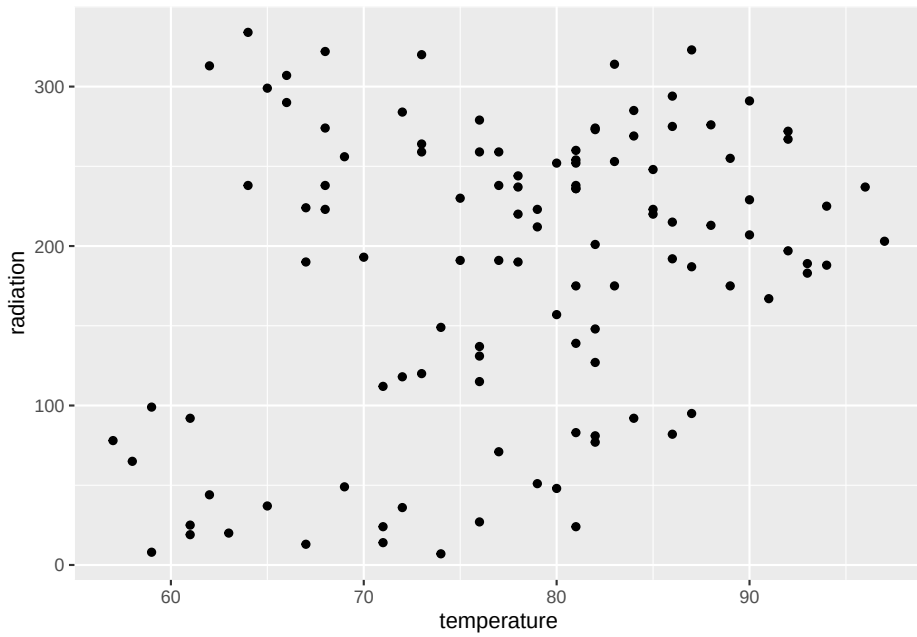
```
ggplot(environmental, aes(x = wind, y = CubeRootOzone)) + geom_point() +
  geom_smooth(span = 2/3, method.args = list(degree = 1))
```



As we go toward the extreme right, the confidence band gets wider, and then ggplot refuses to make any predictions to the right of the highest wind value – that would be extrapolation. We wouldn't want to use our model to predict the ozone concentration if the wind were, say, 35 miles per hour.

The same idea holds when there's more than one predictor, except the problem can be more subtle. Let's do a scatterplot of radiation against temperature.

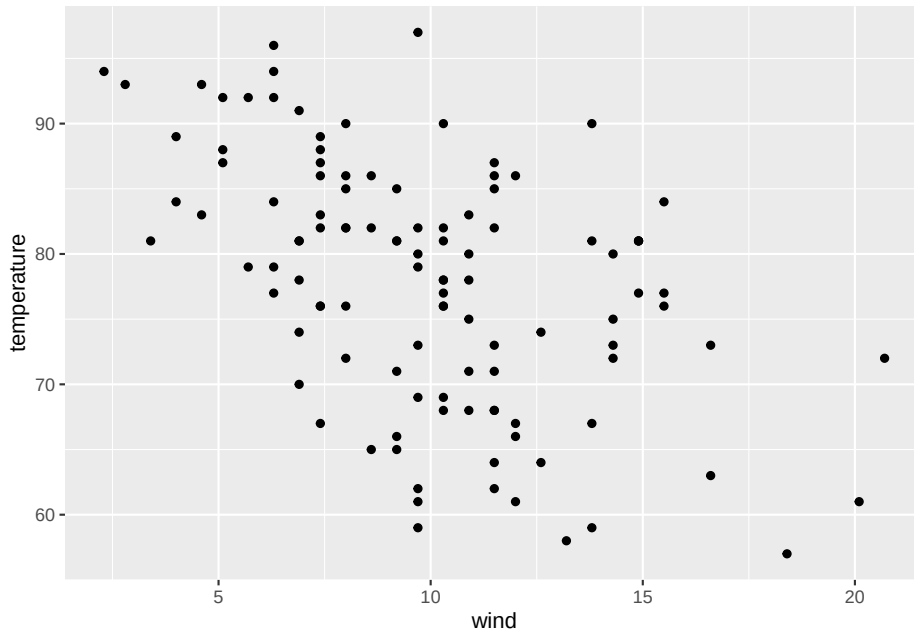
```
ggplot(environmental.cat, aes(x=temperature, y=radiation)) + geom_point()
```



Notice there are no observations in the bottom right corner. (We hinted at this when we faceted on both temperature and radiation above.) On the other hand, this is a fairly small corner of predictor space, so we might be able to ignore this.

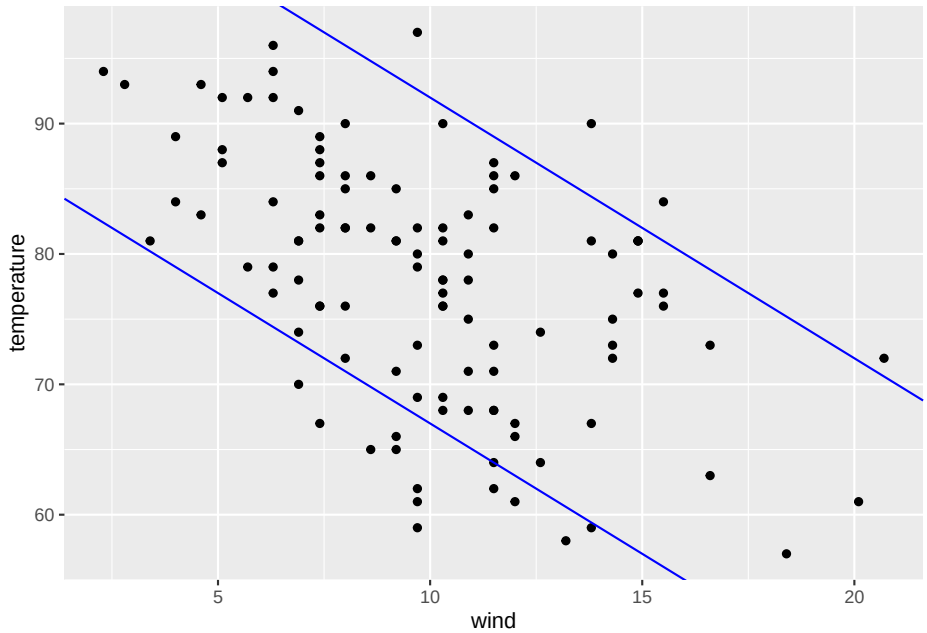
Now plot temperature against wind.

```
ggplot(environmental.cat, aes(x=wind, y=temperature)) + geom_point()
```



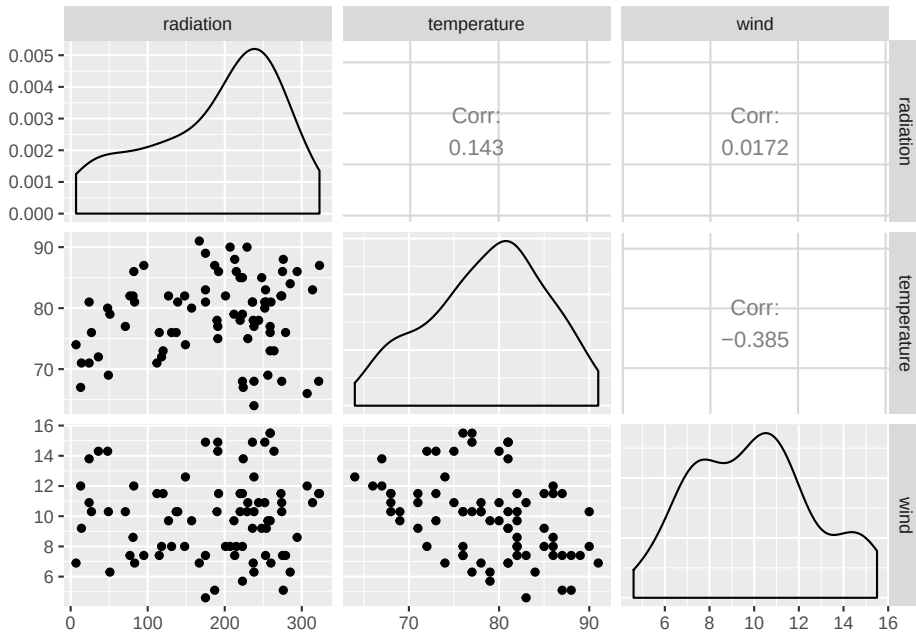
This is potentially a greater concern, because the vast majority of the data falls within a fairly narrow diagonal band of predictor space. Eyeballing the graph, it looks like for each value of wind, there's a range of about 25 degrees of temperature for which observations are dense. We add lines to the plot:

```
ggplot(environmental.cat, aes(x=wind, y=temperature)) + geom_point() +
  geom_abline(intercept = 112, slope = -2, color = "blue") +
  geom_abline(intercept = 87, slope = -2, color = "blue")
```



We shouldn't put too much creedence in our model outside the parallel lines. In addition, data is generally sparse outside temperatures of 60 to about 92 degrees and wind speeds outside 4 to 16 mph. We can *crop* predictor space and see if there are still "holes" in our data.

```
crop = (wind > 4) & (wind < 16) & (temperature < (-2 * wind + 112)) &
  (temperature > (-2 * wind + 87)) & (temperature > 60) & (temperature < 92)
environmental.crop = environmental.cat[crop, ]
ggpairs(environmental.crop, columns = 2:4)
```



Again, the main concern is the low radiation-high temperature corner, but it doesn't look that bad in context. We'll ignore it.

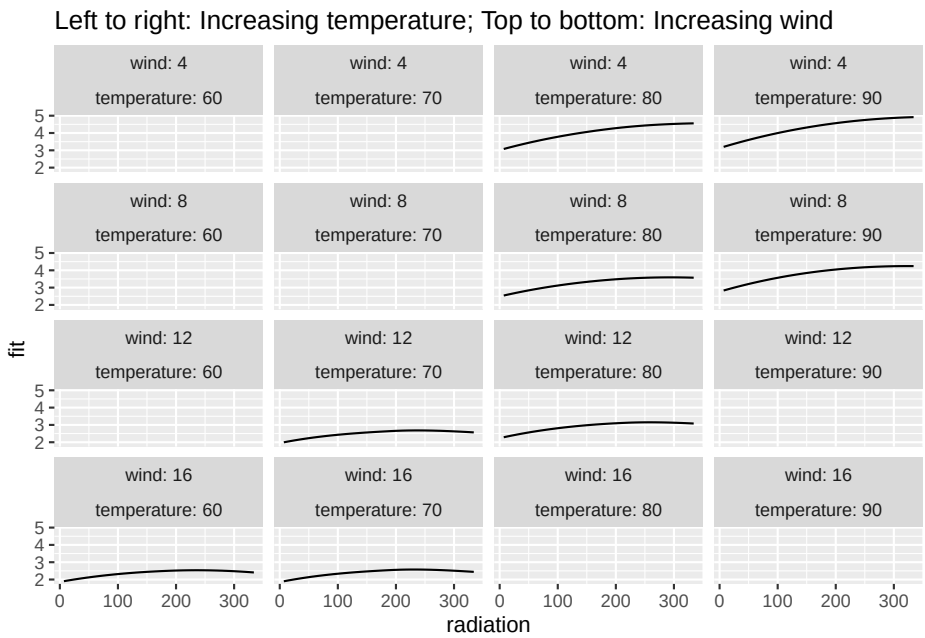
6.1.3 Visualizing the fit

The advantage of reducing the predictor space is that it's safer to fit a model with interactions. We first try a nonparametric loess model with a high `span` to reduce the risk of overfitting. The `loess()` function can fit all interactions up to the three-way, so let's just throw it all in there. There didn't seem to be any horrible outliers, so we fit using least squares.

```
environmental.lo = loess(CubeRootOzone ~ radiation * temperature * wind, span = 1)
```

Construct a grid and graph the relationship of fit to temperature, faceting on the other variables.

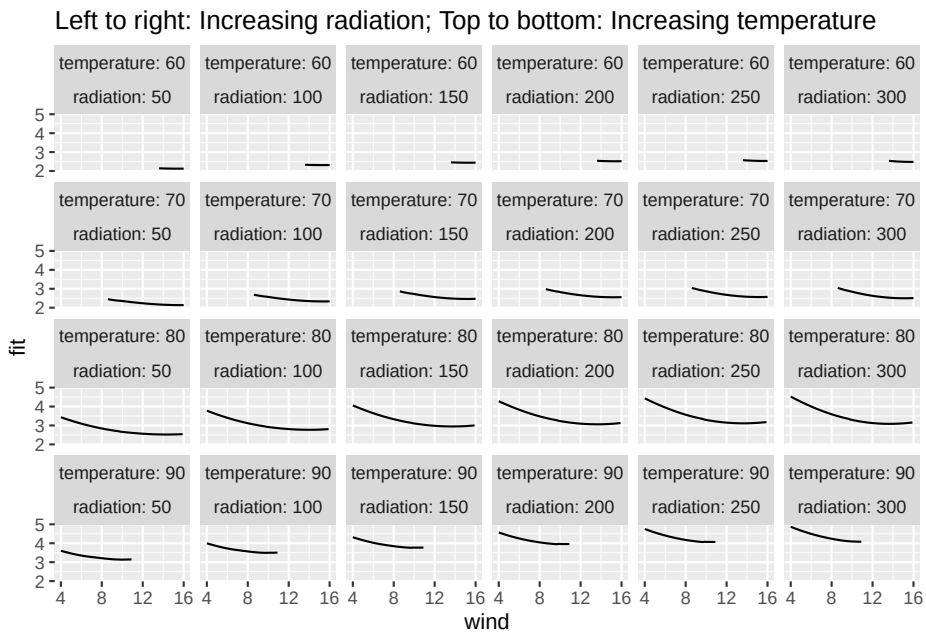
```
loess.grid = expand.grid(radiation = min(radiation):max(radiation),
  temperature = c(60, 70, 80, 90), wind = c(4, 8, 12, 16))
environmental.predict = predict(environmental.lo, newdata = loess.grid)
environmental.df = data.frame(loess.grid, fit = as.vector(environmental.predict))
crop.grid = (loess.grid$temperature < (-2 * loess.grid$wind + 112)) &
  (loess.grid$temperature > (-2 * loess.grid$wind + 87))
environmental.df = environmental.df[crop.grid, ]
gg = ggplot(environmental.df, aes(x = radiation, y = fit)) + geom_line() +
  facet_wrap(~wind + temperature, drop = FALSE, labeller = label_both)
gg + ggtitle("Left to right: Increasing temperature; Top to bottom: Increasing wind")
```



The curves do seem to change slope going downward, justifying the radiation:wind interaction.

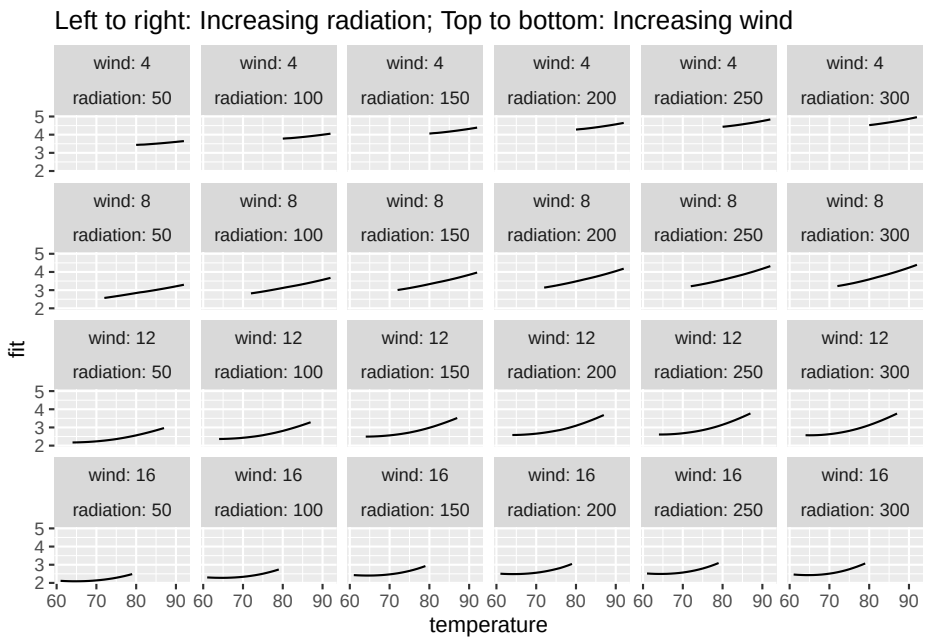
Now graph the relationship of fit with wind:

```
wind.grid = expand.grid(radiation = seq(50, 300, 50),
  temperature = c(60, 70, 80, 90), wind = seq(4, 16, 0.1))
environmental.predict = predict(environmental.lo, newdata=wind.grid)
environmental.df = data.frame(wind.grid, fit = as.vector(environmental.predict))
wind.crop = (wind.grid$temperature < (-2 * wind.grid$wind + 112)) &
  (wind.grid$temperature > (-2 * wind.grid$wind + 87))
environmental.df = environmental.df[wind.crop, ]
gg = ggplot(environmental.df, aes(x = wind, y = fit)) + geom_line() +
  facet_wrap(~temperature + radiation, drop = FALSE, ncol = 6, labeller = label_both)
gg + ggtitle("Left to right: Increasing radiation; Top to bottom: Increasing temperatu
```



And the relationship of fit with temperature:

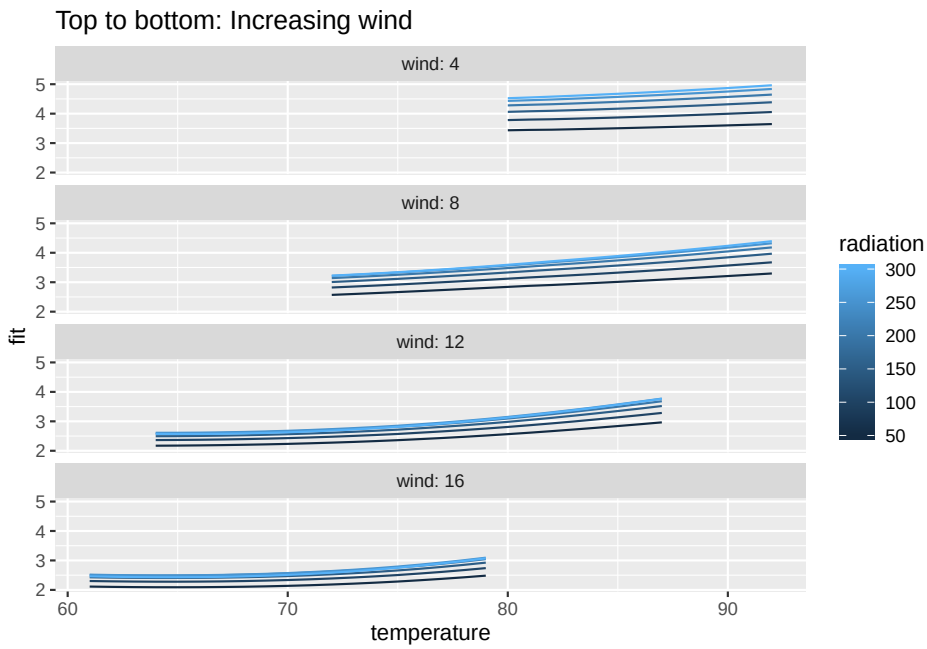
```
temperature.grid = expand.grid(radiation = seq(50, 300, 50),
  temperature = 61:92, wind = c(4, 8, 12, 16))
environmental.predict = predict(environmental.lo, newdata = temperature.grid)
environmental.df = data.frame(temperature.grid,
  fit = as.vector(environmental.predict))
temperature.crop = (temperature.grid$temperature < (-2 * temperature.grid$wind + 112)) &
  (temperature.grid$temperature > (-2 * temperature.grid$wind + 87))
environmental.df = environmental.df[temperature.crop, ]
gg = ggplot(environmental.df, aes(x = temperature, y = fit)) + geom_line() + facet_wrap(~wind + r
gg + ggtitle("Left to right: Increasing radiation; Top to bottom: Increasing wind")
```



Note that just about everything in these plots was curved, justifying the loess fit instead of `lm()`. In the last set, the slope seemed to change going downward, justifying the temperature:wind radiation.

To look more closely at the radiation:temperature interaction, we'll take the last set of plots and collapse all the columns of top of each other.

```
gg = ggplot(environmental.df, aes(x = temperature, y = fit,
    group = radiation, color = radiation)) + geom_line() +
  facet_wrap(~wind, drop = FALSE, ncol = 1, labeller = label_both)
gg + labs(title = "Top to bottom: Increasing wind")
```

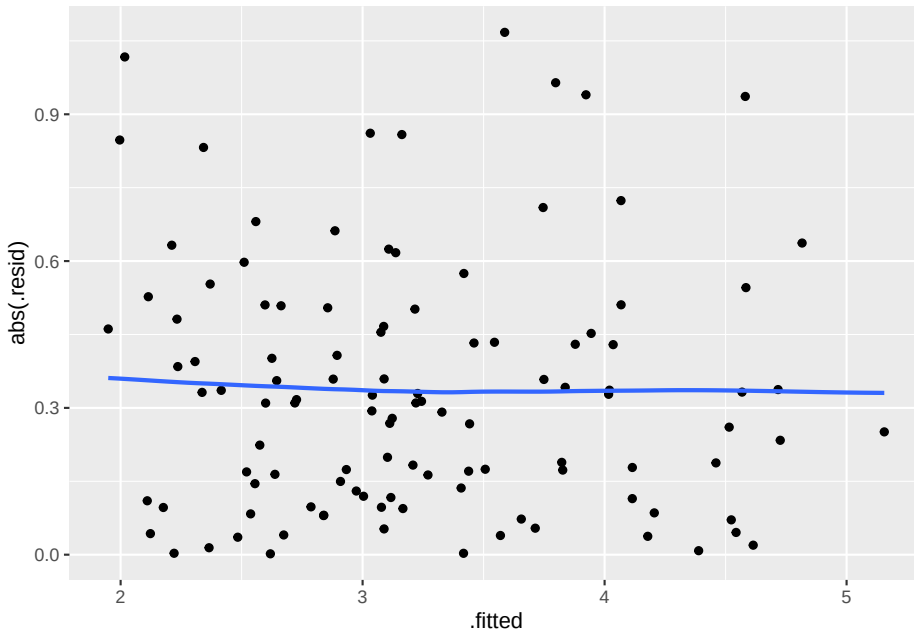


This is a nice one-plot summary of the model. (An alternative would be to draw contour plots faceted by the remaining explanatory variable; let's, uh, leave this as an exercise.) There's probably a little interaction between temperature and radiation, but not much. We could include it or leave it out of the model (by fitting a GAM, for example) depending on whether you prefer maximalism or minimalism.

6.1.4 Was the transformation a good idea?

Homoscedasticity is a good goal: even if we don't strictly need it, it's usually a signal that we're on the right track. We can do our usual spread-location plot of absolute residuals:

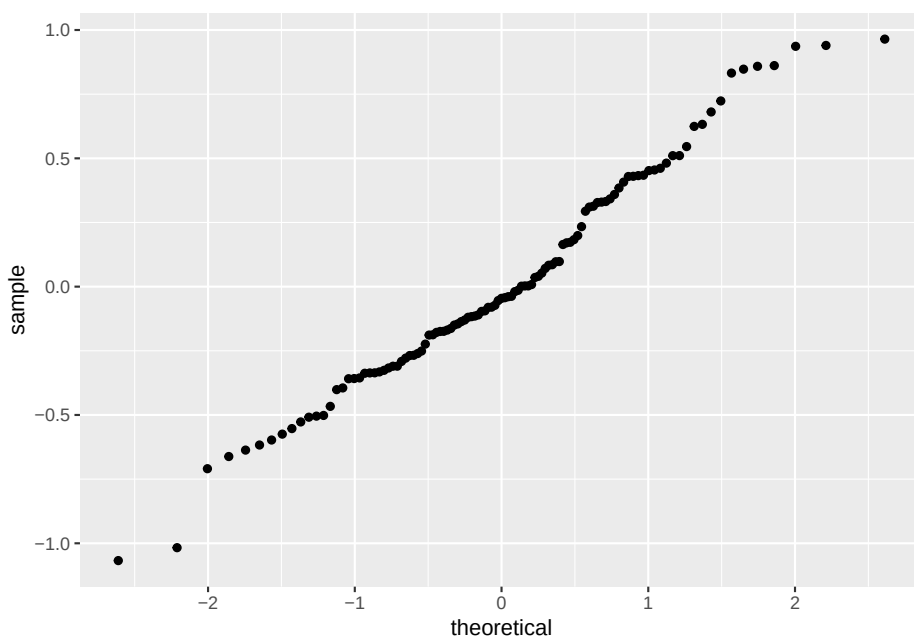
```
environmental.lo.broom = augment(environmental.lo)
ggplot(environmental.lo.broom, aes(x = .fitted, y = abs(.resid))) + geom_point() +
  geom_smooth(span = 1, method.args = list(degree = 1), se  = FALSE)
```



Looks like it was. (Trying out other transformations generally makes this plot worse.)

Since everything is going well, let's check for normality:

```
ggplot(environmental.lo.broom, aes(sample = .resid)) + stat_qq()
```



While not strictly normal, this is surprisingly good, with no heavy tail or outlier issues. Least squares was an appropriate choice for our fitting method.

Finally, compare the fitted values and the residuals.

```
fit.var = var(environmental.lo.broom$.fitted)
fit.var / var(CubeRootOzone)
```

```
## [1] 0.7287185
```

```
fit.var / (fit.var + var(environmental.lo.broom$.resid))
```

```
## [1] 0.7626197
```

Our model captures about three-quarters of the variation in `CubeRootOzone`. Note that even with there relevant predictors, there's still a substantial amount left unexplained. That's okay: nature is inherently hard to explain with just a few variables.

6.2 Four or more variables

READ: Cleveland pp. 293–301.

6.2.1 Hamster organs

Load stuff:

```
load("lattice.RData")
library(ggplot2)
library(GGally)
library(tidyr)
```

For some reason Cleveland has data on the weights (in grams) of six organs from each of 73 hamsters that died of a congenital heart problem. Here, there's no response variable: we're just interested in how the variables relate to each other. Before we jump to the multivariate stuff, we see what the univariate data looks like.

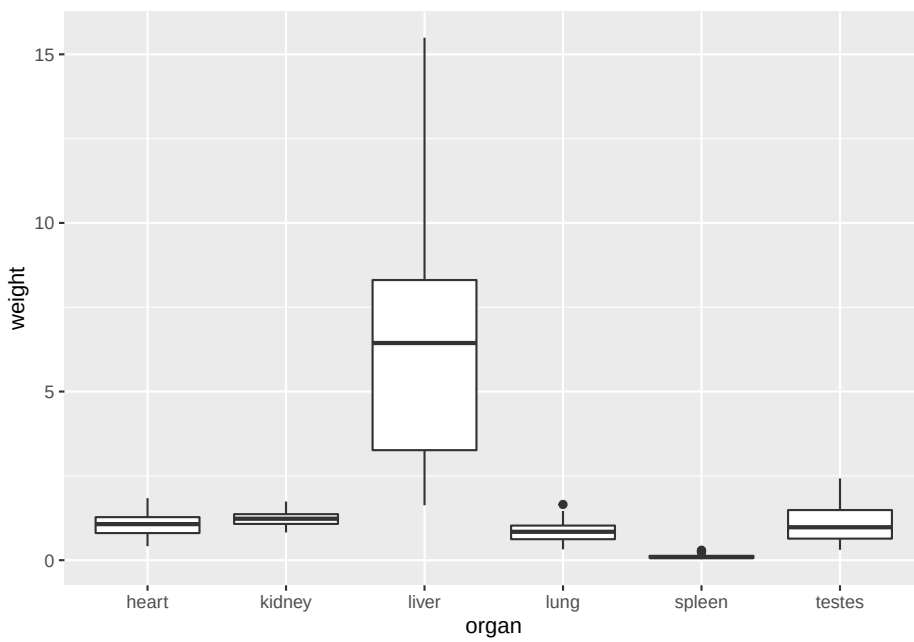
```
summary(hamster)
```

```
## lung heart liver spleen
## Min.  :0.3216 Min.  :0.4134 Min.  : 1.626 Min.  :0.0293
## 1st Qu.:0.6220 1st Qu.:0.8039 1st Qu.: 3.262 1st Qu.:0.0657
## Median :0.8416 Median :1.0690 Median : 6.439 Median :0.0896
## Mean :0.8457 Mean :1.0581 Mean : 6.299 Mean :0.1035
## 3rd Qu.:1.0260 3rd Qu.:1.2763 3rd Qu.: 8.307 3rd Qu.:0.1283
## Max.  :1.6521 Max.  :1.8390 Max.  :15.492 Max.  :0.2959
## kidney testes
## Min.  :0.8199 Min.  :0.3082
```

```
## 1st Qu.:1.0751 1st Qu.:0.6387
## Median :1.2291 Median :0.9761
## Mean :1.2256 Mean :1.0900
## 3rd Qu.:1.3642 3rd Qu.:1.4862
## Max.  :1.7381 Max.  :2.4244
```

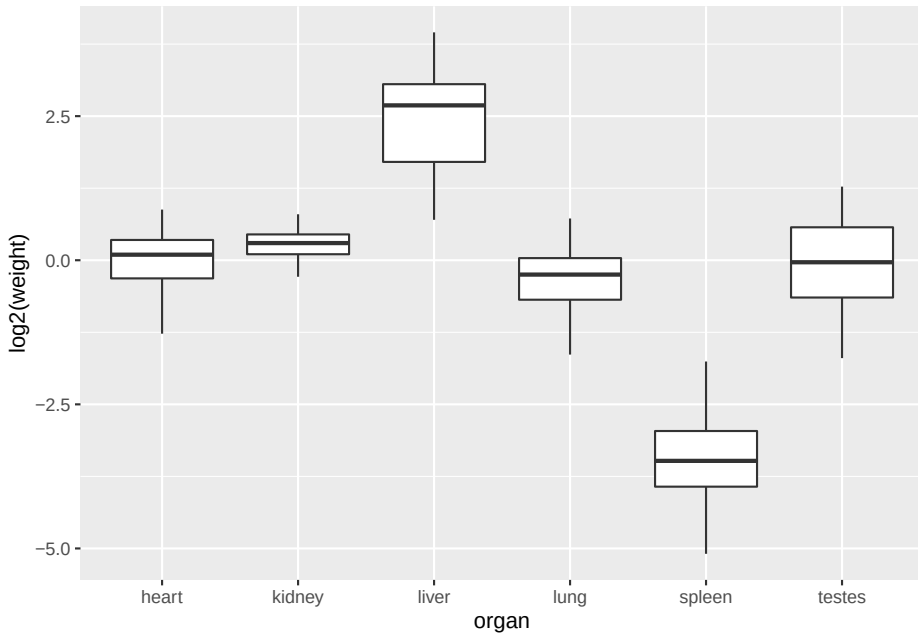
Let's put the data into long form and plot it.

```
hamster.long = hamster %>% gather(organ, weight)
ggplot(hamster.long, aes(x = organ, y = weight)) + stat_boxplot()
```



The different orders of magnitude make it hard to see what's going on. We can use a log scale instead.

```
ggplot(hamster.long, aes(x = organ, y = log2(weight))) + stat_boxplot()
```

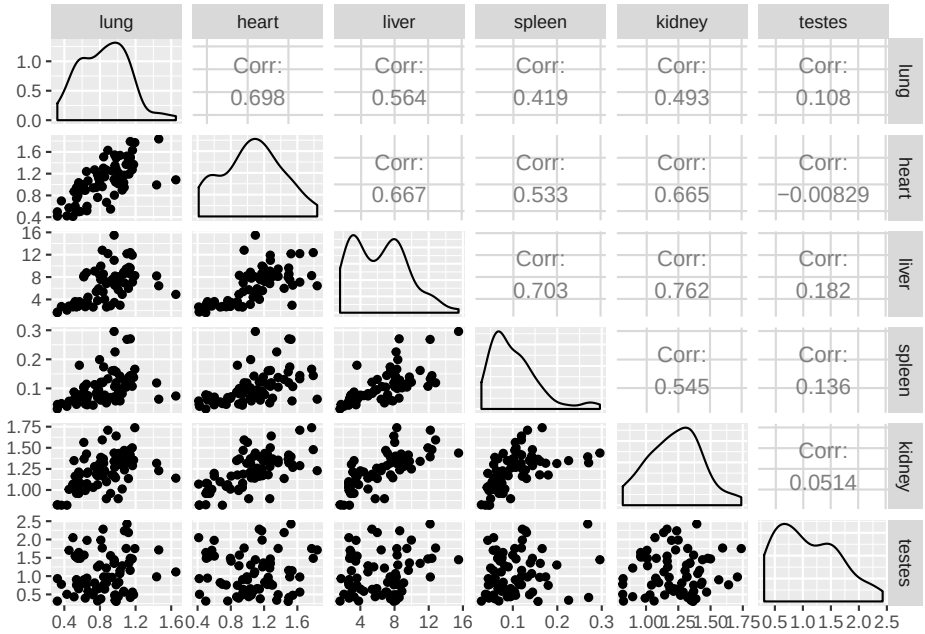


Now it's clear that livers are the heaviest organ and spleens are the smallest. The others are of similar order of magnitude.

Note that the reason we transformed here was magnitude, not skewness. In scatterplots, we can use different scales on different axes, so we don't necessarily have to pursue this transformation.

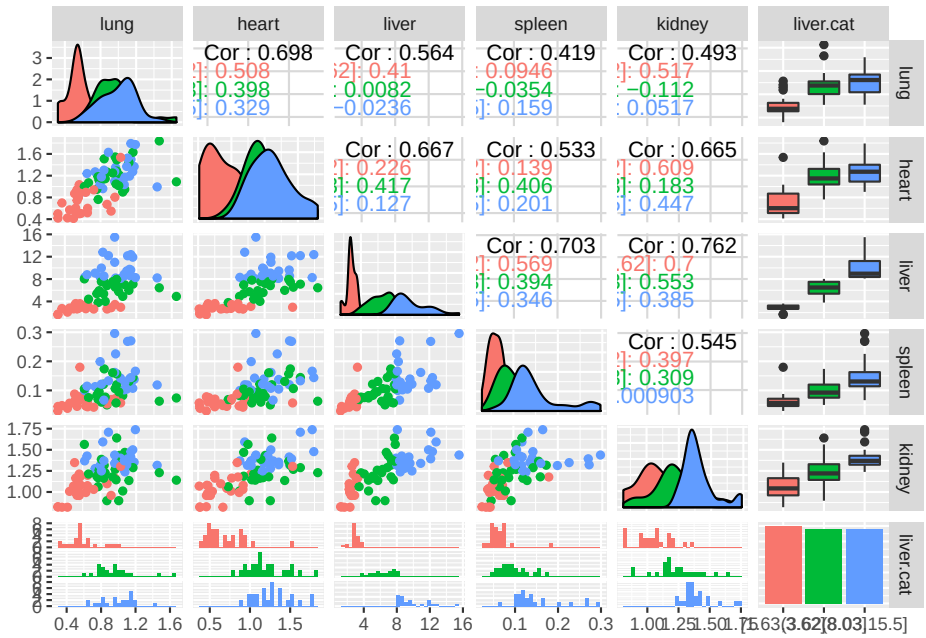
Let's just draw paired scatterplots on the original scale:

```
ggpairs(hamster)
```



Some variables are highly associated, like heart-lung-liver and liver-kidney-spleen. Only testes weight doesn't seem to be related to anything else, so we'll forget it. In some ways liver seems to be the key variable, since it's highly correlated with everything besides testes. We can make chopped liver and add color:

```
liver.cat = cut_number(hamster$liver, n=3)
ggpairs(data.frame(hamster[,1:5], liver.cat), aes(color = liver.cat))
```



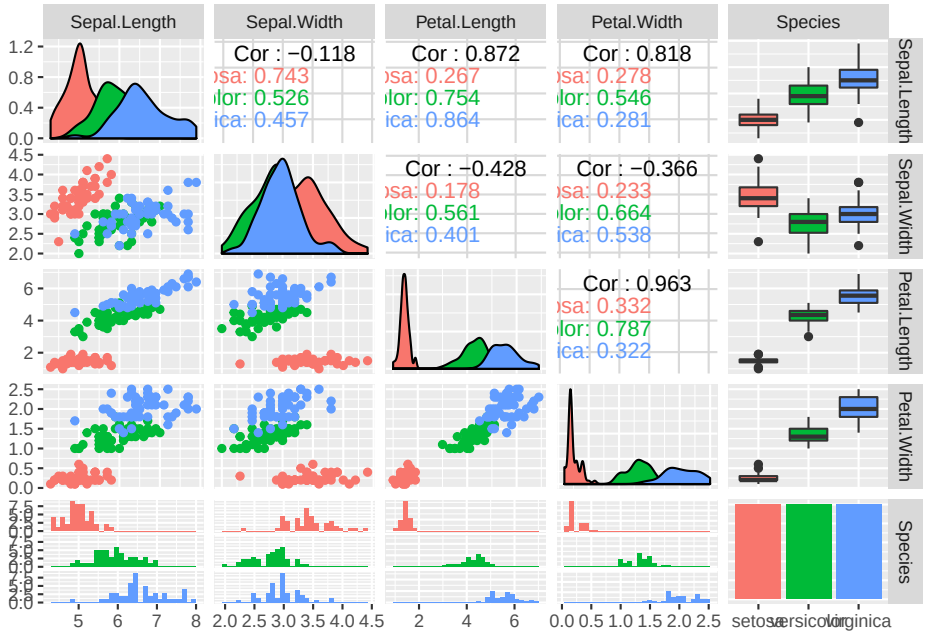
We see that hamsters with small livers have small other organs. Hamsters with big livers generally have the biggers kidneys and spleens, but their lung and heart sizes substantially overlap with those of medium-livered hamsters.

If you want to try out the *brushing* method described in Cleveland, you can try out the `rggobi` package. We're not going to bother though.

6.2.2 Is it the iris data? Yes, it's the iris data

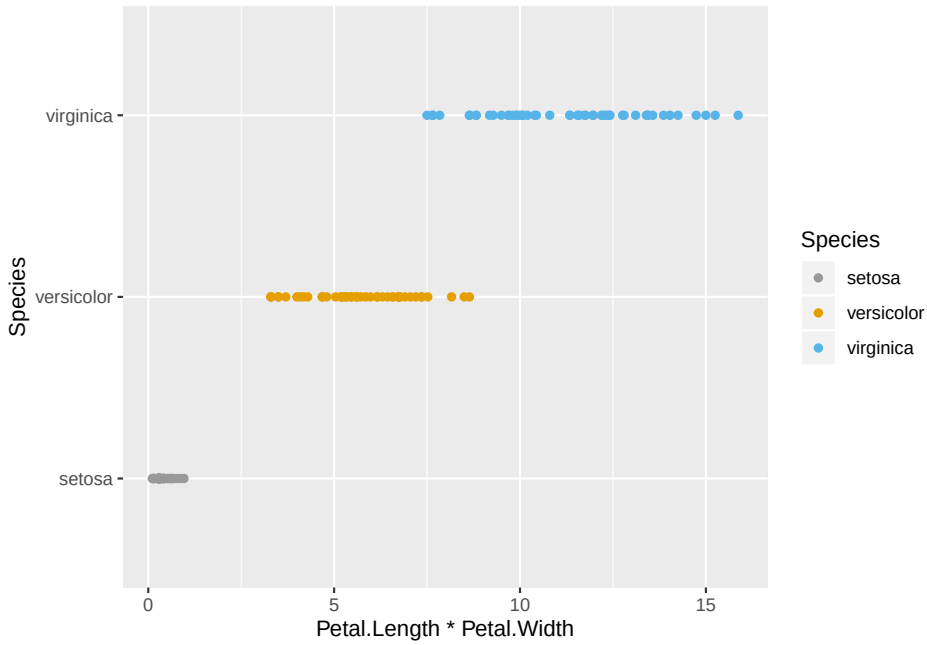
The iris data set, which is studied in every statistics class, consists of measures of sepal and petal length and width (in cm) of 50 setosa irises, 50 versicolor irises, and 50 virginica irises. We want to be able to tell the three species apart based on these measurements. In our case, we want a practical rule rather than a "machine learning" rule – one that we can understand without spending half our lives studying irises.

```
ggpairs(iris, aes(color = Species))
```



The 1D plots are clear enough – blue is generally bigger than green, and green is generally bigger than red (except in terms of sepal width.) Since after years of looking at this data set I still don't know what a sepal is, let's stick to petals. How do we combine length and width into one number? Well, if the petal was rectangular, length times width would give area. The petals obviously aren't rectangular, but we can still do the multiplication and get something roughly proportional to area.

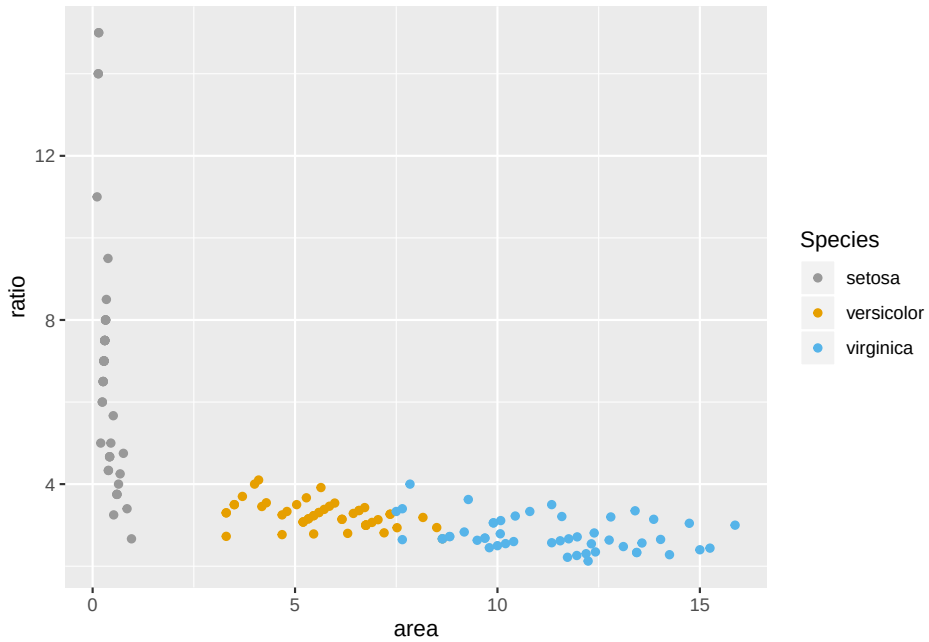
```
ggplot(iris, aes(y = Species, x = Petal.Length * Petal.Width, color = Species)) +
  geom_point() + scale_color_manual(values = cb_palette)
```



It looks like apart from the very biggest versicolor and the very smallest virginica, petal area classifies correctly.

What can we plot area against? Well, the information you lose when you multiply is the ratio of the two numbers. If we have both the area and the ratio of length to width, then we lose no information (we can recover both length and width.) Let's scatterplot these:

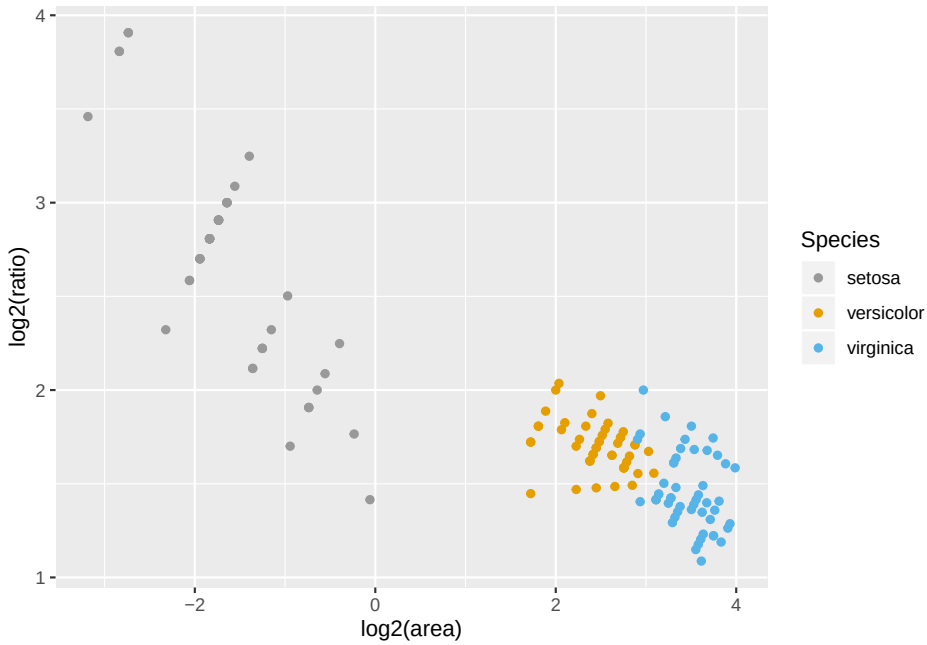
```
area = iris$Petal.Length * iris$Petal.Width
ratio = iris$Petal.Length / iris$Petal.Width
iris.size = data.frame(iris, area, ratio)
ggplot(iris.size, aes(x = area, y = ratio, color=Species)) +
  geom_point() + scale_color_manual(values = cb_palette)
```



So we get a simple rule: If petal length times width is less than 1 cm^2, it's setosa. If it's 3 to 7, it's versicolor, and if it's more than 9, it's virginica. If it's between 7 and 9 then you need to look more carefully.

The boundary between versicolor and virginica is clearer on a log scale.

```
ggplot(iris.size, aes(x = log2(area), y = log2(ratio), color = Species)) +
  geom_point() + scale_color_manual(values = cb_palette)
```



If we simply draw vertical lines at 1 and 3 (which would be 2 and 8 cm^2 on the original scale), we'll only be wrong 7 out of 150 times. (By drawing the lines more carefully we can reduce the training set errors to 3, but if you're out hiking in Quebec, you probably don't want to have to run a linear discriminant analysis every time you see flowers.)

Chapter 7

Binary responses

7.1 Logistic regression, plus dealing with messy data

Recommended reading: Gelman & Hill pp. 79–86. For a more detailed and more traditional treatment, try Agresti, Categorical Data Analysis, ch. 5 & 6 (3rd edition.)

7.1.1 ANES

Download the ANES Time Series Cumulative Data File, containing data from 1948 to 2016 (registration required):

https://electionstudies.org/data-center/

The raw data is just an astonishing mess, so get the Stata file. The `import` function in `library(rio)` is my preference for reading exotic data formats into R, though the `haven` package also works if you want something tidyverse-friendly.

```
library(rio)
ANES = import("anes_timeseries_cdf.dta")
```

What do we care about? Suppose we care about income. Looking at the codebook for the study, the relevant variable is `VCF0114`.

```
income = ANES$VCF0114
summary(income)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   0.000   2.000   3.000   2.676   4.000   5.000    1511
```

Income is measured on a scale from 1 to 5:

1: 0 to 16 percentile

2: 17 to 33 percentile

3: 34 to 67 percentile

4: 68 to 95 percentile

5: 96 to 100 percentile

(The zeroes and NA's are missing values.) This allows comparability between years. This is really an ordinal variable but we might find some advantages in treating it as quantitative.

Next we need the year for each observation. This is `VCF0004`.

```
year = ANES$VCF0004
summary(year)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1948    1970    1986    1985    2002    2016
```

Now we need a vote variable. In order to do logistic regression, we need a binary variable. The variable `VCF0704a` gives the *two-party* Presidential vote: that is, those who voted for third-parties or didn't vote should be missing values.

```
vote = ANES$VCF0704a
summary(vote)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   0.000   0.000   1.000   0.938   2.000   2.000   20724
```

The coding is that "1" means the Democrat, "2" means the Republican, and "0" or "NA" means some other outcome. We want everything to be coded as 0, 1, or NA. First, change the zeroes to NA's:

```
vote[vote == 0] = NA
summary(vote)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.00    1.00    1.00    1.48    2.00    2.00   35118
```

Now suppose we make 1 represent Republican and 0 represent Democrat. This just requires subtracting 1:

```
vote = vote - 1
summary(vote)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    0.00    0.00    0.00    0.48    1.00    1.00   35118
```

The variable really represents a two-party vote for a Republican now, so for clarity let's just rename it as such.

```
Republican = vote
```

Let's also include one other variable called `survey.weights` that we'll end up using later on.

```
survey.weights = ANES$VCF0009z
summary(survey.weights)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0212  1.0000  1.0000  1.0453  1.0000  6.8130
```

Now let's put our four variables into a data frame.

```
ANES.df = data.frame(year, income, Republican, survey.weights)
```

Finally, I only want data for Presidential election years (years divisible by 4.) I'll use the `filter()` function in `dplyr`:

```
ANES.df = filter(ANES.df, year %in% seq(1948, 2016, 4))
summary(ANES.df)
```

```
## year income Republican survey.weights
## Min.  :1948 Min.  :0.000 Min.  :0.000 Min.  :0.0212
## 1st Qu.:1972 1st Qu.:2.000 1st Qu.:0.000 1st Qu.:0.9038
## Median :1992 Median :3.000 Median :0.000 Median :1.0000
## Mean :1989 Mean :2.677 Mean :0.481 Mean :1.0356
## 3rd Qu.:2012 3rd Qu.:4.000 3rd Qu.:1.000 3rd Qu.:1.0000
## Max.  :2016 Max.  :5.000 Max.  :1.000 Max.  :6.8130
## NA's :14394
```

Let's save this so we can use it later:

```
write.table(ANES.df, file = "ANES.txt", row.names = FALSE)
```

7.1.2 The 1992 election

Let's start by picking out one year and looking at the relationship between income and vote. Let's choose 1992, which pitted incumbent Republican George H.W. Bush against Democratic challenger Bill Clinton.

```
ANES1992 = subset(ANES.df, year == 1992)
summary(ANES1992)
```

```
## year income Republican survey.weights
## Min.  :1992 Min.  :0.000 Min.  :0.0000 Min.  :0.8462
## 1st Qu.:1992 1st Qu.:2.000 1st Qu.:0.0000 1st Qu.:0.9872
## Median :1992 Median :3.000 Median :0.0000 Median :1.0000
## Mean :1992 Mean :2.688 Mean :0.4156 Mean :1.0011
## 3rd Qu.:1992 3rd Qu.:4.000 3rd Qu.:1.0000 3rd Qu.:1.0189
## Max.  :1992 Max.  :5.000 Max.  :1.0000 Max.  :1.4615
```

```
## NA's :1128
```

The summary is a bit indirect for the `Republican` variable. With `tidyverse`, it's easy to do the count directly:

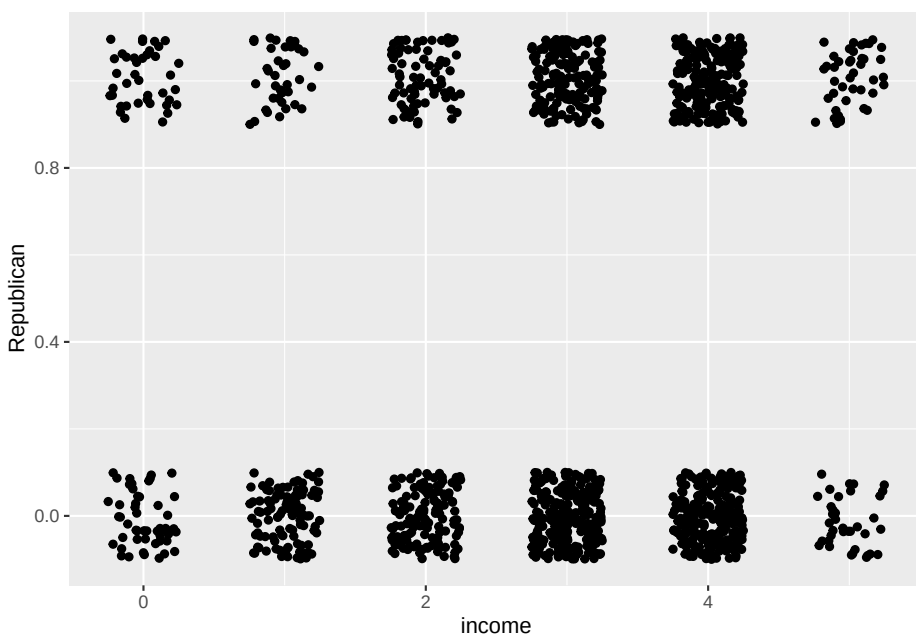
```
count(ANES1992, Republican)
```

```
## # A tibble: 3 x 2
##   Republican     n
##        <dbl> <int>
## 1          0   793
## 2          1   564
## 3         NA  1128
```

7.1.3 Logistic regression with one predictor

Let's now look at the relationship between income and vote in 1992. If we want to look at a scatterplot, we need to jitter it, as there are only five or six x-values and two y-values.

```
ggplot(ANES1992, aes(x = income, y = Republican)) + geom_jitter(height = 0.1, width = 0
```



We can also summarize the data quantitatively:

```
aggregate(Republican ~ income, mean, data = ANES1992)
```

```
##   income Republican
## 1      0  0.4333333
```

```
## 2       1  0.2671233
## 3       2  0.3454545
## 4       3  0.4035533
## 5       4  0.4883178
## 6       5  0.5316456
```

This gives the proportion (out of major party voters) who voted for Bush for each income group. Aside from group zero, which represents missing values of income, we see a strictly increasing pattern. How do we model this? Three options (not the only three) include:

1. Linear regression with income as a numerical predictor.
2. Logistic regression with income as a numerical predictor.
3. Regression with income as a categorical (factor) predictor. (In this case, linear and logistic give identical predictions.)

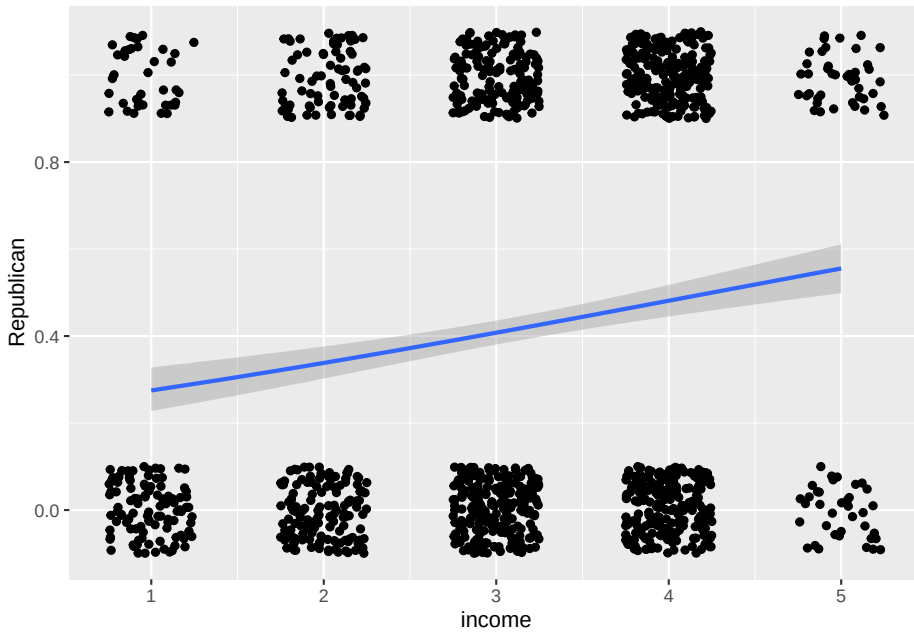
Method 1 is the easiest to interpret: we get a slope that directly tells us the change in model probability of voting Republican as income goes up one category. However, linear regression for binary responses has both technical and social limitations. The technical limitation is that it only works well when probability are far from 0 and 1. Otherwise, if x is unbounded, you can end up with negative probabilities or probabilities greater than 1. The social limitation is that about two-thirds of statistics professors will never speak to you again if they see you doing linear regression on binary data. While this may be considered a positive by some, it is not really feasible for the author, so I do not pursue it here.

Method 3 isn't really a model at all: it just returns the proportion within each category who voted for Bush, the same as our `aggregate()` call gave us above. There's something to be said for not fitting restrictive models when you don't have to. However, if our goal is to fit more complex models or make comparisons between different sets of data, as it eventually will be, then some degree of simplification may be useful to understand the patterns in the data. Or we might fit a simplifying model first, then go back and look at the data in more detail and see if there are any important features our model missed. That will be our basic approach here.

Method 2, logistic regression, should work well. It does require treating a predictor that isn't *inherently* a numeric variable as numeric, and requires a parametric form (effects are linear on a logit scale.) However, most of the time, doing this is reasonable as long as the pattern of the probability with x is monotonic and as long as predictive accuracy is not the sole goal. (If the pattern was non-monotonic, a nonparametric method like a generalized additive model (GAM) would be a better modeling choice. If predictive accuracy is the sole goal then it increasingly seems like you have to learn neural networks, which I haven't.)

We fit such a logistic regression using `income` as a quantitative variable and omitting missing values. Logistic regression is a special case of a GLM, so we use the `glm()` function; specifying a binomial family fits a logistic regression by default. Firstly, we can just add the fitted curve to the jittered plot:

```
ANES1992 = subset(ANES1992, income > 0)
ggplot(ANES1992, aes(x = income, y = Republican)) +
  geom_jitter(height = 0.1, width = 0.25) +
  geom_smooth(method = "glm", method.args = list(family = "binomial"))
```



We can also fit it explicitly:

```
Bush.logit = glm(Republican ~ income, family = binomial, data = ANES1992)
summary(Bush.logit)
```

```
##
## Call:
## glm(formula = Republican ~ income, family = binomial, data =
ANES1992)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -1.2733 -1.0235 -0.9086 1.2096 1.6069
##
## Coefficients:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.26750 0.17789 -7.125 1.04e-12 ***
## income 0.29802 0.05379 5.541 3.01e-08 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 1719.1 on 1266 degrees of freedom
## Residual deviance: 1687.2 on 1265 degrees of freedom
## (1014 observations deleted due to missingness)
## AIC: 1691.2
##
## Number of Fisher Scoring iterations: 4
```

The summary gives a lot of information; we'll focus on the coefficients. The summary tells us that

$$\text{logit}[\text{P(Bush)}] = -1.27 + 0.298 \times \text{income}$$

where the logit function is

$$\text{logit}(p) = \log_e \frac{p}{1-p}.$$

To find P(Bush), we invert the logit:

$$\text{P(Bush)} = \frac{e^{g(x)}}{1+e^{g(x)}}$$

where

$$y = \text{logit}[\text{P(Bush)}].$$

For a quick and dirty interpretation, the "divide by 4" rule is useful: the maximum change in probability associated with a one unit change in x is the coefficient of x divided by four. So going one income group changes the model probability by up to about 7.5%. This looks like it's about the increase in the curve from income group 4 to group 5.

```r
library(boot)
inv.logit(-1.27 + 0.298 * 4)
```

```
## [1] 0.4805099
```

```r
inv.logit(-1.27 + 0.298 * 5)
```

```
## [1] 0.5547792
```

7.1.4 Weighted regression

For various reasons like design and nonresponse bias, modern survey results are rarely a true simple random sample from the population. To adjust for groups being underrepresented or overrepresented in a sample, survey results are **weighted**. In particular, the ANES variable `VCF0009z` contains weights to make the sample resemble the demographics of the Current Population Survey. (Note that this doesn't remove all biases, e.g. it doesn't account for people lying to you about whether they voted, so further adjustments using a voter file would improve things further.)

Using weights in logistic (and linear) regression in R is easy: just use the `weights` argument. Technically once we have weights we're no longer fitting a binomial, so use `family = quasibinomial` (this actually doesn't make a numerical difference here, but it doesn't give a warning message.)

```
Bush.weighted.logit = glm(Republican ~ income, family = quasibinomial,
  weights = survey.weights, data = ANES1992)
summary(Bush.weighted.logit)
```

```
##
## Call:
## glm(formula = Republican ~ income, family = quasibinomial,
data = ANES1992,
## weights = survey.weights)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -1.3005 -1.0331 -0.8763 1.2091 1.7976
##
## Coefficients:
## Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.27068 0.17721 -7.170 1.27e-12 ***
## income 0.29908 0.05373 5.566 3.18e-08 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for quasibinomial family taken to be
0.9998299)
##
## Null deviance: 1715.7 on 1266 degrees of freedom
## Residual deviance: 1683.5 on 1265 degrees of freedom
## (1014 observations deleted due to missingness)
## AIC: NA
##
## Number of Fisher Scoring iterations: 4
```

The weights only make a small difference to the coefficients. Does this make

much difference to the fit? We write a function to describe the fit:

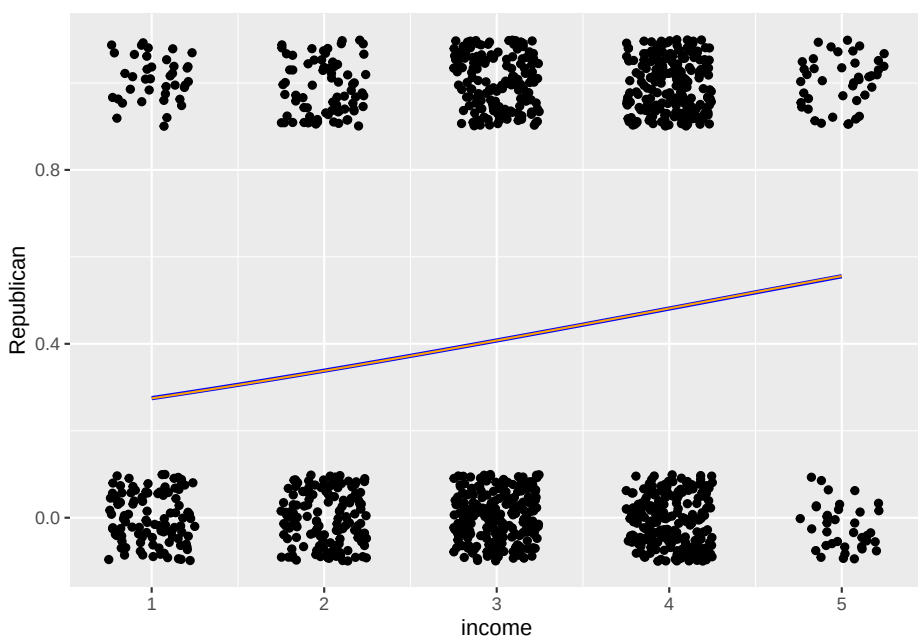
```
our.logit = function(x){
  coe = coef(Bush.weighted.logit)
  y = coe[1] + coe[2] * x
  return(exp(y) / (1 + exp(y)))
}
```

Now plot the unweighted and weighted fits.

```
ggplot(ANES1992, aes(x = income, y = Republican)) +
  geom_jitter(height = 0.1, width=0.25) +
  geom_smooth(method = "glm", method.args = list(family = "binomial"),
              se = FALSE, color = "blue") +
  stat_function(fun = our.logit, color = "orange")
```

```
## Warning: Removed 1014 rows containing non-finite values (stat_smooth).

## Warning: Removed 1014 rows containing missing values (geom_point).
```



The weighted and unweighted fits are nearly indistinguishable. This is quite often the case when creating regression models with high-quality data. (On the other hand, if you're just doing simple summaries of the data, or if the data is from a low-quality source such as a random digit dial poll, weights often make a big difference.) If someone has gone to the trouble of finding weights for you, they're easy to use, so you should use them.

7.1.5 Fitting a series of regressions

We're not just interested in 1992. We want to know the relationship between income and vote for every Presidential election we have data for – is the relationship similar for every election, or are some elections different? Has there been a consistent change over time?

In programming terms, the easiest way to fit the same model on many subsets of the data is to write a function that subsets the data and fits the model, then to write a `for` loop to fit the model for each subset. If you're literally going to run out of RAM, there are much more computationally efficient approaches, but otherwise more efficiency usually isn't worth the effort.

Here's a function to fit our weighted logistic regression of vote on income for any given year.

```
logit.ANES.subset = function(my.year, data){
  newdata = subset(data, year == my.year)
  newdata = subset(newdata, income > 0)
  model = glm(Republican ~ income, family = quasibinomial, weights = survey.weights, da
  output = c(my.year, summary(model)$coef[2,1:2])
  return(output)
}
```

The function returns the year, the model's coefficient for income, and the standard error of that coefficient. We shouldn't take the standard error too literally, because we haven't accounted for the complex design of the ANES surveys – if you really care about getting these right, take a sampling course.

Let's test the function out on Bush-Clinton.

```
logit.ANES.subset(my.year = 1992, data = ANES.df)
```

```
##                  Estimate   Std. Error
## 1.992000e+03 2.990845e-01 5.373261e-02
```

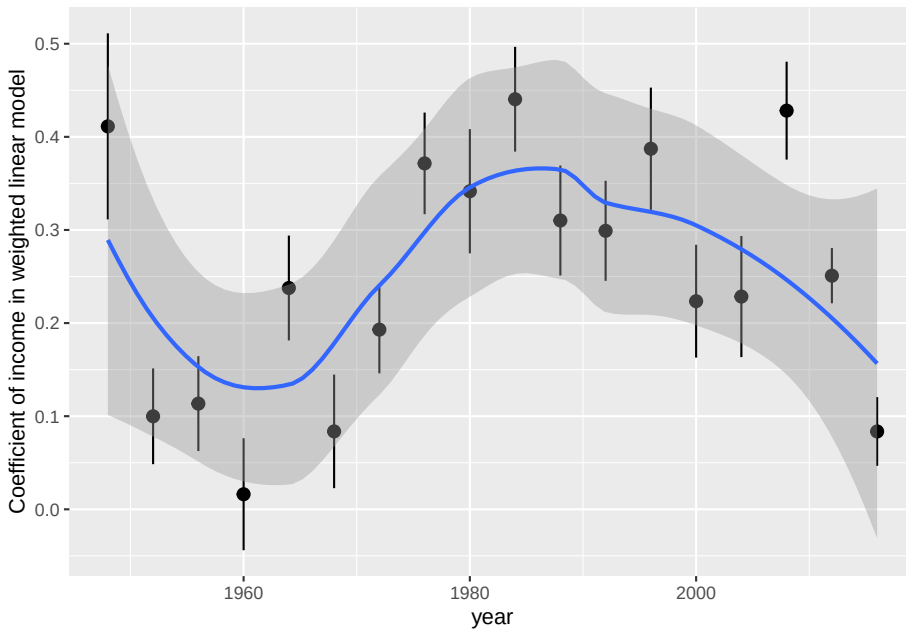
The "estimate" is the same as the weighted regression we fitted above, so it seems the function is working fine. Now we want to run the function for every Presidential election year from 1948 to 2012.

```
years = seq(1948, 2016, 4)
n = length(years)
income.by.year = data.frame(year = rep(NA, n), income.coef = rep(NA, n), income.se = re
for (J in 1:n){
  my.year = years[J]
  income.by.year[J,] = logit.ANES.subset(my.year = my.year, data = ANES.df)
}
```

We'll display the results using `ggplot()`. The nifty `geom_pointrange()` lets us add one standard error bounds. Again, we shouldn't take these too literally,

just use them to get a ballpark idea of uncertainty.

```
gg = ggplot(income.by.year, aes(x = year, y = income.coef, ymin = income.coef - income.se,
                                ymax = income.coef + income.se))
gg + geom_pointrange() + geom_smooth(method.args = list(family = "symmetric")) +
  ylab("Coefficient of income in weighted linear model")
```



The income coefficient is positive for every election, meaning richer people were more likely to vote Republican every time (though 1960 was close.) The general trend was an increase in the income coefficient from 1952 to 1984, then a leveling-off. There was a huge drop from 1948 to 1952; unfortunately we don't have data from before 1948 to know if the election was typical. Otherwise there are a couple of elections outside the confidence band: 1964 (Johnson over Goldwater) and 2008 (Obama over McCain.)

7.1.6 Less modeling, more detail

In our regressions, we treated `income` as a quantitative variable. A simpler approach would be to treat it as a factor, and simply track the weighted proportion of each income group that (two-party) voted Republican by year. Again, this is easiest to program (if inefficient) using a `for` loop.

To find weighted means, I used use `weighted.mean()` in conjunction with `summarise()` in `dplyr`. Let's first work out how to do it for the 1992 data.

```
group_by(ANES1992, income) %>% summarise(weighted.mean(Republican, w = survey.weights, na.rm = TR
```

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

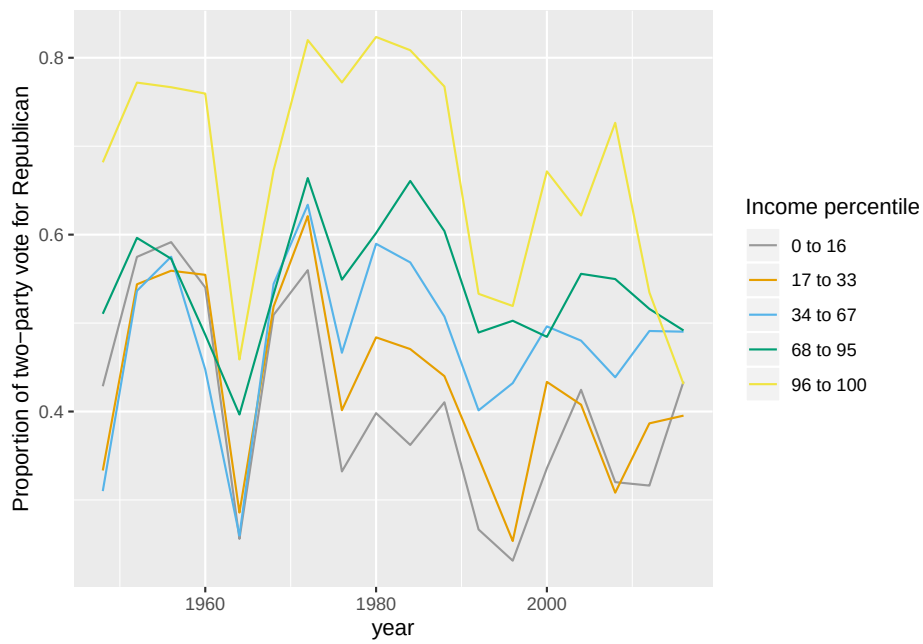
```
## income `weighted.mean(Republican, w = survey.weights, na.rm =
TRUE)`
## <dbl> <dbl>
## 1 1 0.267
## 2 2 0.348
## 3 3 0.401
## 4 4 0.489
## 5 5 0.533
```

Now we do the same thing to the bigger data set, this time grouping by both income and year, then removing the "0" income category:

```
income.prop = group_by(ANES.df, income, year) %>%
  summarise(weighted.mean(Republican, w = survey.weights, na.rm = TRUE))
names(income.prop) = c("income", "year", "prop.Republican")
income.prop = income.prop[income.prop$income > 0,]
```

Plot the results:

```
gg = ggplot(income.prop, aes(x = year, y = prop.Republican,
    group = income, color = factor(income))) + geom_line() +
  scale_color_manual(values = cb_palette,
    labels = c("0 to 16", "17 to 33", "34 to 67", "68 to 95", "96 to 100"))
gg + ylab("Proportion of two-party vote for Republican") +
  labs(color = "Income percentile")
```

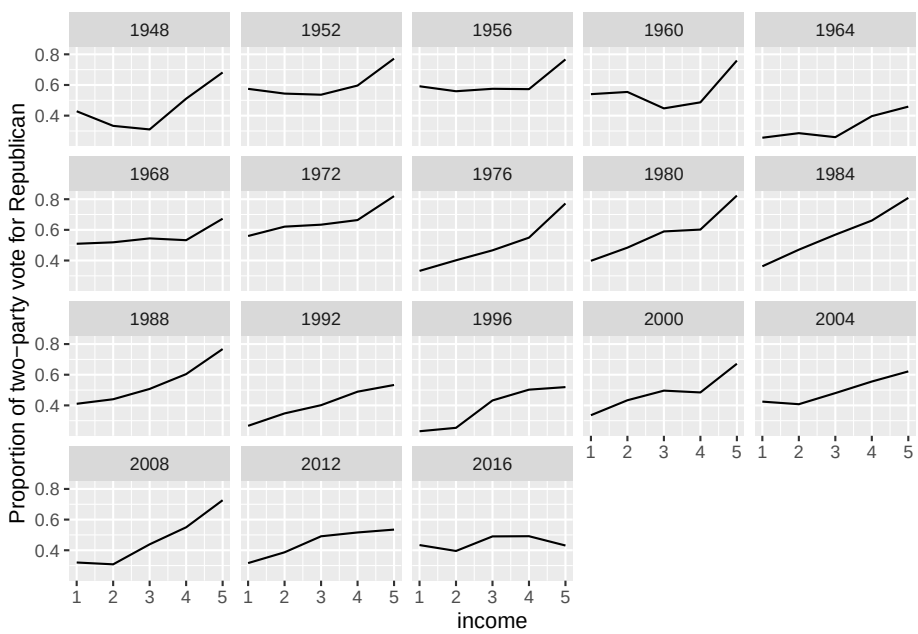


We now have a bit more detail on the trends and the aberrant results.

- The top income group is reliably the most Republican, but the bottom income group isn't always the most Democratic (although it was in the middle part of the time period.)
- In 1948 there were pretty big differences between income groups, but in the 1950s the differences between all groups except the richest were small. It's guess work whether 1948 was an aberration or whether the small income differences from 1952 to 1968 were historical unusual (though I suspect it's the latter.)
- The big coefficient for 1964 (compared to the elections before and after) might be in part an artifact of the logit scale.
- In 2008 there really was a big difference between income group, which is likely attributable to the financial crisis.

We can also draw lines to connect income groups by year. Because there are so many different years, we'll facet them rather than color them.

```
ggplot(income.prop, aes(x = income, y = prop.Republican)) + geom_line() +
  facet_wrap(~year, ncol = 5) + ylab("Proportion of two-party vote for Republican")
```



This perhaps yields less insight, but still has interesting features: notably the big magnitude of the uptick in Republicanism for the highest income group for almost every year. (It would be interesting to check if this continued to hold in 2016.)

7.1.7 Data summaries vs. models

Both data summaries (like our last plot) and models (like our logistic regressions) have their uses.

- Data summaries require fewer assumptions, and often give you a fuller picture than a model. However, they can be noisy or just too complicated to easily get a grip on.
- Models require assumptions, so in addition to being reductive, there's more potential for things to go horribly wrong. However, models can be a easy way into the data: everything gets summarized in a couple of parameters, and you can put your effort into understanding what those parameters really mean. Furthermore, complexity can easily be added to models – for example, it's easy to build a logistic regression model with multiple predictors (as we'll see in the next set of notes.)

In practice, going back and forth between models and data summaries, as we did here, is often very useful in exploratory work. Models can usefully simplify the data so you can get the big picture. Then you can look a fuller data summary and bear in results that the big picture doesn't explain.

7.2 Logistic regression with multiple predictors

Supplementary reading: Gelman & Hill pp. 86–104.

7.2.1 Arsenic in wells

The file `wells.dat` contains data on 3,020 households in Bangladesh whose wells had arsenic levels above the national drinking water standard of 50 micrograms per liter. The variables are:

- `switch`: the response: whether or not the household switched after being encouraged to do so (after high arsenic was measured.)
- `arsenic`: the arsenic concentration in hundreds of micrograms per liter.
- `dist`: the distance to the nearest safe well in meters.
- `assoc`: whether anyone in the household is in a community association.
- `educ`: the years of education of the head of the household.

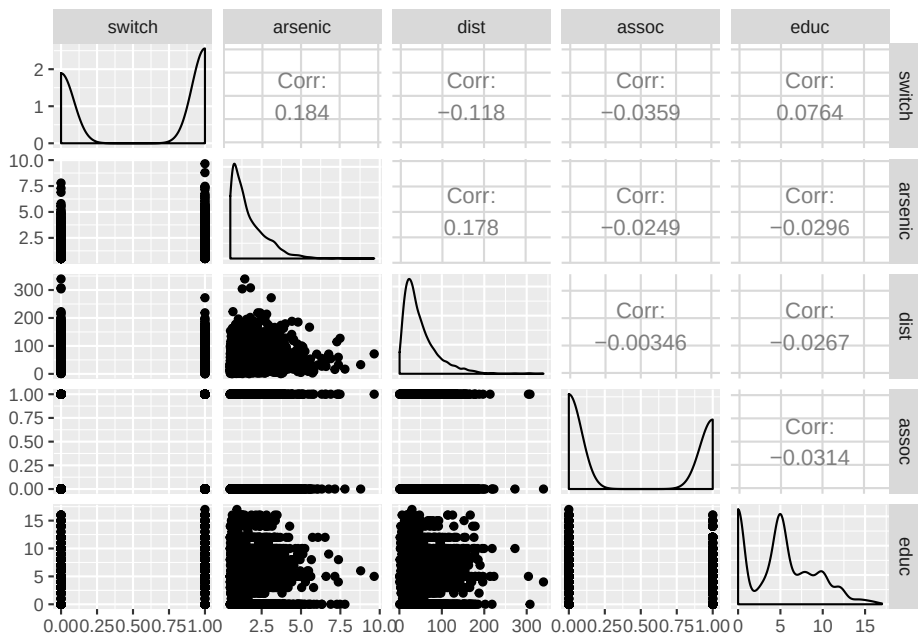
We load at the data and look at the numerical summary and a pairs plot:

```r
library(GGally)
wells = read.table("wells.dat")
summary(wells)
```

```
## switch arsenic dist assoc
## Min.  :0.0000 Min.  :0.510 Min.  : 0.387 Min.  :0.0000
## 1st Qu.:0.0000 1st Qu.:0.820 1st Qu.: 21.117 1st Qu.:0.0000
## Median :1.0000 Median :1.300 Median : 36.761 Median :0.0000
```

```
## Mean :0.5752 Mean :1.657 Mean : 48.332 Mean :0.4228
## 3rd Qu.:1.0000 3rd Qu.:2.200 3rd Qu.: 64.041 3rd Qu.:1.0000
## Max.  :1.0000 Max.  :9.650 Max.  :339.531 Max.  :1.0000
## educ
## Min.  : 0.000
## 1st Qu.: 0.000
## Median : 5.000
## Mean : 4.828
## 3rd Qu.: 8.000
## Max.  :17.000
```

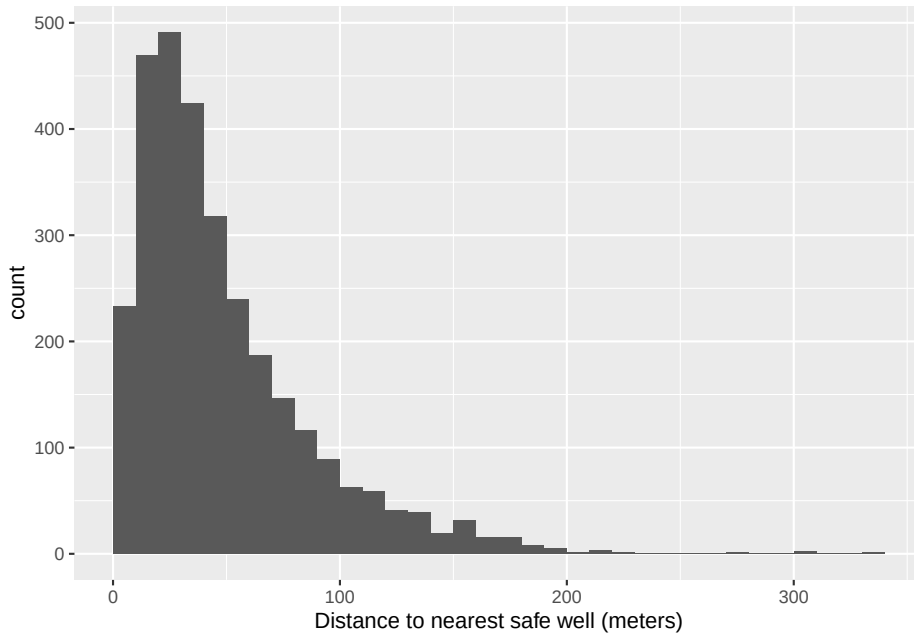
```
ggpairs(wells)
```



As we might expect, distance and education level are positively associated with switching, while arsenic level is negatively associated. Surprisingly, the correlation between belong to an association and switching is negative, though the relationship is very weak.

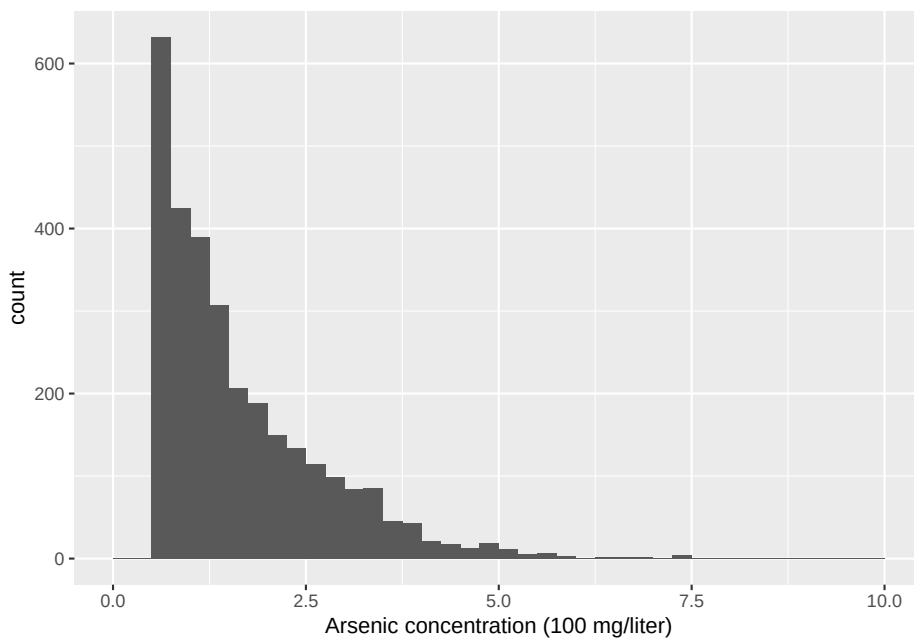
Distance and arsenic levels should be the best predictors, so let's look more closely at the distributions of those variables.

```
gg = ggplot(wells, aes(x = dist)) + geom_histogram(breaks = seq(0, 340, 10))
gg + xlab("Distance to nearest safe well (meters)")
```



We see the distribution of distance peaks around 40–60 meters, then is strongly right-skewed. We should keep the possibility of a transformation in mind.

```
ggplot(wells, aes(x = arsenic)) + geom_histogram(breaks = seq(0, 10, 0.25)) +
  xlab("Arsenic concentration (100 mg/liter)")
```

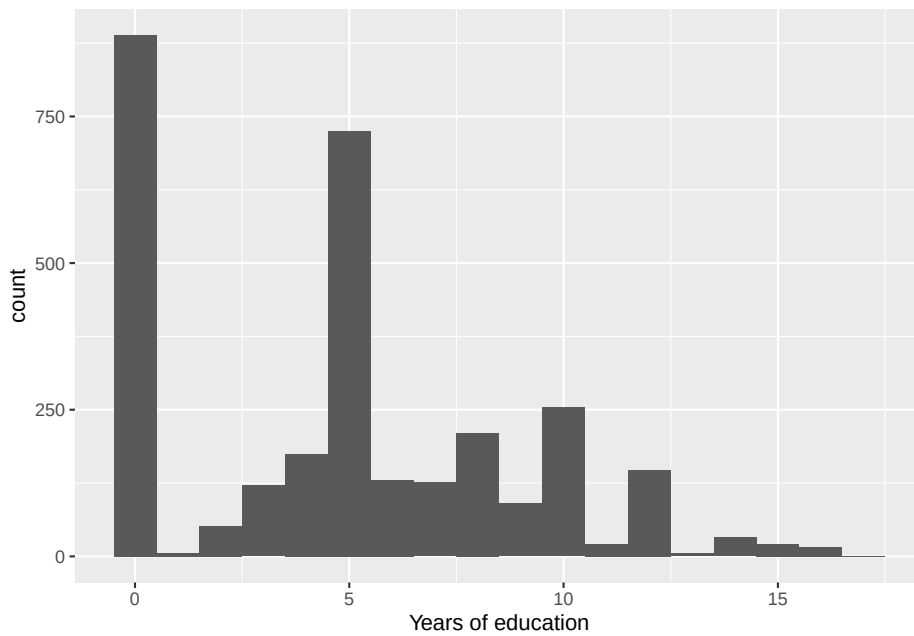


There are no observations with arsenic below 0.5, because these are considered "safe" and the households don't get advised to switch wells. Otherwise, the distribution is against strongly right-skewed, suggesting a transformation.

Note that the scales of `dist` and `arsenic` are quite different. Gelman and Hill suggest standardizing the data for interpretability, so that predictors are on approximately the same scale. That's not a bad idea but we won't bother.

Here's education:

```
ggplot(wells, aes(x = educ)) + geom_histogram(breaks = -0.5:17.5) + xlab("Years of education")
```



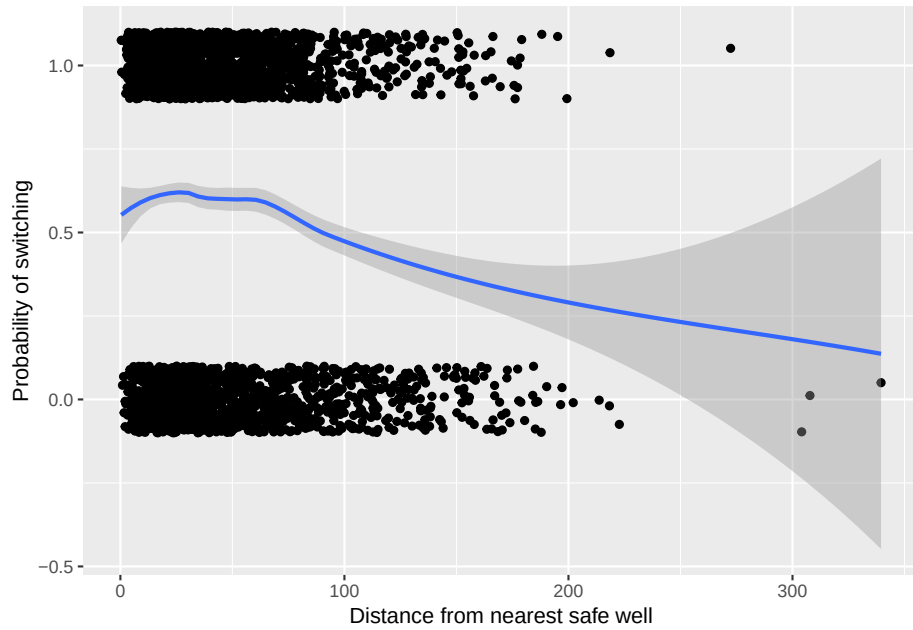
The distribution is weird. 0 (no education) and 5 (primary school only) are the magic numbers.

7.2.2 Starting off simple

One way of approaching EDA is to throw everything into your model and then get rid of terms you don't need. Another strategy is to start with simple models and gradually build up to complex models. Let's take this second approach for now.

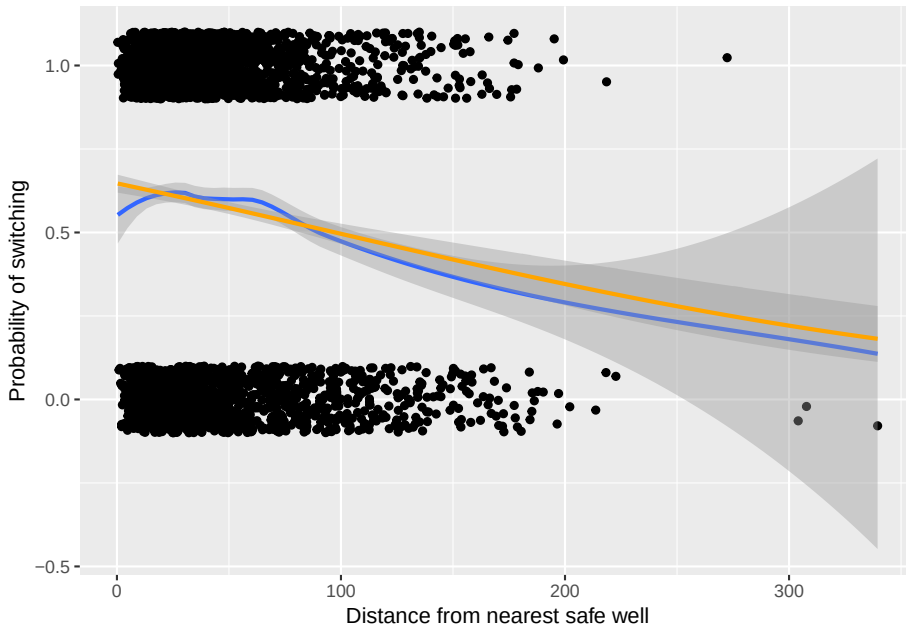
We'll start by predicting switching, using distance as the only response. There's no reason why we can't try loess:

```
ggplot(wells, aes(x = dist, y = switch)) + geom_jitter(width = 0, height = 0.1) +
  geom_smooth(method = "loess") +
  xlab("Distance from nearest safe well") + ylab("Probability of switching")
```



We can interpret the fit as the probability of switching wells, given the distance to the nearest safe well. There's a bump for very low distances (which is probably just noise) and then a decline. We could play around with the smoothing parameters to get something more pleasant-looking. Instead, we'll fit a logistic regression to guarantee a decreasing relationship.

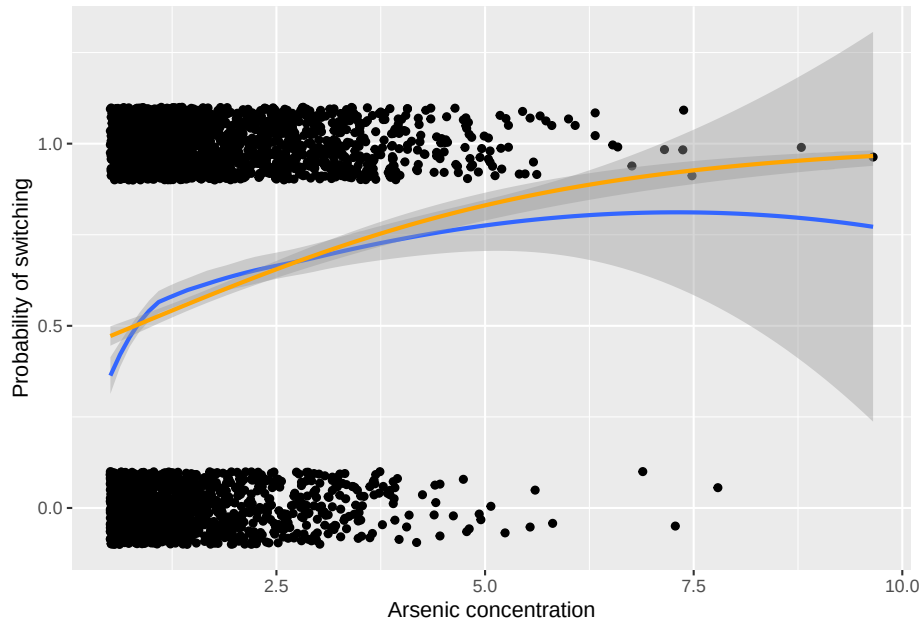
```
gg = ggplot(wells, aes(x = dist, y = switch)) + geom_jitter(width = 0, height = 0.1) +
  geom_smooth(method = "loess")
gg + geom_smooth(method = "glm", method.args = list(family = "binomial"), color = "oran
  xlab("Distance from nearest safe well") + ylab("Probability of switching")
```



The parametric method imposes a functional form on the data. This aids in interpretability: the fit can now be described in a couple of parameters, and the weird bump at the beginning is gone. The cost is some lack of fit: perhaps the probability should decrease more quickly as the distance gets beyond about 60 meters (remember that water is heavy and it's tiring to carry buckets back and forth over long distances.)

Let's try the same kind of approach using arsenic concentration as the sole predictor.

```
gg = ggplot(wells, aes(x = arsenic, y = switch)) + geom_jitter(width = 0, height = 0.1) +
  geom_smooth(method = "loess")
gg + geom_smooth(method = "glm", method.args = list(family = "binomial"), color = "orange") +
  xlab("Arsenic concentration") + ylab("Probability of switching")
```

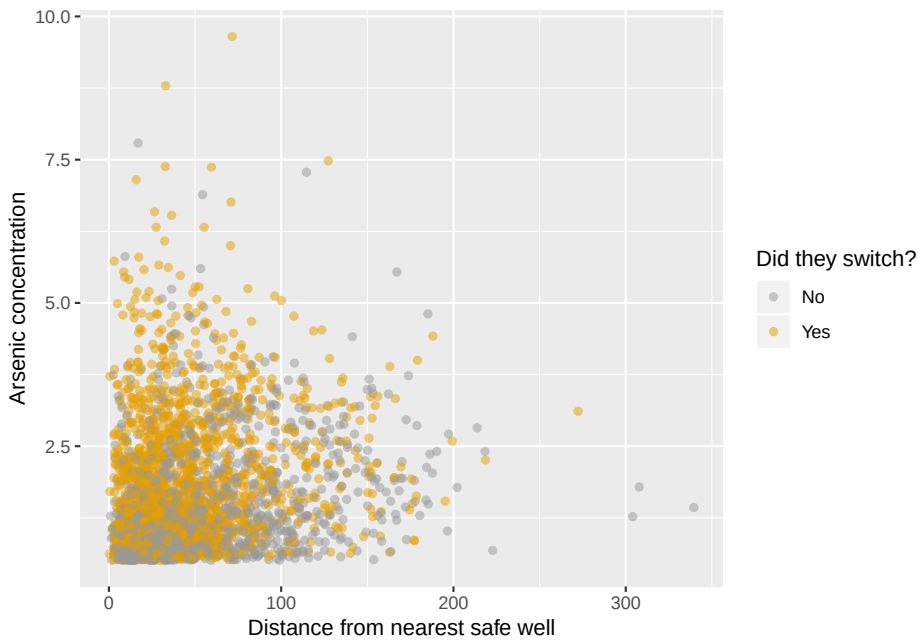


The fits diverge at arsenic levels above about 5. The seems to be because of a couple of extremely high arsenic observations which led to switching. Note however that not much of the data has arsenic above 5, so it remains to be seen how much this matters.

7.2.3 Two predictors

We now want to know how the chance of switching depends on distance and arsenic simulataneously. Before fitting a model, we plot the data to get a feel for it. We'll use color to distinguish between households that switch and households that don't.

```
ggplot(wells, aes(x = dist, y = arsenic, color = factor(switch))) + geom_point(alpha =
  xlab("Distance from nearest safe well") + ylab("Arsenic concentration") +
  labs(color = "Did they switch?") +
  scale_color_manual(values = cb_palette, labels = c("No", "Yes"))
```



The `alpha = 0.5` makes the points slightly transparent, which can help visually when you have a lot of data. Still, it's hard to work out how switching depends on the other two variables from this graph; that's why we're fitting a model. The main thing to take home from the graph is the lack of data in the top right: we have no observations at all with both distance above 200 and arsenic above 4, so we should not try to generalize to this region.

Now we'll fit a logistic regression using both distance and arsenic as predictors. We first fit an additive model, i.e. with no interaction.

```
switch.logit = glm(switch ~ dist + arsenic, family = "binomial", data = wells)
summary(switch.logit)
```

```
##
## Call:
## glm(formula = switch ~ dist + arsenic, family = "binomial",
data = wells)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -2.6351 -1.2139 0.7786 1.0702 1.7085
##
## Coefficients:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.002749 0.079448 0.035 0.972
## dist -0.008966 0.001043 -8.593 <2e-16 ***
```

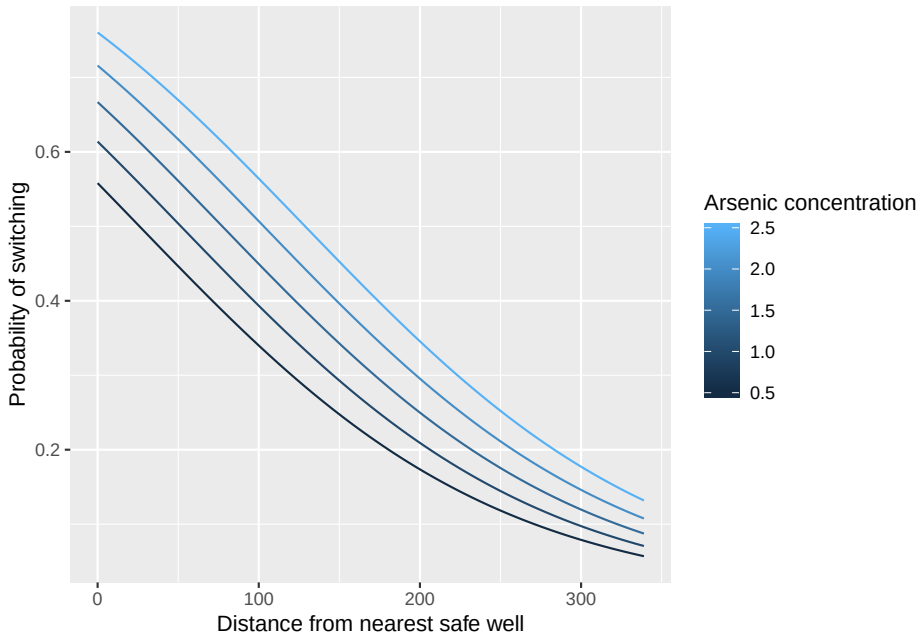
```
## arsenic 0.460775 0.041385 11.134 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 4118.1 on 3019 degrees of freedom
## Residual deviance: 3930.7 on 3017 degrees of freedom
## AIC: 3936.7
##
## Number of Fisher Scoring iterations: 4
```

As always, the coefficients are the most important things here. According to the model,

$$\text{logit}[P(\text{switch}|\text{dist, arsenic})] = 0.003 - 0.00897 \times \text{dist} + 0.4608 \times \text{arsenic}.$$

As in the continuous case, we visualize the fit by drawing multiple curves representing different values of one of the predictors. Let's display the switching probability as a function of distance for a few values of arsenic. Note that when we use the `predict()` function, we specify `type = "response"` to get the probabilities and not their logits.

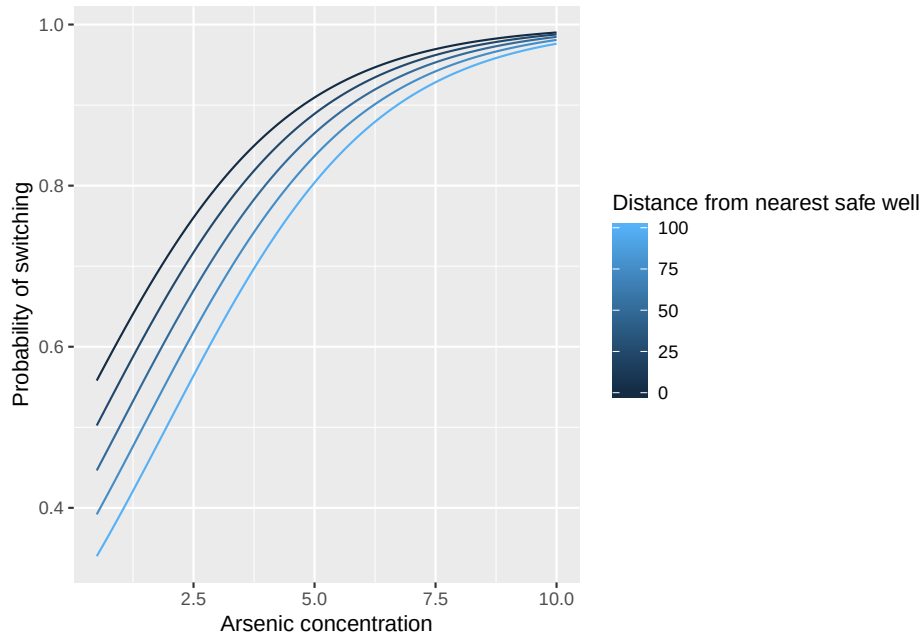
```
dist.df = expand.grid(dist = 0:339, arsenic = seq(0.5, 2.5, 0.5))
dist.pred = predict(switch.logit, type = "response", newdata = dist.df)
dist.pred.df = data.frame(dist.df, switch.prob = as.vector(dist.pred))
ggplot(dist.pred.df, aes(x = dist, y = switch.prob,
                         group = arsenic, color = arsenic)) + geom_line() +
  xlab("Distance from nearest safe well") + ylab("Probability of switching") +
  labs(color = "Arsenic concentration")
```



The median arsenic level is 1.3. If arsenic is near the median, the probability of switching declines from 60-something percent if you're right by a safe well to less than 10% if the nearest safe well is hundreds of meters away.

Now find prediction curves for arsenic for a few different distances:

```
arsenic.df = expand.grid(arsenic = seq(0.5, 10, 0.01), dist = seq(0, 100, 25))
arsenic.pred = predict(switch.logit, type = "response", newdata = arsenic.df)
arsenic.pred.df = data.frame(arsenic.df, switch.prob = as.vector(arsenic.pred))
ggplot(arsenic.pred.df, aes(x = arsenic, y = switch.prob,
                            group = dist, color = dist)) + geom_line() +
  xlab("Arsenic concentration") + ylab("Probability of switching") +
  labs(color = "Distance from nearest safe well")
```



The median distance to a safe well is about 37 meters. At that distance, if the arsenic level is only just over the safety threshold, it's about 50–50 whether a household switched. On the other hand, at the highest levels of arsenic, households will almost certainly switch even if the nearest safe well is quite far.

7.2.4 Adding an interaction

We now add an interaction term between distance and arsenic.

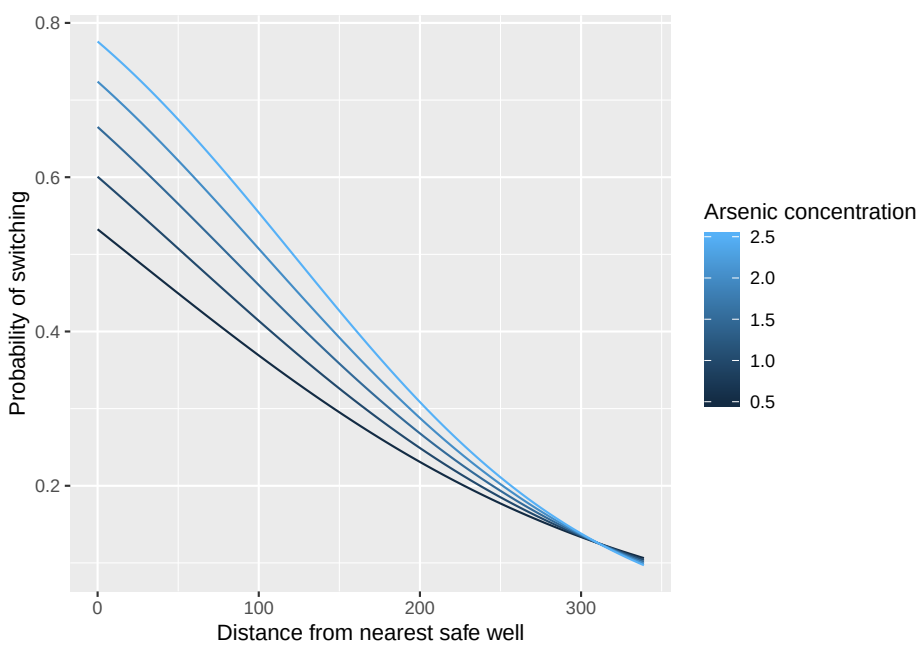
```
wells = read.table("wells.dat")
switch.int = glm(switch ~ dist * arsenic, family = "binomial", data = wells)
summary(switch.int)
```

```
##
## Call:
## glm(formula = switch ~ dist * arsenic, family = "binomial",
data = wells)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -2.7823 -1.2004 0.7696 1.0816 1.8476
##
## Coefficients:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.147868 0.117538 -1.258 0.20838
## dist -0.005772 0.002092 -2.759 0.00579 **
```

```
## arsenic 0.555977 0.069319 8.021 1.05e-15 ***
## dist:arsenic -0.001789 0.001023 -1.748 0.08040 .
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 4118.1 on 3019 degrees of freedom
## Residual deviance: 3927.6 on 3016 degrees of freedom
## AIC: 3935.6
##
## Number of Fisher Scoring iterations: 4
```

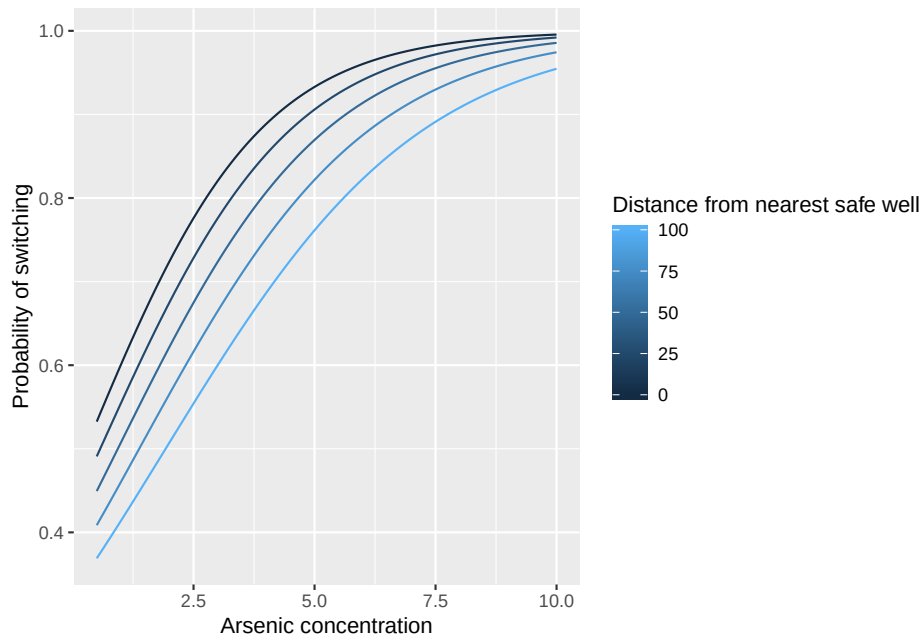
The numbers are a bit hard to interpret. For example, the sign of the interaction term is negative, but it's hard to know exactly what this means (especially since the signs for distance and arsenic go in different directions.) Let's just plot curves.

```
dist.df = expand.grid(dist = 0:339, arsenic = seq(0.5, 2.5, 0.5))
dist.int.pred = predict(switch.int, type = "response", newdata = dist.df)
dist.int.pred.df = data.frame(dist.df, switch.prob = as.vector(dist.int.pred))
ggplot(dist.int.pred.df, aes(x = dist, y = switch.prob,
                             group = arsenic, color = arsenic)) + geom_line() +
  xlab("Distance from nearest safe well") + ylab("Probability of switching") +
  labs(color = "Arsenic concentration")
```



The interaction brings the curves together as distance increases. If the nearest safe well is close, it makes a big difference whether the arsenic concentration is just over the limit or much bigger. If the nearest safe well is far, it makes little difference: people are unlikely to switch no matter the concentration. The curves meet up beyond 300 meters, though we only have three observations where the distance exceeds 300 meters.

```
arsenic.df = expand.grid(arsenic = seq(0.5, 10, 0.01), dist = seq(0, 100, 25))
arsenic.int.pred = predict(switch.int, type = "response", newdata = arsenic.df)
arsenic.int.pred.df = data.frame(arsenic.df, switch.prob = as.vector(arsenic.int.pred))
ggplot(arsenic.int.pred.df, aes(x = arsenic, y = switch.prob,
                                group = dist, color = dist)) + geom_line() +
  xlab("Arsenic concentration") + ylab("Probability of switching") +
  labs(color = "Distance from nearest safe well")
```



The curves actually get further apart at first as arsenic increases (up to a point.) That is, the curves for short distances rise quite quickly as arsenic increases, while the curves for long distances rise more slowly. Eventually, for exceptionally high levels of arsenic, the curves come together (because they can't go any higher than 1.)

7.2.5 Lots of predictors

Distance and arsenic are both good predictors and there's no reason to think there shouldn't be an interaction, so keep the interaction term. Now let's add `assoc` and `educ` to the model and see if the fit makes sense.

```r
summary(glm(switch ~ dist * arsenic + assoc + educ, family = "binomial", data = wells))
```

```
##
## Call:
## glm(formula = switch ~ dist * arsenic + assoc + educ, family
= "binomial",
## data = wells)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -2.7303 -1.1892 0.7444 1.0675 1.6987
##
## Coefficients:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.291120 0.131427 -2.215 0.02675 *
## dist -0.006081 0.002093 -2.905 0.00367 **
## arsenic 0.553238 0.069542 7.955 1.79e-15 ***
## assoc -0.123188 0.076977 -1.600 0.10953
## educ 0.041948 0.009594 4.372 1.23e-05 ***
## dist:arsenic -0.001612 0.001022 -1.577 0.11482
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 4118.1 on 3019 degrees of freedom
## Residual deviance: 3905.4 on 3014 degrees of freedom
## AIC: 3917.4
##
## Number of Fisher Scoring iterations: 4
```

Years of education has a positive coefficient, which makes sense. We can keep that in the model.

Association membership has a *negative* coefficient, which doesn't make causal sense: it would be strange for association membership to make you *less* likely to switch wells, unless you were a member of the Association for Arsenic Being Good for You. If our sole goal was prediction, we could do a careful cross-validation to see if association membership did help predict more accurately. Since we're doing EDA, we can just drop the association term without thinking too hard about it.

```r
summary(glm(switch ~ dist * arsenic + educ, family = "binomial", data = wells))
```

```
##
## Call:
## glm(formula = switch ~ dist * arsenic + educ, family =
```

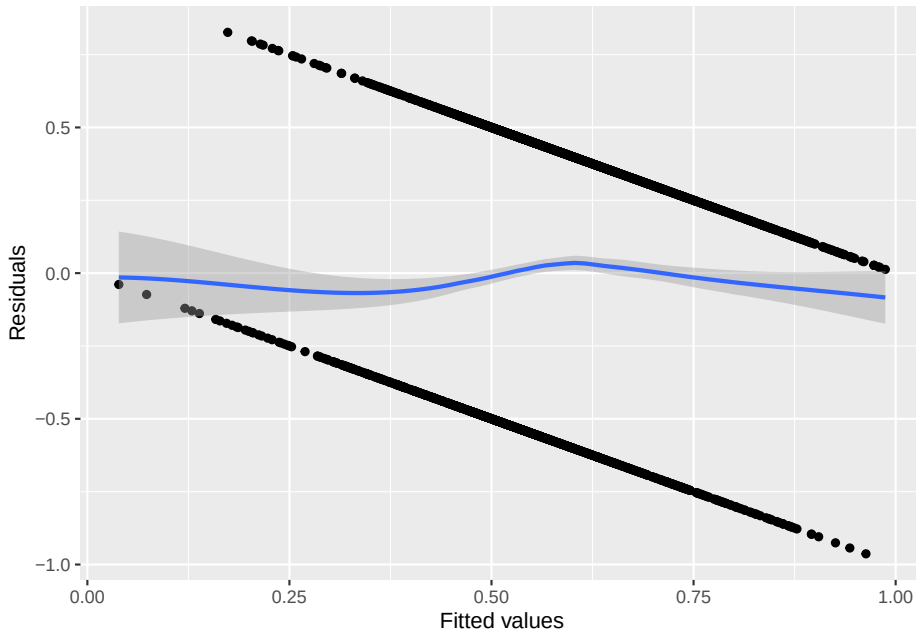
```
"binomial",
## data = wells)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -2.7149 -1.1886 0.7478 1.0689 1.7223
##
## Coefficients:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.349044 0.126360 -2.762 0.00574 **
## dist -0.006047 0.002095 -2.886 0.00390 **
## arsenic 0.555367 0.069531 7.987 1.38e-15 ***
## educ 0.042306 0.009581 4.415 1.01e-05 ***
## dist:arsenic -0.001629 0.001023 -1.592 0.11145
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 4118.1 on 3019 degrees of freedom
## Residual deviance: 3907.9 on 3015 degrees of freedom
## AIC: 3917.9
##
## Number of Fisher Scoring iterations: 4
```

We now add the remaining two-way interactions. In general, if you only have a few terms, you might as well include all the two-way interactions unless you have a good reason not to.

```
switch.model = glm(switch ~ dist + arsenic + educ + dist:arsenic + dist:educ + arsenic
```

At this point our model is too complicated for it to be worth trying to interpret the numbers. (If we had centered the variables, we might be able to do some numerical interpretation.) Let's extract the fitted values and residuals, and plot the latter against the former. Note that the default of `augment()` is to extract the deviance residuals instead of the response residuals, so we won't use `augment()`.

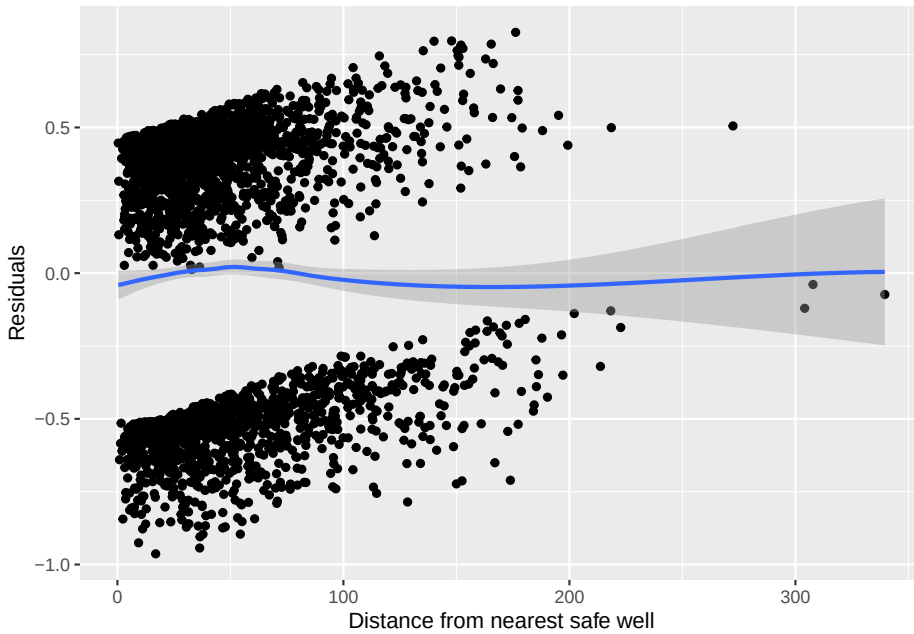
```
switch.model.df = wells
switch.model.df$.fitted = fitted.values(switch.model)
switch.model.df$.resid = residuals(switch.model, type="response")
ggplot(switch.model.df, aes(x = .fitted, y = .resid)) + geom_point() +
  geom_smooth(method = "loess", method.args = list(degree = 1)) +
  xlab("Fitted values") + ylab("Residuals")
```



The points on the plot will always fall exactly on two lines, so look at the smooth instead. We see there's a little bit of waviness. This is fairly typical for logistic regression: there's no reason why the relationship between probability and the predictors should have that exact functional form. But the improvement in fit you get from fitting a nonparametric model is fairly small.

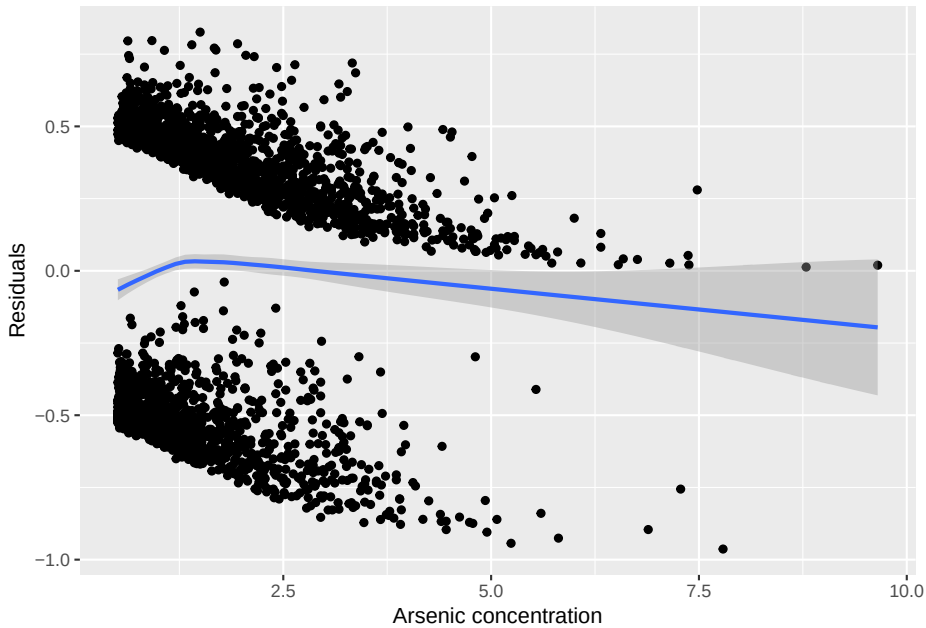
Now plot the residuals against distance.

```
ggplot(switch.model.df, aes(x = dist, y = .resid)) + geom_point() +
  geom_smooth(method = "loess", method.args = list(degree = 1)) +
  xlab("Distance from nearest safe well") + ylab("Residuals")
```



There's a wiggle, but this is basically fine – the zero line is enclosed in the confidence band. Now do the same plot with arsenic on the x-axis:

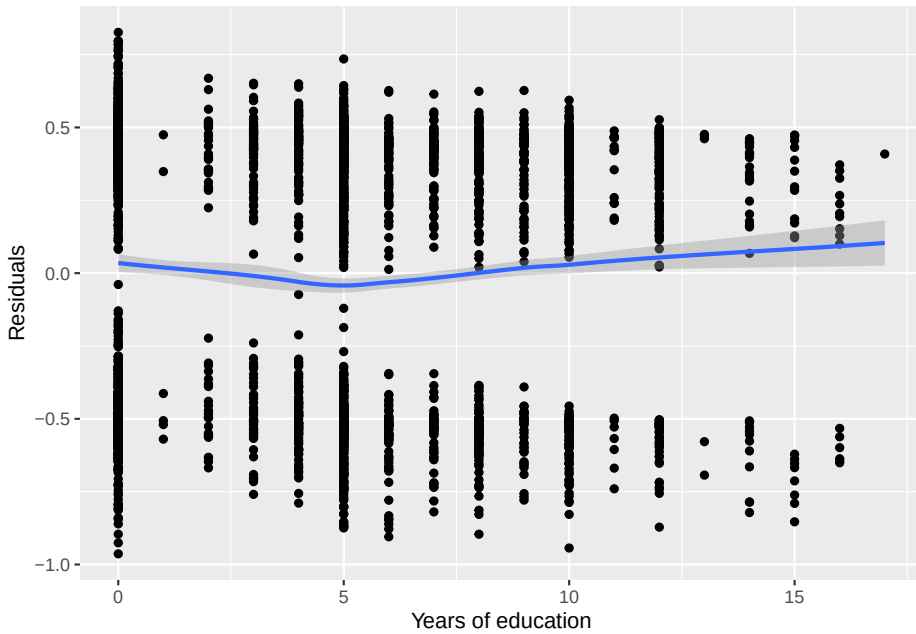
```
ggplot(switch.model.df, aes(x = arsenic, y = .resid)) + geom_point() +
  geom_smooth(method = "loess", method.args = list(degree = 1)) +
  xlab("Arsenic concentration") + ylab("Residuals")
```



This is much worse. The curve is too low, then rises too high, then declines again. In fact, the extreme left hand side is the biggest problem, since the lack is fit is real and the data is dense there. The negative average residual there means that for arsenic levels just over the threshold, the probability of switching is *overestimated* (negative average residuals occur when there are more zeroes in the response than the model expects.) When the residual curve becomes positive for arsenic between 1 and 2, that means the model *underestimates* the probability of switch for those arsenic values.

Education isn't that interesting since the effect is much smaller, but let's do it for completeness.

```
ggplot(switch.model.df, aes(x = educ, y = .resid)) + geom_point() +
  geom_smooth(method = "loess", method.args = list(degree = 1)) +
  xlab("Years of education") + ylab("Residuals")
```



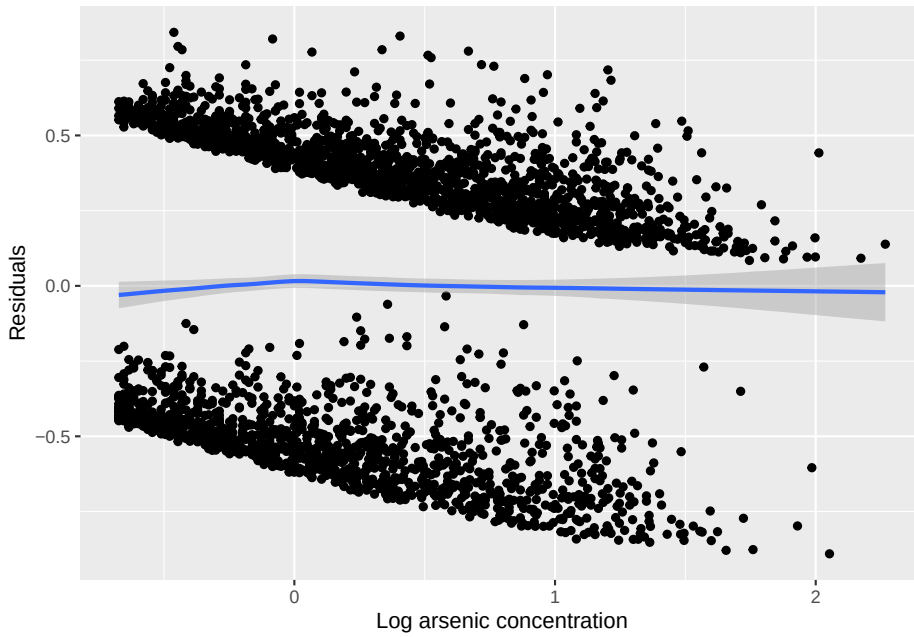
Again, the fit is imperfect.

Remember when we said we might need to transform? The last couple of graphs suggest that while a transformation of distance is unnecessary, a transformation of arsenic is. A transformation of education would probably help too, but since the variable is weird and not that important we won't bother. We'll take the log arsenic and refit the model. It's up to you what kind of log you use; it doesn't matter much in my opinion.

```
wells$log.arsenic = log(wells$arsenic)
log.arsenic.model = glm(switch ~ dist + log.arsenic + educ + dist:log.arsenic +
    dist:educ + log.arsenic:educ, family = "binomial", data = wells)
log.arsenic.model.df = wells
log.arsenic.model.df$.fitted = fitted.values(log.arsenic.model)
log.arsenic.model.df$.resid = residuals(log.arsenic.model, type="response")
```

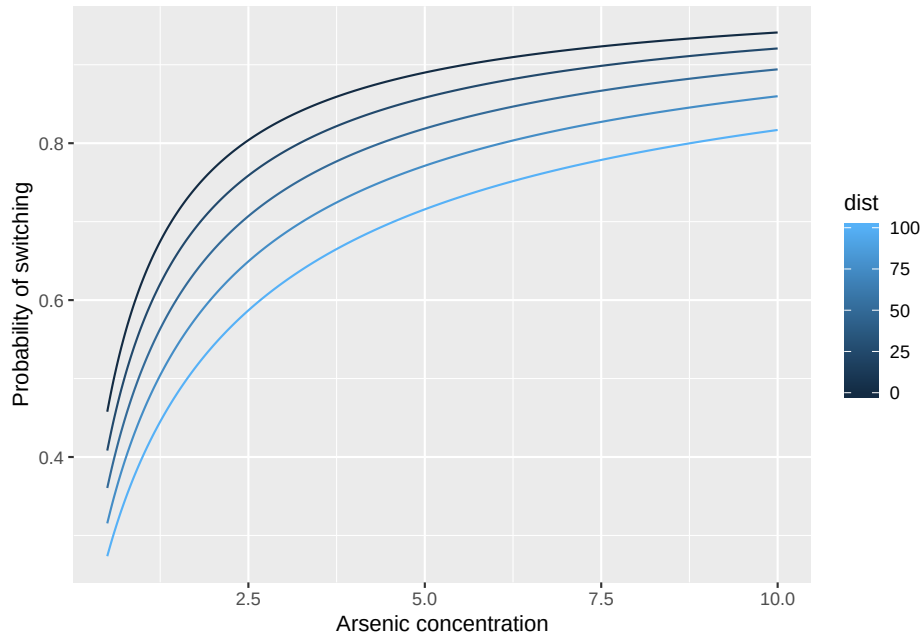
Now plot the residuals of this new model against log arsenic:

```
ggplot(log.arsenic.model.df, aes(x = log.arsenic, y = .resid)) + geom_point() +
  geom_smooth(method = "loess", method.args = list(degree = 1)) +
  xlab("Log arsenic concentration") + ylab("Residuals")
```



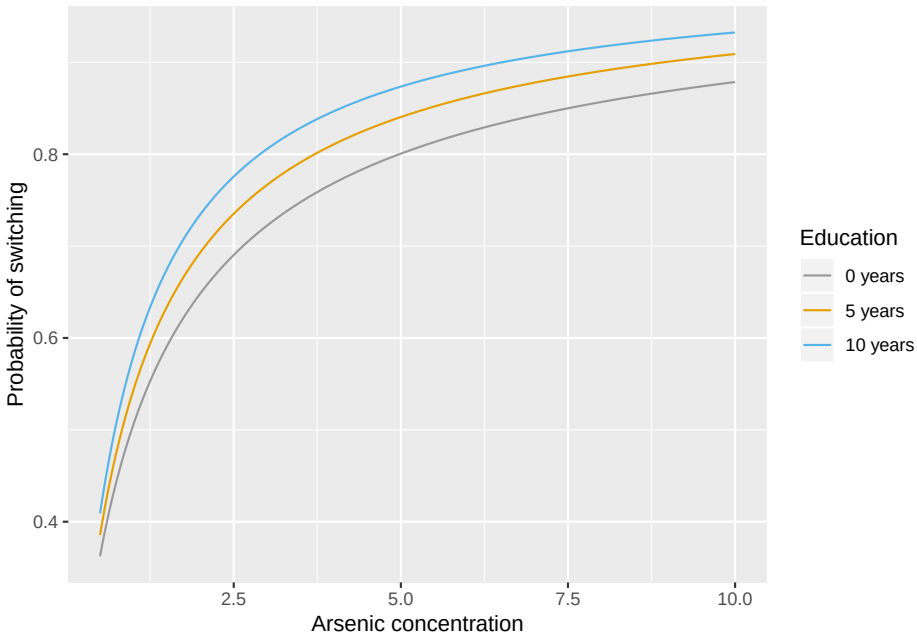
It's a lot better. I'm fairly happy with the model at this point. Let's take a look at the fit. Fix education at its median, and draw arsenic curves for several distances:

```
switch.grid = expand.grid(arsenic = seq(0.5, 10, 0.01), dist = seq(0, 100, 25), educ = 5)
switch.grid$log.arsenic = log(switch.grid$arsenic)
log.arsenic.pred = predict(log.arsenic.model, newdata = switch.grid, type = "response")
log.arsenic.grid = data.frame(switch.grid, switch.prob = as.vector(log.arsenic.pred))
ggplot(log.arsenic.grid, aes(x = arsenic, y = switch.prob,
                             group = dist, color = dist)) + geom_line() +
  xlab("Arsenic concentration") + ylab("Probability of switching")
```



There are 3! ways you can assign the three variables to x-axis, conditioning variable, and fixed. You probably get the idea at this point, though, so let's just do one more:

```
switch.grid2 = expand.grid(arsenic = seq(0.5, 10, 0.01), dist = median(wells$dist), edu
switch.grid2$log.arsenic = log(switch.grid2$arsenic)
log.arsenic.pred2 = predict(log.arsenic.model, newdata = switch.grid2, type = "response
log.arsenic.grid2 = data.frame(switch.grid2, switch.prob = as.vector(log.arsenic.pred2)
ggplot(log.arsenic.grid2, aes(x = arsenic, y = switch.prob,
                              group = educ, color = factor(educ))) +
  geom_line() + xlab("Arsenic concentration") + ylab("Probability of switching") +
  labs(color = "Education") + scale_color_manual(values = cb_palette,
    labels = c("0 years", "5 years", "10 years"))
```



7.2.6 Improving the model

If you really care about prediction but still want to be able to interpret the model, try a nonparametric model such as a GAM or loess. To overgeneralize, GAM is often better/easier when you have lots of predictors, while loess is often better/easier when you have complex interactions.

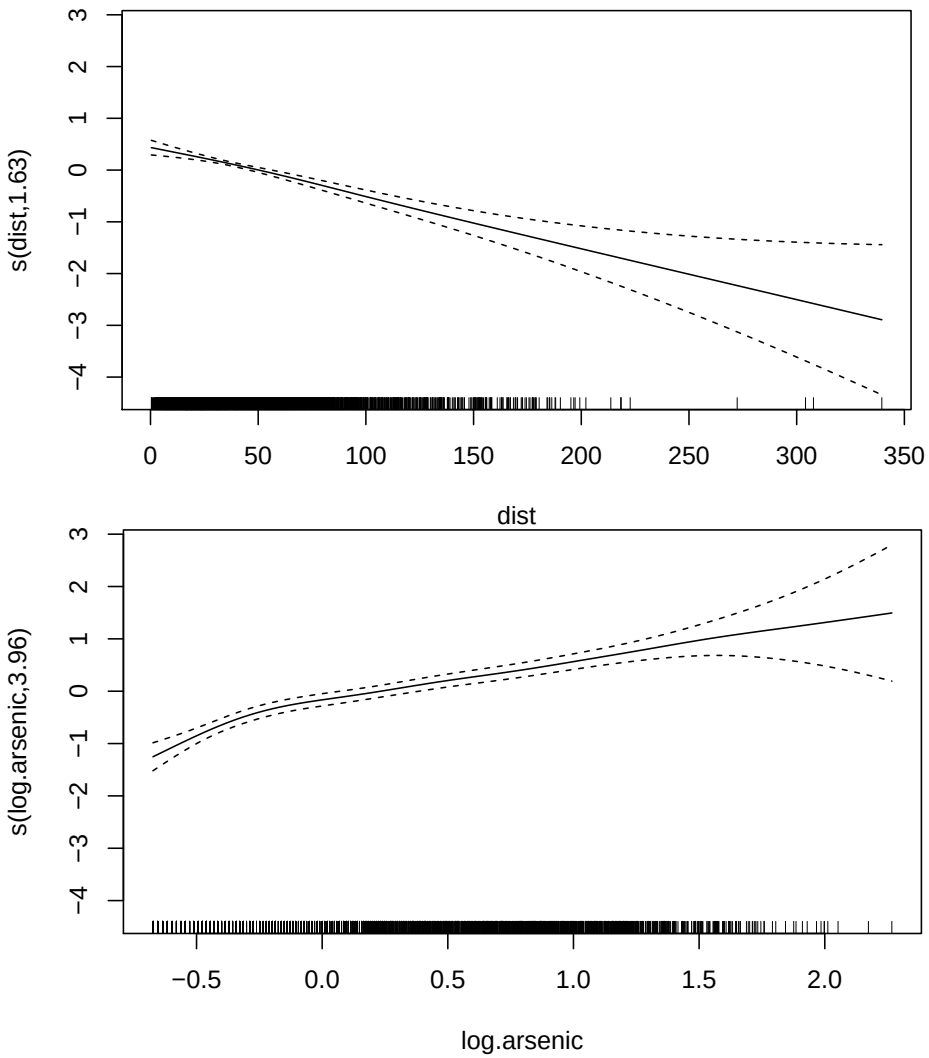
If you really, really care about predictions then you can use machine learning techniques. The improvement in prediction is usually quite small, however, and you'll lose a lot or all the interpretability.

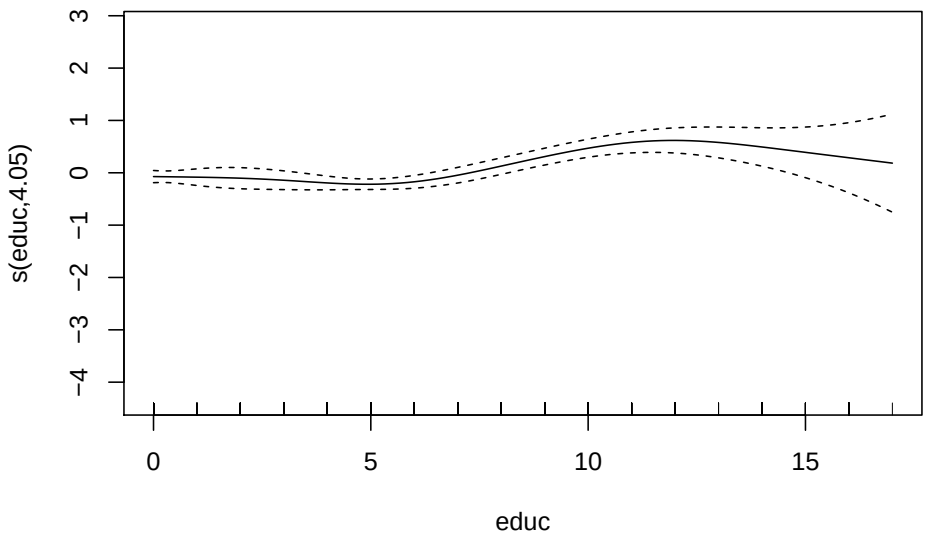
Try fitting a GAM. Note that for models that are not fundamentally least-squares based (e.g. almost anything non-Gaussian), using REML (restricted maximum likelihood) reduces the risk of overfitting.

```
library(mgcv)
switch.gam = gam(switch ~ s(dist) + s(log.arsenic) + s(educ),
                 family = "binomial", data = wells, method = "REML")
```

Plot the partial response function:

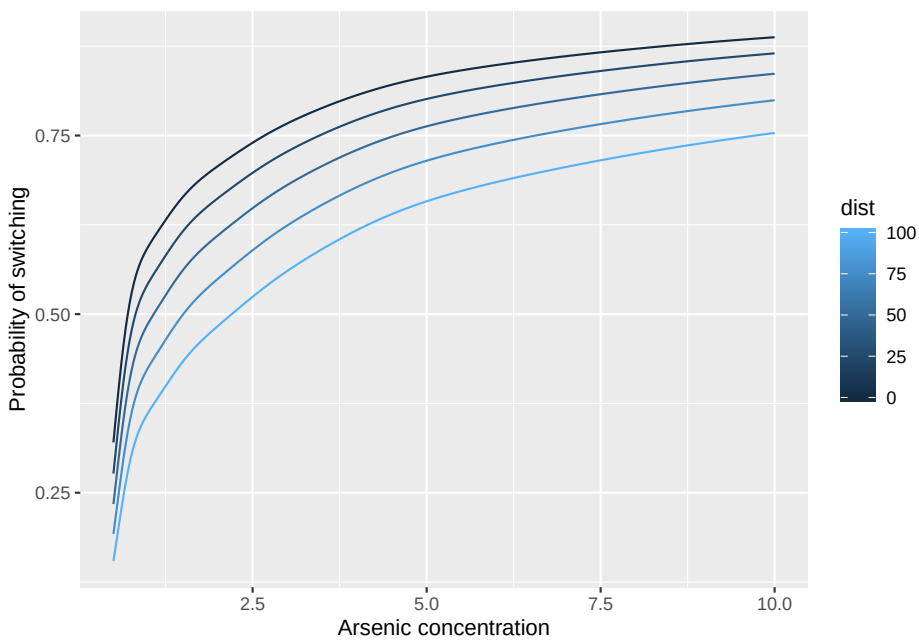
```r
plot(switch.gam)
```





Plot the results:

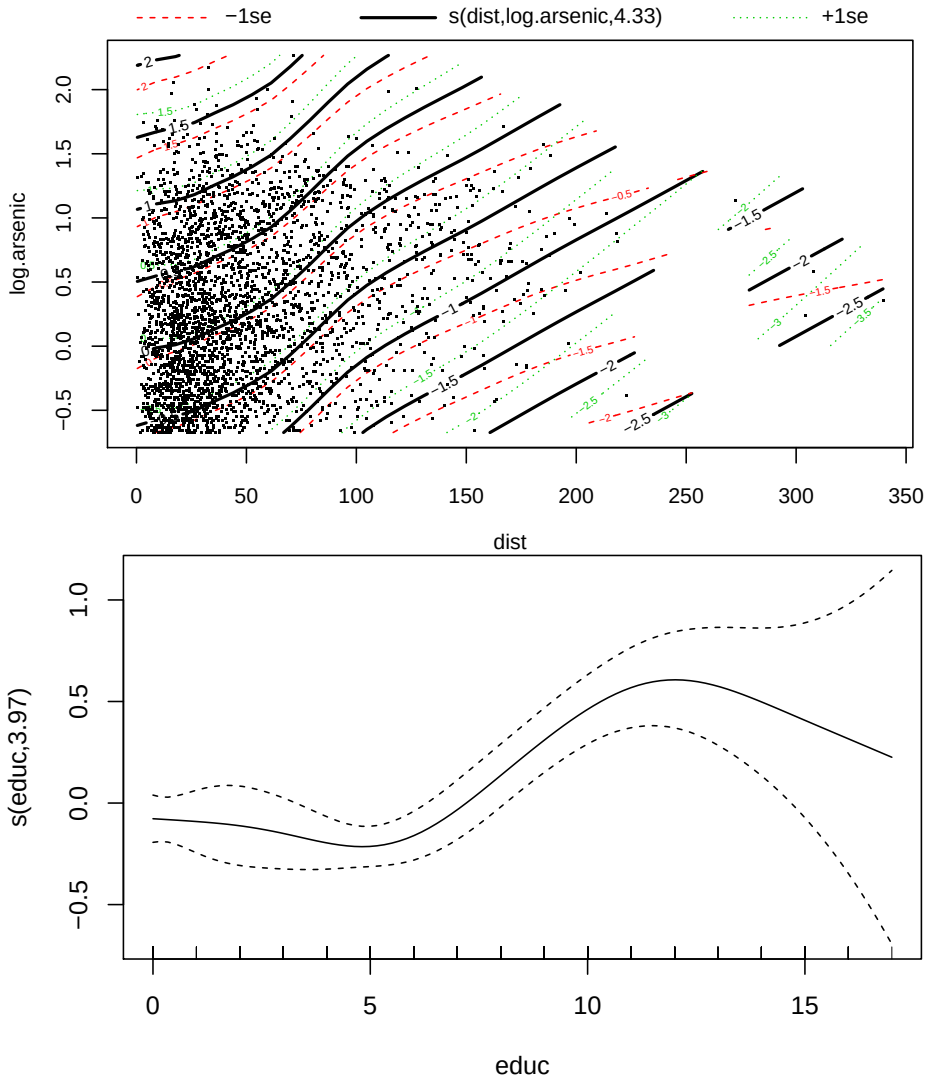
```
gam.arsenic.pred = predict(switch.gam, newdata = switch.grid, type = "response")
gam.arsenic.grid = data.frame(switch.grid, switch.prob = as.vector(gam.arsenic.pred))
ggplot(gam.arsenic.grid, aes(x = arsenic, y = switch.prob,
                             group = dist, color = dist)) + geom_line() +
  xlab("Arsenic concentration") + ylab("Probability of switching")
```



Does an interaction work?

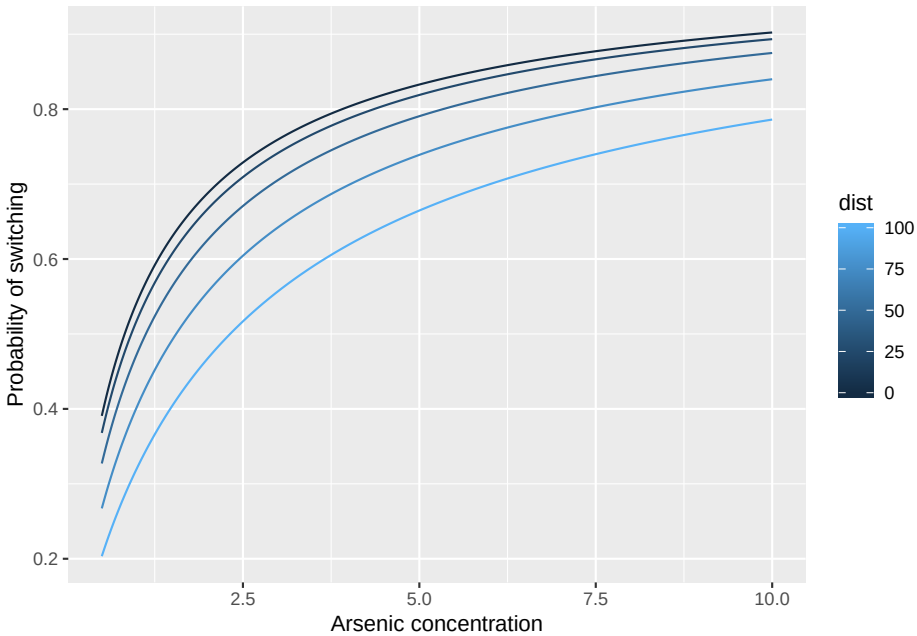
```
switch.gam2 = gam(switch ~ s(dist, log.arsenic) + s(educ),
                  family = "binomial", data = wells, method = "REML")
```

```
plot(switch.gam2)
```

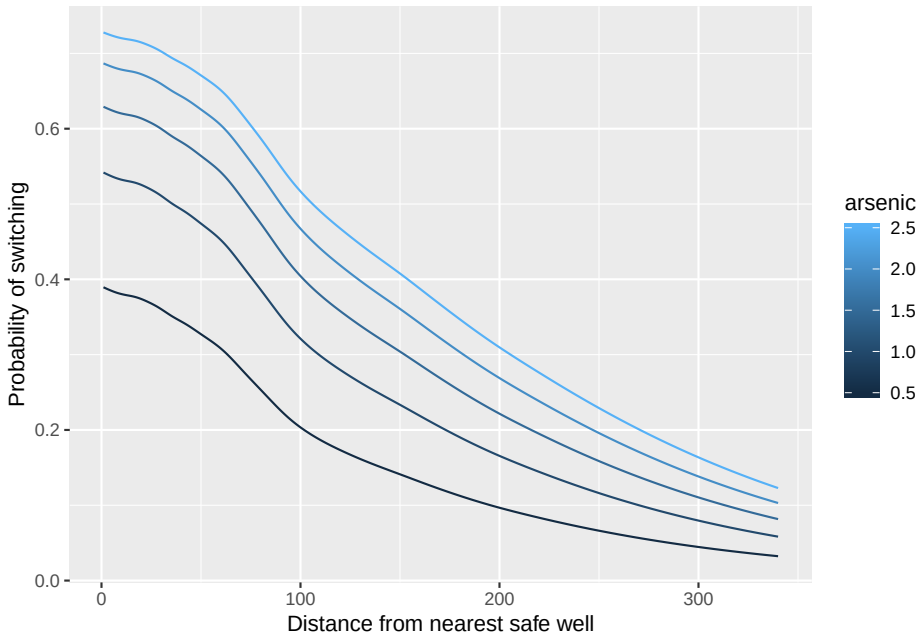


Plot on a grid:

```
gam.arsenic.pred2 = predict(switch.gam2, newdata = switch.grid, type = "response")
gam.arsenic.grid2 = data.frame(switch.grid, switch.prob = as.vector(gam.arsenic.pred2))
ggplot(gam.arsenic.grid2, aes(x = arsenic, y = switch.prob,
                              group = dist, color = dist)) + geom_line() +
  xlab("Arsenic concentration") + ylab("Probability of switching")
```




```
switch.grid3 = expand.grid(arsenic = seq(0.5, 2.5, 0.5), dist = seq(0:339), educ = 5)
switch.grid3$log.arsenic = log(switch.grid3$arsenic)
gam.dist.pred = predict(switch.gam2, newdata = switch.grid3, type = "response")
gam.dist.grid = data.frame(switch.grid3, switch.prob = as.vector(gam.dist.pred))
ggplot(gam.dist.grid, aes(x = dist, y = switch.prob,
                          group = arsenic, color = arsenic)) + geom_line() +
  xlab("Distance from nearest safe well") + ylab("Probability of switching")
```



What do you think?

Chapter 8

Multiway data

By multiway data, here we mean data with multiple *categorical* explanatory variables.

8.1 Two-way data, and also maps

READ: Cleveland pp. 302–319.

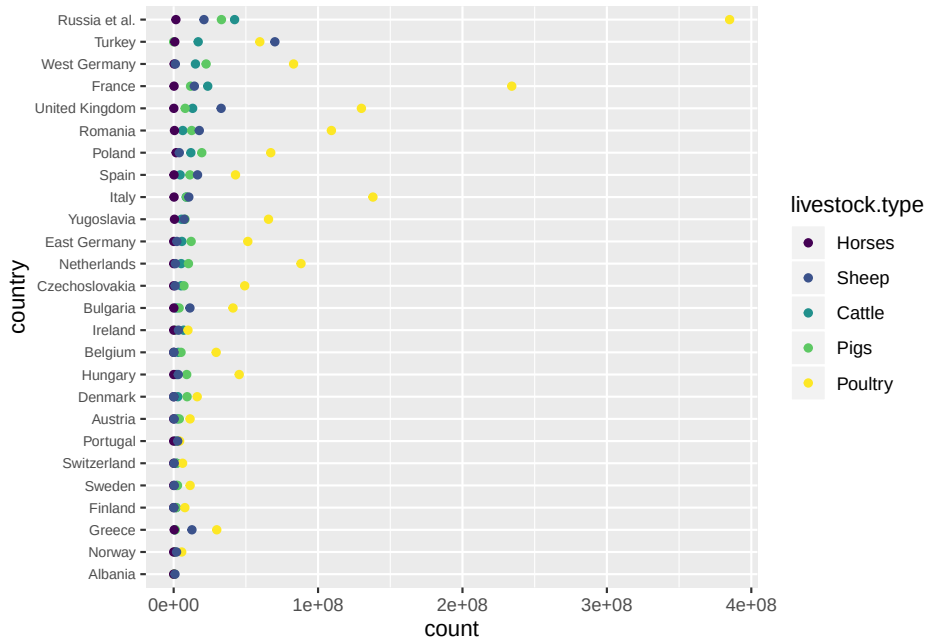
8.1.1 Livestock

Load the data and the `MASS` package:

```
load("lattice.RData")
library(MASS)
```

The data frame `livestock` contains the number of horses, sheep, cattle, pigs, and poultry in 26 European countries in the 1980s. (You can tell it's from the Eighties because some of the countries don't exist anymore.) The data is in long form, with variable names `livestock.type`, `country`, and `count`. Let's first show a dot plot that gives the big picture.

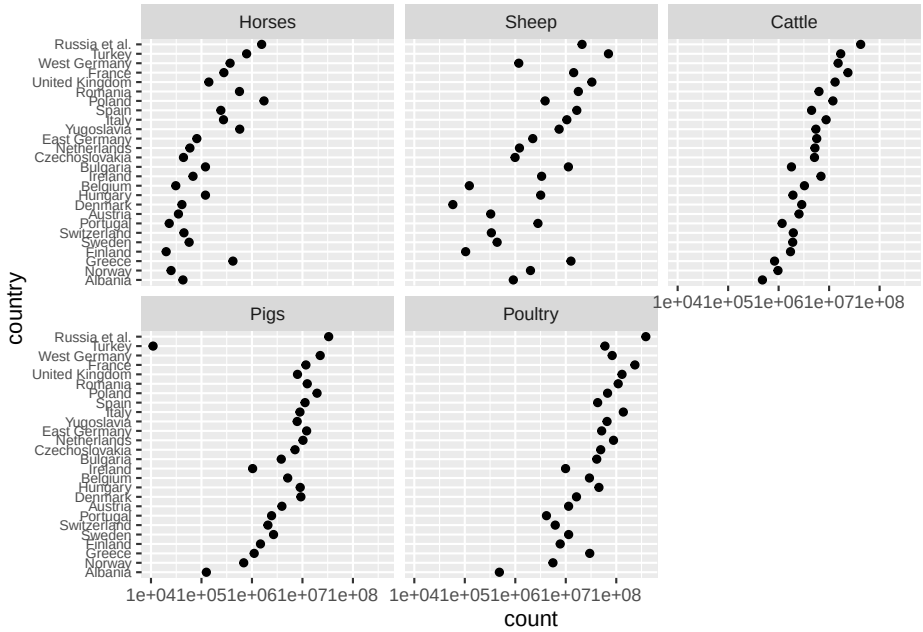
```
ggplot(livestock, aes(x = count, y = country, color = livestock.type)) +
  geom_point() + theme(axis.text.y = element_text(size=7))
```



The data is extremely skewed, because some countries have a lot of chickens. From hereon out we'll do a $\log_{10}$ transformation of count.

We'll condition on animal and country in turn. First, facet wrap on `livestock.type`:

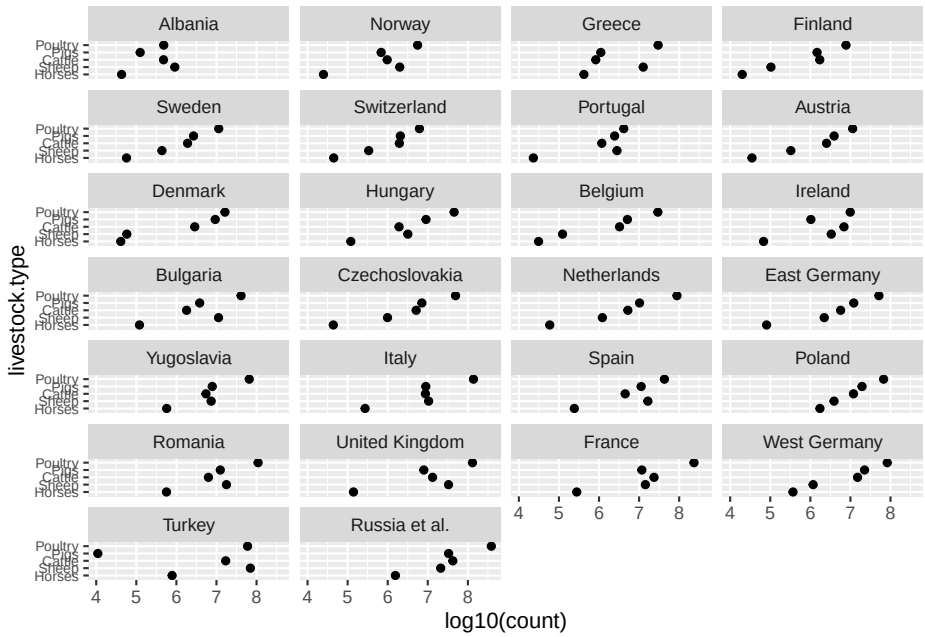
```
ggplot(livestock, aes(x = count, y = country)) + geom_point() +
  facet_wrap(~livestock.type, ncol=3) + scale_x_log10() +
  theme(axis.text.y = element_text(size=7))
```



Note that Turkey has a huge outliers for "pigs" due to being a predominantly Muslim country. The other historically Muslim country in the data set is Albania, which also has a low value for pigs. However, at this stage it's hard to tell if Albania's low pig count is unusual or just because the country is small.

Next, facet wrap on country.

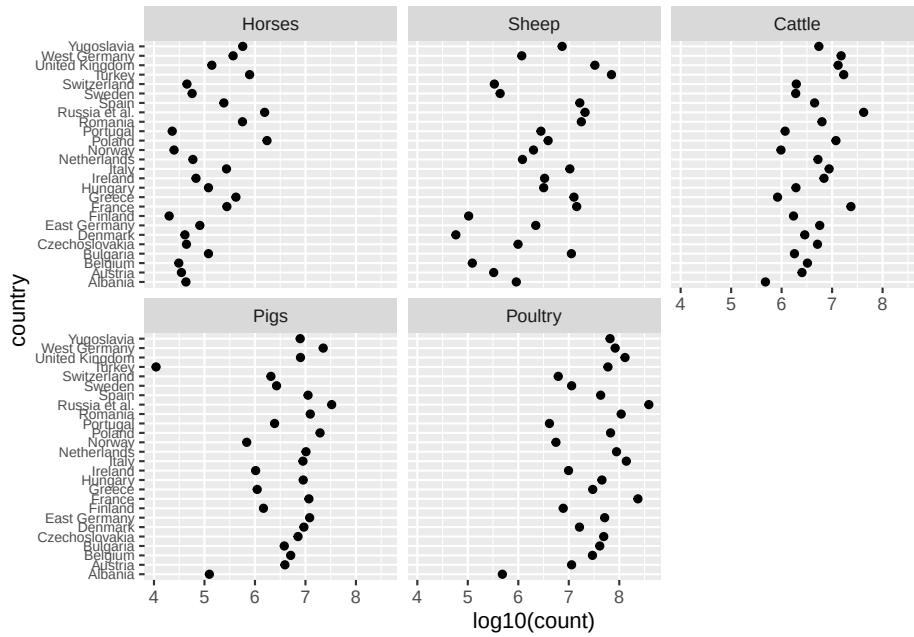
```
ggplot(livestock, aes(x = log10(count), y = livestock.type)) + geom_point() +
  facet_wrap(~country, ncol=4) + theme(axis.text.y = element_text(size=7))
```



The majority of the countries have the lowest count for horses and the higher count for poultry. The other three animals are not in a consistent order.

So far, we've just plotted the data in the order that it came in (which was sorted by country median.) To change the order, we specify the order of levels in the underlying factor. For example, we can put the country levels in alphabetical order:

```
country.alpha = factor(livestock$country, sort(levels(livestock$country)))
livestock.alpha = data.frame(livestock.type = livestock$livestock.type,
                             country = country.alpha, count = livestock$count)
ggplot(livestock.alpha, aes(x=log10(count), y=country)) + geom_point() +
  facet_wrap(~livestock.type, ncol=3) + theme(axis.text.y = element_text(size=7))
```



Alphabetical order isn't very useful though. Usually increasing or decreasing numerical order will make the patterns clearer.

8.1.2 Fitting a model

There are lots of ways of varying complexity to fit a model to multiway data. One basic idea is to break the data down into additive pieces: in our case, the logged count would be an animal effect plus a country effect plus a residual. (Of course, we've taken logs, so it's multiplicative on the original scale.)

The simplest way to fit an additive model is to just use `lm()`, i.e. do an ANOVA. As usual, this fits using least squares:

```
livestock.lm = lm(log10(count) ~ livestock.type + country, data = livestock)
dummy.coef(livestock.lm)$livestock.type
```

```
##      Horses       Sheep      Cattle        Pigs     Poultry
## -1.34532352 -0.02910687  0.17887036  0.14358357  1.05197646
```

However, least squares is (also as usual) potentially misleading when there are outliers. Here, Turkey will throw off least squares pretty badly.

An outlier-resistant alternative is to use `rlm()` with bisquare. Because Turkey is pretty far out there, we need to allow a large number of iterations for the algorithm to converge.

```
livestock.rlm = rlm(log10(count) ~ livestock.type + country,
                    psi = psi.bisquare, maxit = 50, data = livestock)
```

```
dummy.coef(livestock.rlm)$livestock.type
```

```
##      Horses       Sheep      Cattle        Pigs     Poultry
## -1.43805090  0.07393821  0.11930031  0.23322721  1.01158517
```

The coefficients are now estimating median effects instead of mean effects. The coefficients for horses, sheep, and pigs change a lot.

An alternative method (by Tukey) is to use **median polish**:

```
livestock.wide = livestock %>% spread(livestock.type, count)
row.names(livestock.wide) = livestock.wide$country
livestock.wide = livestock.wide[,-1]
livestock.mp = medpolish(log10(livestock.wide))
```

```
## 1: 37.85129
## 2: 35.64557
## Final: 35.42729
```

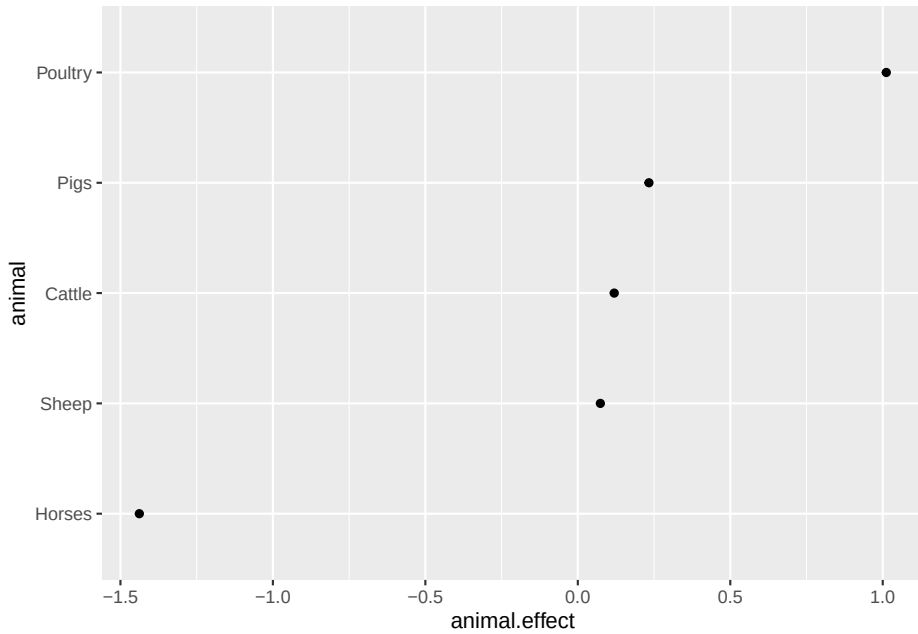
```
livestock.mp$col
```

```
##      Horses       Sheep      Cattle        Pigs     Poultry
## -1.62480255 -0.15773973  0.00000000  0.04181838  0.86008387
```

Note that here the median is set to zero rather than the sum.

Anyway, I don't really understand median polish, so let's stick with the `rlm` for now. We wish to plot the effect sizes using dot plots. We first do the animal effects:

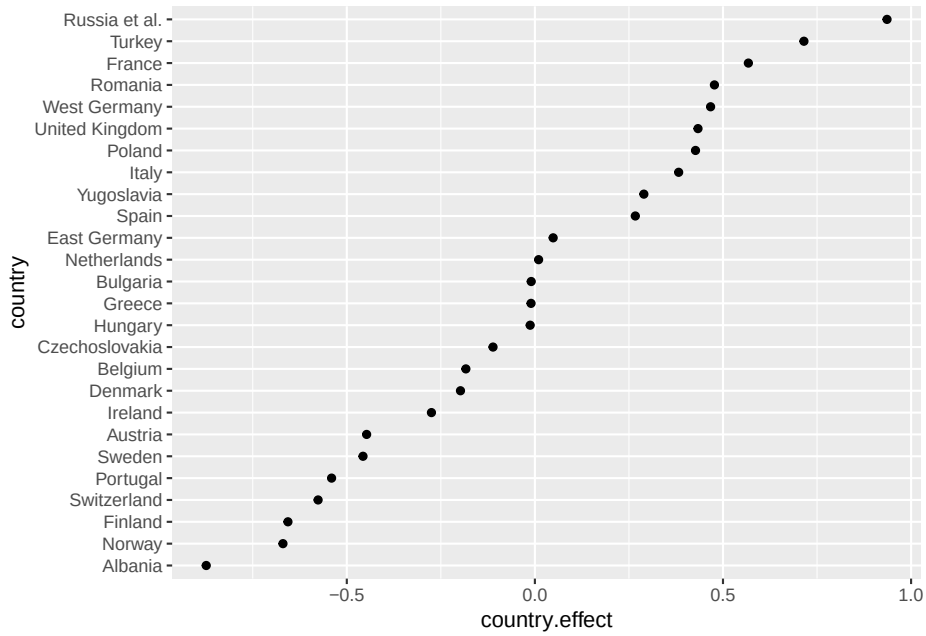
```
animal.effect = sort(dummy.coef(livestock.rlm)$livestock.type)
animal = factor(names(animal.effect), levels = names(animal.effect))
animal.effect.df = data.frame(animal, animal.effect)
ggplot(animal.effect.df, aes(x = animal.effect, y = animal)) + geom_point()
```



As we saw visually, poultry is the highest and horses are the lowest. The other three are all pretty close.

Now do country effects:

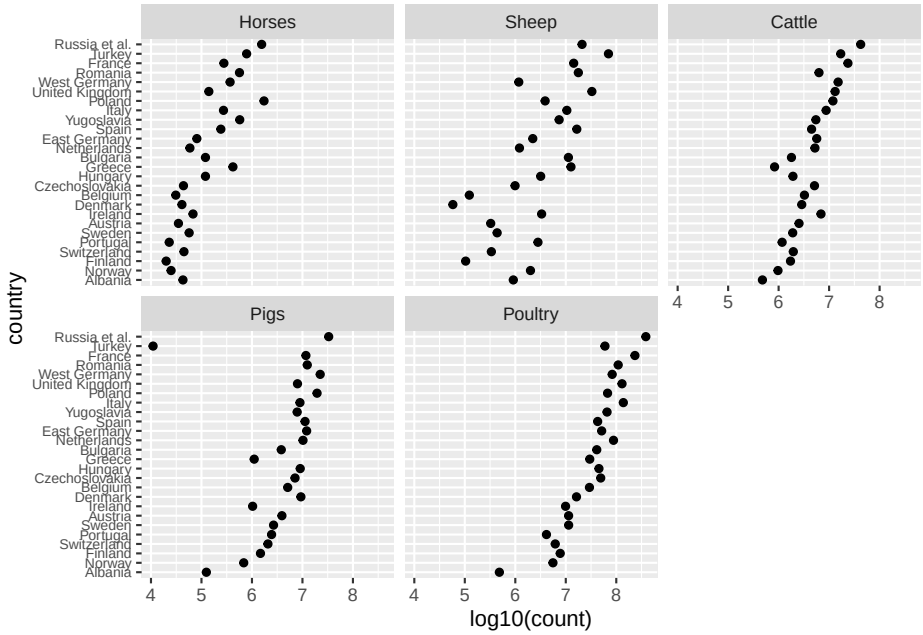
```
country.effect = sort(dummy.coef(livestock.rlm)$country)
country = factor(names(country.effect), levels = names(country.effect))
country.effect.df = data.frame(country, country.effect)
ggplot(country.effect.df, aes(x = country.effect, y = country)) + geom_point()
```



The former U.S.S.R. has the biggest numbers, while Albania has the smallest. It would be interesting to see whether the effects relate more closely to geographic area or population size.

We can now redraw our original faceted plots, reordering the countries by their effect size (the animals are already in the right order.) First, condition on animal:

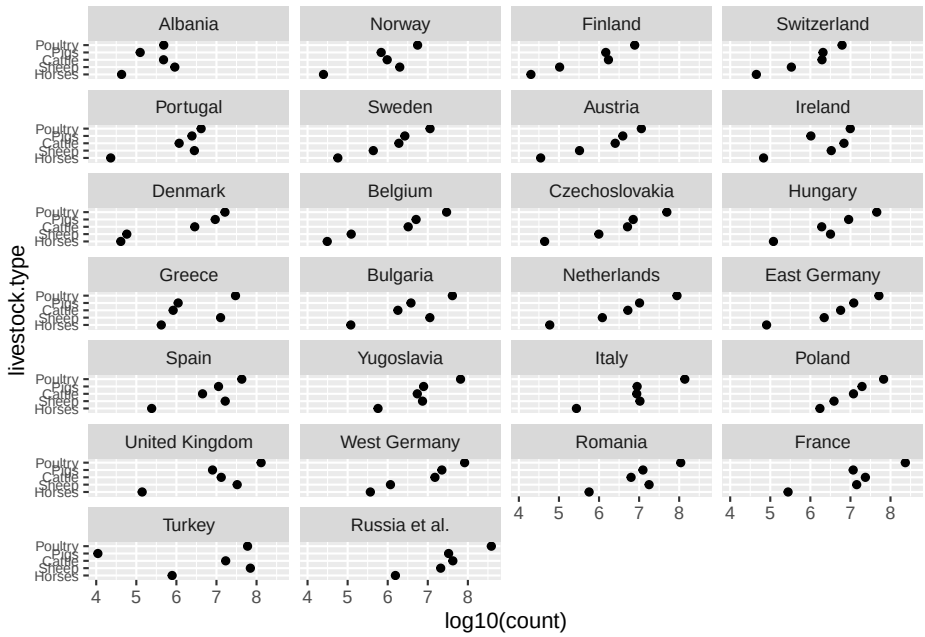
```
livestock.rlm.df = data.frame(livestock,
  .fitted = fitted.values(livestock.rlm), .resid = residuals(livestock.rlm))
livestock.rlm.df$country = factor(livestock$country,
                                  levels = names(country.effect))
livestock.rlm.df$livestock.type = factor(livestock$livestock.type,
                                         levels = names(animal.effect))
ggplot(livestock.rlm.df, aes(x=log10(count), y=country)) + geom_point() +
  facet_wrap(~livestock.type, ncol=3) + theme(axis.text.y = element_text(size=7))
```



The order is slightly different now; France, for example, is up from fourth to third. (Note that our order is different from in Cleveland's book; I think he didn't do enough iterations.)

Now condition on country. Only the order of the panels changes:

```
ggplot(livestock.rlm.df, aes(x=log10(count), y=livestock.type)) + geom_point() +
  facet_wrap(~country, ncol=4) + theme(axis.text.y = element_text(size=7))
```

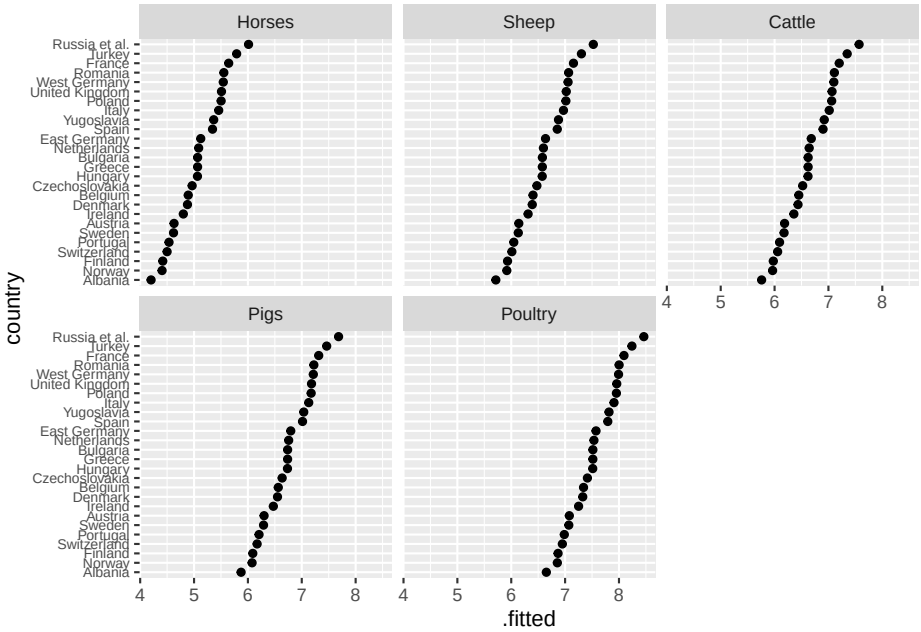


We could also combine the facets on to one plot and distinguish by color. However, that would be a bit busy and disguise the separation into within-country and across-country variation that's the point of the additive model.

8.1.3 Fitted values and residuals

The fitted values consist of center plus animal effect plus country effect. Condition on animal:

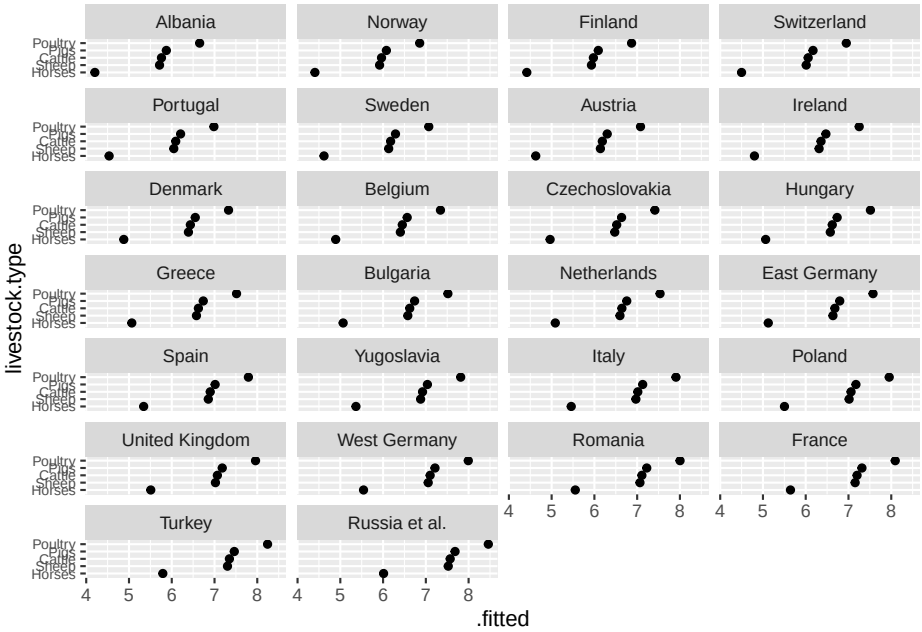
```
ggplot(livestock.rlm.df, aes(x=.fitted, y=country)) + geom_point() +
  facet_wrap(~livestock.type, ncol=3) + theme(axis.text.y = element_text(size=7))
```



We see the "curve" just shifts to the right as we get to the more popular animals. The "slope" of the curve is perhaps surprisingly close to vertical: on a log scale, there isn't that much difference between Russia and Albania. Even on the original scale, the Russia multiplier is only about 60 times the Albania multiplier – this might sound like a lot but the former U.S.S.R. was really, really big.

Now condition on country.

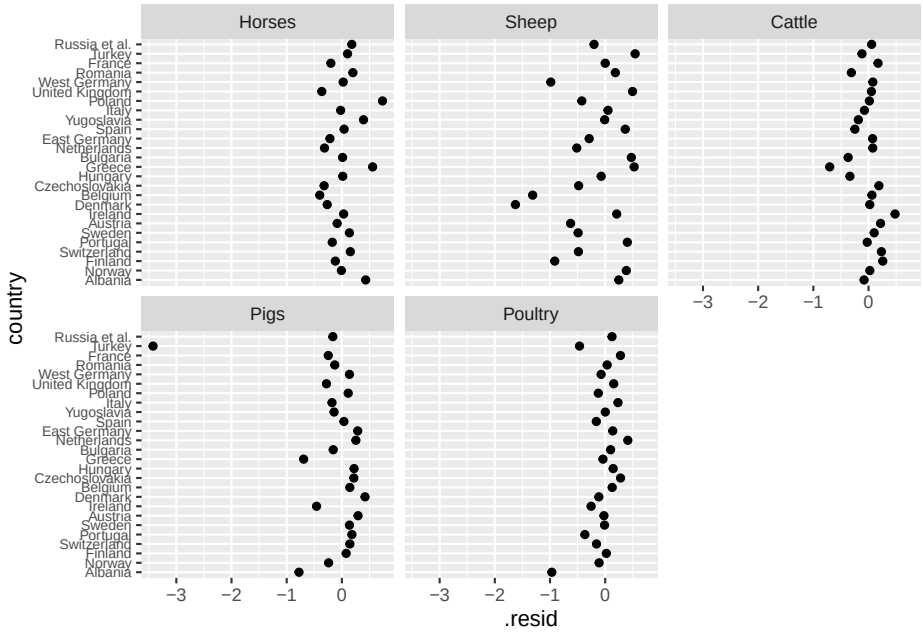
```
ggplot(livestock.rlm.df, aes(x=.fitted, y=livestock.type)) + geom_point() +
  facet_wrap(~country, ncol=4) + theme(axis.text.y = element_text(size=7))
```



This is pretty boring – the additive model means we see exactly the same pattern 26 times. The only difference is a very gradual shift to the right (more animals.)

Looking at the residuals should be more interesting. Condition on animal type:

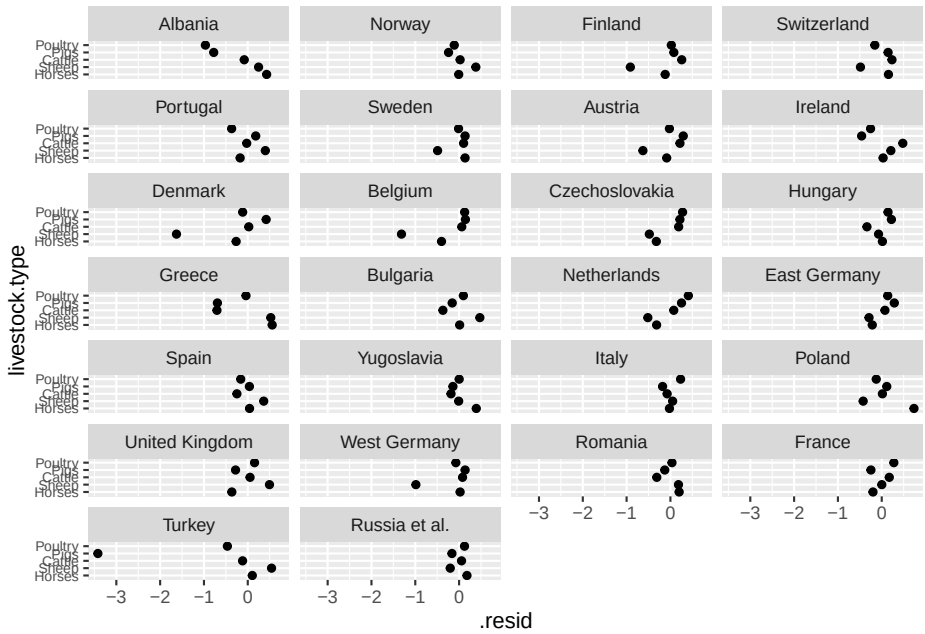
```
ggplot(livestock.rlm.df, aes(x=.resid, y=country)) + geom_point() +
  facet_wrap(~livestock.type, ncol=3) + theme(axis.text.y = element_text(size=7))
```



The Turkey-pig outlier is still apparent. We now see that Albania's pig population is somewhat lower than we'd expect from the additive model, but is nowhere near as extreme. The variation in sheep is also notable: some northern European countries just don't like sheep very much.

Condition on country:

```
ggplot(livestock.rlm.df, aes(x=.resid, y=livestock.type)) + geom_point() +
  facet_wrap(~country, ncol=4) + theme(axis.text.y = element_text(size=7))
```



Again, we see that sheep are nearly an outlier in countries like Denmark, Belgium, and West Germany. In European countries (in the 1980s), milk and eggs were a consistent part of every culture, while wool and sheep meat were more specialized.

8.1.4 Maps

Maps are overrated. They can be useful when:

- Longitude and latitude are inherently of interest (e.g. when studying climate)
- The audience knows the underlying political geography well (e.g. most Americans know more or less which state goes where.) Even in this case maps can be distracting (because it's not like the shape of the state matters) or misleading (because geographic size might not matter at all.)

It's quite common that neither of these conditions are met. In our case, most people outside Europe won't know which country is Romania and which is Bulgaria, so we might as well show them dot plots. Nevertheless, maps look cool and knowing how to draw them might impress someone enough that they give you a job, so we'll address them briefly. First, let's set up a data frame:

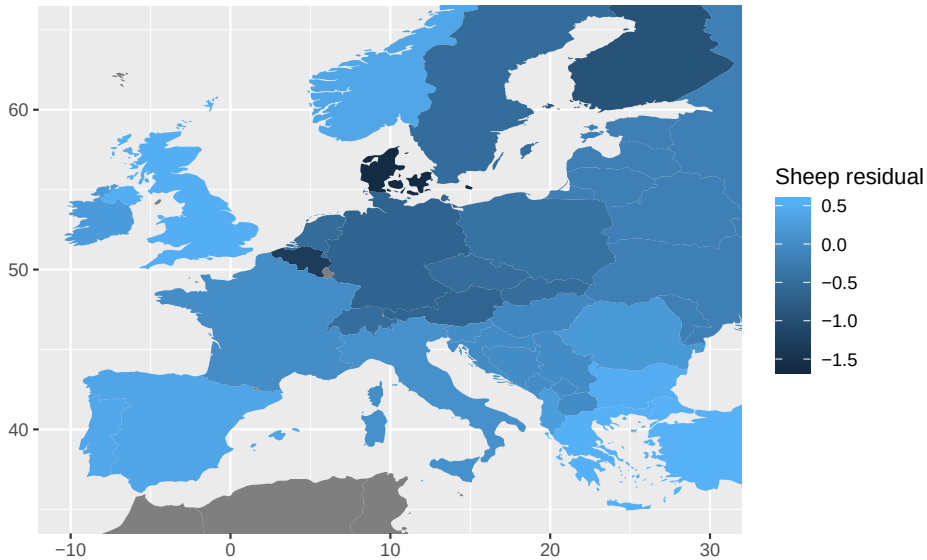
```
sheep.df = subset(livestock.rlm.df, livestock.type == "Sheep")
names(sheep.df)
```

```
## [1] "livestock.type" "country" "count" ".fitted"
## [5] ".resid"
```

```
# Change "country" column name to "region"
names(sheep.df)[2] = "region"
# What about Germany?
# Fudge: just average West and East
# (this is terrible code
# but this is *really* hard to do in R)
sheep.df = rbind(sheep.df, rep(NA, 5))
n = nrow(sheep.df)
WG = which(sheep.df$region == "West Germany")
EG = which(sheep.df$region == "East Germany")
sheep.df[n, 1] = "Sheep"
levels(sheep.df$region) = c(levels(sheep.df$region), "Germany")
sheep.df[n, 2] = "Germany"
sheep.df[n, 3] = sheep.df[WG, 3] + sheep.df[EG, 3]
sheep.df[n, 4] = (sheep.df[WG, 4] + sheep.df[EG, 4]) / 2
sheep.df[n, 5] = (sheep.df[WG, 5] + sheep.df[EG, 5]) / 2
```

One option is to use the `maps` library in conjunction with `geom_map()` in `ggplot2`. We have to fiddle a bit with the country names, e.g. the world map calls the United Kingdom "UK" and doesn't know that these are the same place.

```
library(maps)
map.world = map_data(map = "world")
# Pretend it's the Eighties again
map.world$region[map.world$region == "UK"] = "United Kingdom"
map.world$region[map.world$region == "Czech Republic"] = "Czechoslovakia"
map.world$region[map.world$region == "Slovakia"] = "Czechoslovakia"
map.world$region[map.world$region == "Russia"] = "Russia et al."
map.world$region[map.world$region == "Latvia"] = "Russia et al."
map.world$region[map.world$region == "Lithuania"] = "Russia et al."
map.world$region[map.world$region == "Estonia"] = "Russia et al."
map.world$region[map.world$region == "Belarus"] = "Russia et al."
map.world$region[map.world$region == "Ukraine"] = "Russia et al."
map.world$region[map.world$region == "Moldova"] = "Russia et al."
map.world$region[map.world$region == "Serbia"] = "Yugoslavia"
map.world$region[map.world$region == "Montenegro"] = "Yugoslavia"
map.world$region[map.world$region == "Croatia"] = "Yugoslavia"
map.world$region[map.world$region == "Bosnia and Herzegovina"] = "Yugoslavia"
map.world$region[map.world$region == "Kosovo"] = "Yugoslavia"
map.world$region[map.world$region == "Macedonia"] = "Yugoslavia"
map.world$region[map.world$region == "Slovenia"] = "Yugoslavia"
sheep.merge = as.data.frame(merge(map.world, sheep.df, by="region", all.x = TRUE, all.y = FALSE))
ggplot(sheep.merge, aes(fill=.resid)) + geom_map(map=map.world, aes(map_id=region)) +
  coord_equal() + xlim(-10, 30) + ylim(35, 65) +
  scale_fill_continuous(name = "Sheep residual")
```



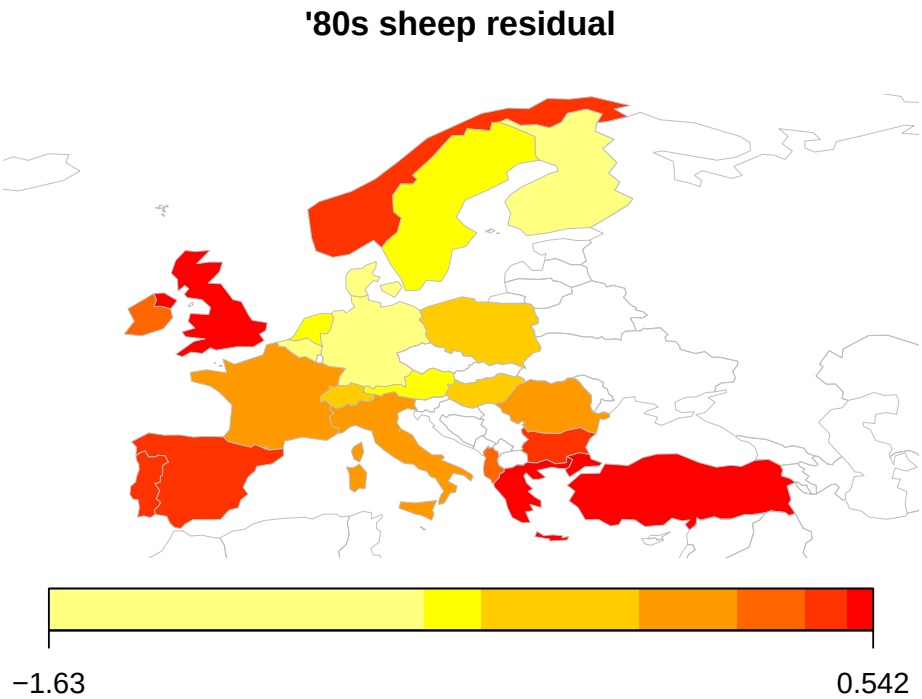
The lighter colors are sheepy countries, while the darker ones are less sheepy. In general, the map gets darker as you move north, though there are plenty of exceptions, like the U.K. Note that the one thing I didn't fix was combining West and East Germany, because I was too lazy to do it properly. (The easiest non-cheating solution would be to combine both Germanies in the original data set and refit the whole model.)

A slightly more user-friendly option is the `rworldmap` package.

```
# install.packages("rworldmap")
library(rworldmap)
sheep.join = joinCountryData2Map(sheep.df, joinCode = "NAME",
                                 nameJoinColumn = "region")
```

```
## 22 codes from your data successfully matched countries in the
map
## 5 codes from your data failed to match with a country code in
the map
## 221 codes from the map weren't represented in your data
```

```
mapCountryData(sheep.join, nameColumnToPlot = ".resid",
               mapRegion = "europe", mapTitle = "'80s sheep residual")
```



Again, you can rename things to fill in some of the blanks for newly split countries. You probably get the point by now, though.

8.2 Multiway data

READ: Cleveland pp. 320–340.

8.2.1 Run time

The `run.time` data frame contains experimental results by Bentley and McIllroy (1992) on the running time required for three sorting algorithms. The variables are:

- `time` in seconds. This is strongly skewed, so we'll mostly use $\log_2$ of time;
- `algorithm`: `7th` (qsort Seventh Edition), `ucb` (qsort in Berkeley UNIX), or `new` (Bentley & McIllroy's new algorithm);
- `machine`: `vax` or `mips`;
- `input`: The type of data to sort: `integer`, `float`, `double`, `record`, `pointer`, or `string`.

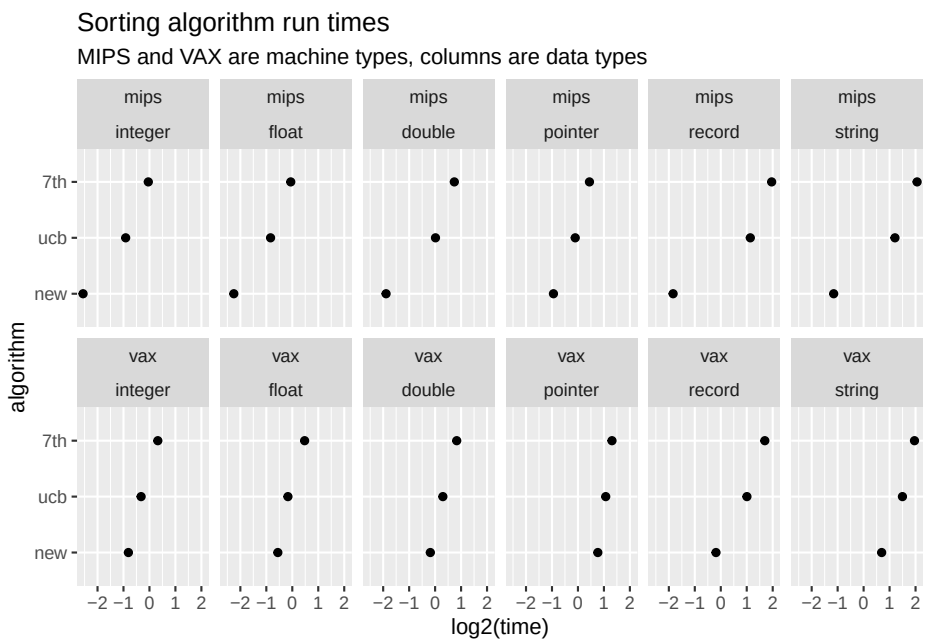
This time we have *three-way* data: there are three explanatory factor variables, and one numerical response (time.) To display the data, we can:

- Facet on two factors, and draw dotplots showing the relationship of the response with the third factor; or

- Facet on one factor, distinguish a second factor by color, and draw dotplots showing the relationship of the response with the third factor.

First, we try faceting two ways. The algorithm is the main explanatory variable of interest, so condition on the other two.

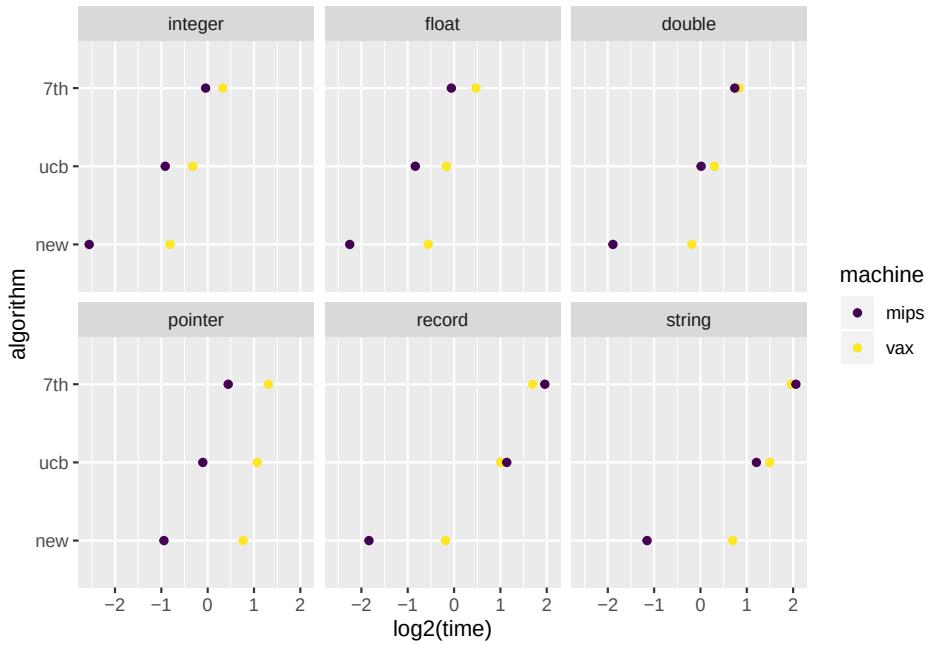
```
gg = ggplot(run.time, aes(x = log2(time), y = algorithm)) + geom_point() +
  facet_wrap(~machine + input, ncol=6) + ggtitle("Sorting algorithm run times")
gg + labs(subtitle = "MIPS and VAX are machine types, columns are data types")
```



This shows that the order (from shortest to longest run time) is always new < ucb < 7th. However, the differences aren't constant.

Now keep algorithm as the main explanatory, but use color to distinguish one of the others. Since there are only two machines, color by that variable to avoid too many different colors (which requires your eyes going to the legend and back repeatedly and is thus annoying.)

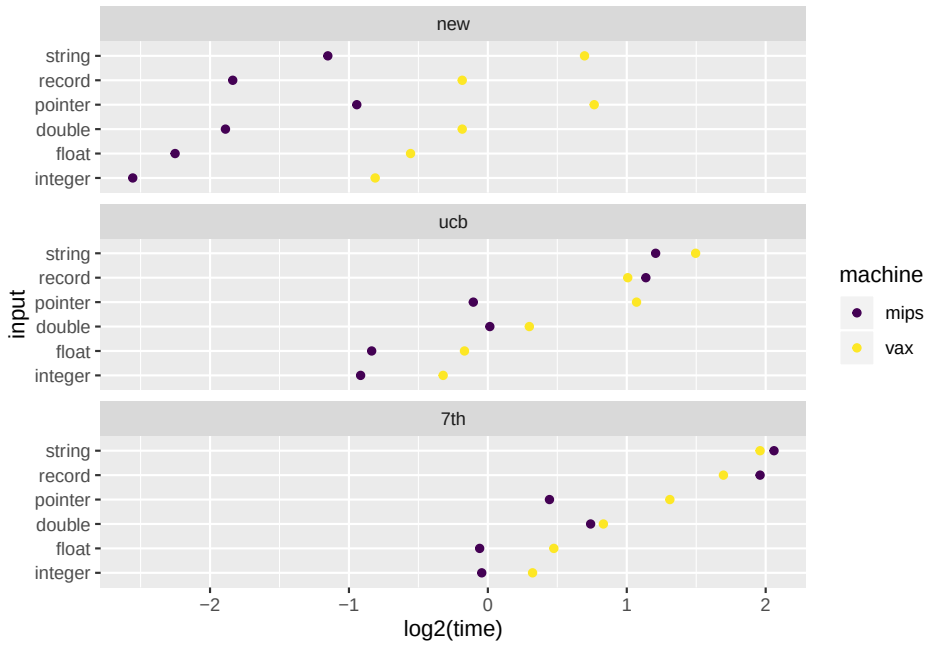
```
ggplot(run.time, aes(x = log2(time), y = algorithm, color = machine)) +
  geom_point() + facet_wrap(~input, ncol=3)
```



The "slopes" of the red dots are systematically steeper than the "slopes" of the blue dots. That suggest an algorithm-machine interaction.

Now make input the main explanatory. Facet on algorithm and keep machine as color:

```
ggplot(run.time, aes(x = log2(time), y = input, color = machine)) +
  geom_point() + facet_wrap(~algorithm, ncol = 1)
```



Interestingly, the top plot looks like a simple shift from red to blue, the other two plots look much more complicated. To be on the safe side, we should keep an input-machine interaction in the model.

8.2.2 Interlude: The `spread()` function

We know that the new algorithm gives the fastest run times. So what we want to quantify is how much faster it is than the other two algorithms (on the log 2 scale.) In other words, we're interested in the *differences* between the two old algorithms and the new one. With a well-organized data set like this one, it's straightforward to find the differences manually. However, with messier data, it might not be so easy. We thus take this opportunity to learn the `spread()` function in `tidyr`.

`spread()` is nearly the opposite of `gather()`: instead of collapsing columns and making our data long, we break up columns and make our data wide. Run the following line:

```
run.wide = run.time %>% spread(algorithm, time)
```

What this does is create new columns, one for each level of `algorithm`. The numbers that go into these columns are the ones currently in `time`. This is the result:

```
run.wide
```

```
##    machine   input  new  ucb  7th
```

```
## 1     mips integer 0.17 0.53 0.97
## 2     mips   float 0.21 0.56 0.96
## 3     mips  double 0.27 1.01 1.67
## 4     mips pointer 0.52 0.93 1.36
## 5     mips  record 0.28 2.20 3.89
## 6     mips  string 0.45 2.31 4.17
## 7      vax integer 0.57 0.80 1.25
## 8      vax   float 0.68 0.89 1.39
## 9      vax  double 0.88 1.23 1.78
## 10     vax pointer 1.70 2.10 2.48
## 11     vax  record 0.88 2.01 3.24
## 12     vax  string 1.62 2.82 3.89
```

Now create two new variables with the improvements in logged run time of the new algorithm over the two old ones. More positive means more improvement.

```
run.wide$imp.ucb = log2(run.wide$ucb) - log2(run.wide$new)
run.wide$imp.7th = log2(run.wide$"7th") - log2(run.wide$new)
```

Now that we've created these two variables, we can switch back to long form.

```
run.long = run.wide %>% gather(algorithm, improvement, imp.ucb:imp.7th)
```

Fix the level names for the `algorithm` variable:

```
run.long$algorithm = recode(run.long$algorithm, "imp.7th" = "7th",
                            "imp.ucb" = "ucb")
```

Let's summarize our new `improvement` variable by finding its mean for each input.

```
improvement.means = aggregate(improvement ~ input, mean, data = run.long)
improvement.means
```

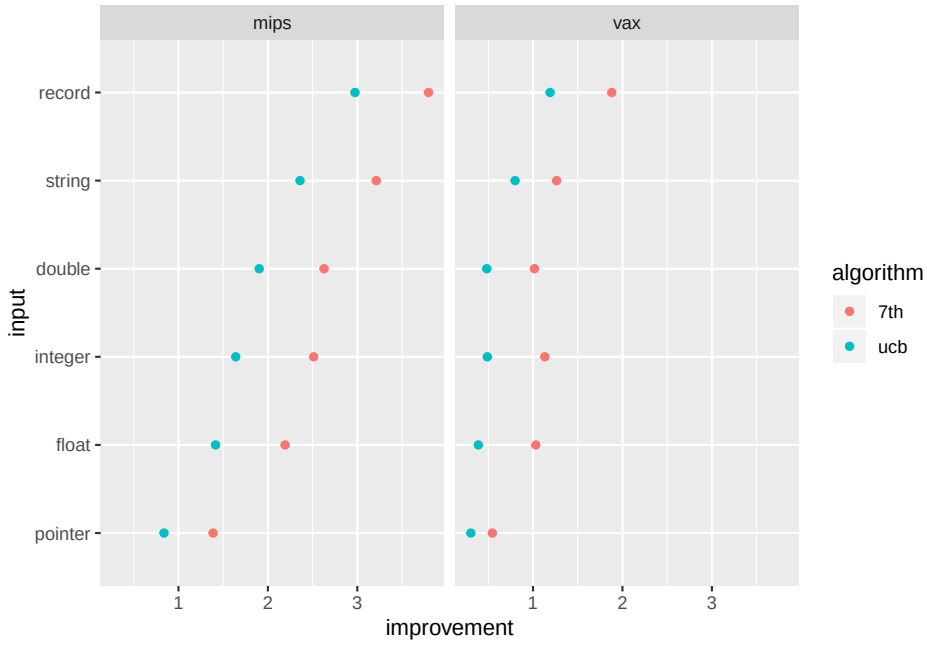
```
##     input improvement
## 1 integer   1.4437100
## 2   float   1.2568578
## 3  double   1.5078814
## 4 pointer   0.7688505
## 5  record   2.4605787
## 6  string   1.9088560
```

For aesthetic purposes, reorder the `input` variable by mean improvement:

```
input.order = improvement.means$input[order(improvement.means$improvement)]
run.long$input = factor(run.long$input, levels = input.order)
```

Plot the data, this time coloring by algorithm:

```
ggplot(run.long, aes(x = improvement, y = input, color = algorithm)) +
  geom_point() + facet_grid(~machine) + scale_color_discrete()
```



Interestingly, the difference between blue and red is close to constant on the left panel and close to (a different) constant on the right panel. In other words, we might not need an algorithm-input interaction.

8.2.3 Multiway linear model

Let's say we want to fit input:machine and algorithm:machine interactions, but not one for input:algorithm. There are several ways to specify this. Since there are no outliers, we can just fit an ordinary linear model. We can write out the interactions explicitly:

```
runtime.lm = lm(improvement ~ input + algorithm + machine + input:machine +
                  algorithm:machine, data = run.long)
```

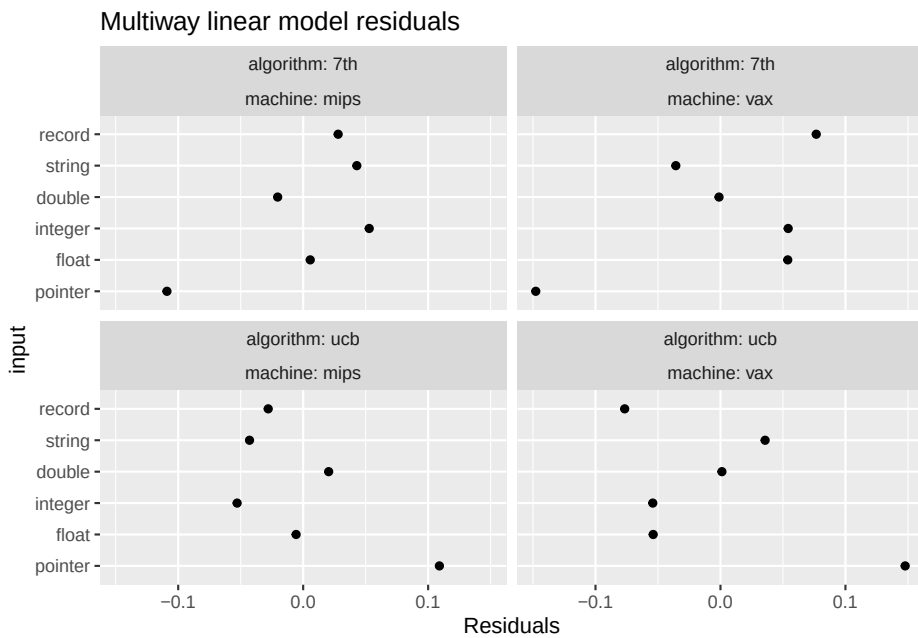
Or we can think of it the followay way: "Without machine, input and algorithm are addtive. But both interact with machine."

```
runtime.lm = lm(improvement ~ (input + algorithm) * machine, data = run.long)
```

You should get the same thing either way. We check the residuals by plotting them faceted two ways:

```
library(broom)
runtime.lm.df = augment(runtime.lm)
```

```
gg = ggplot(runtime.lm.df, aes(x = .resid, y = input)) + geom_point() +
  facet_wrap(~ algorithm + machine, labeller = label_context)
gg + xlab("Residuals") + ggtitle("Multiway linear model residuals")
```



There's not much to note here except that the "pointer" residuals are the largest in magnitude. Do we need an input-algorithm interaction just for pointer? Perhaps, but this could also just be luck.

Now check R^2:

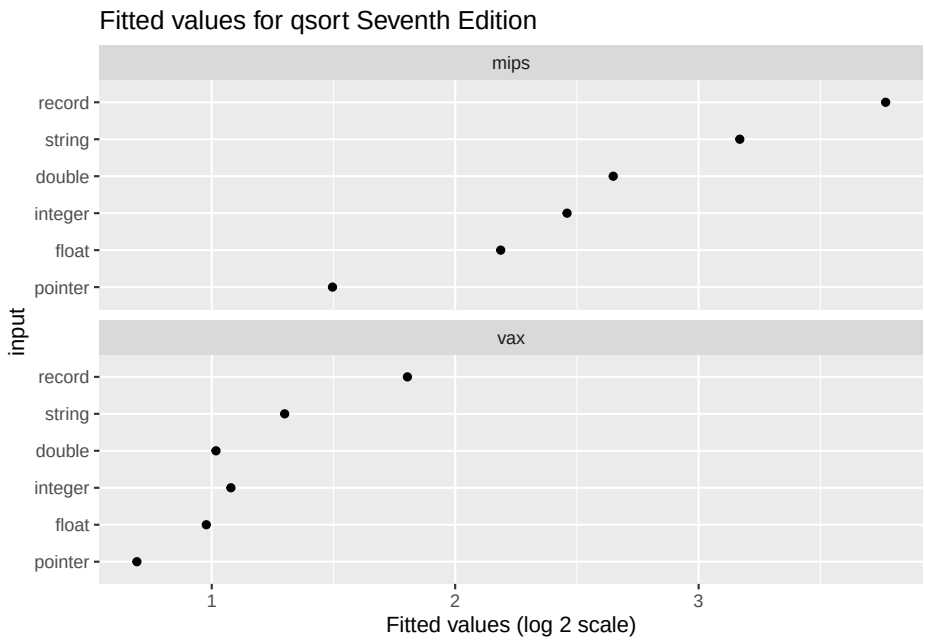
```
var(runtime.lm.df$.fitted) / var(runtime.lm.df$improvement)
```

```
## [1] 0.9951009
```

We've captured almost all of the variation. This may not be as impressive as it seems, since we fitted a 14-parameter model to 24 data points, but we'll take it.

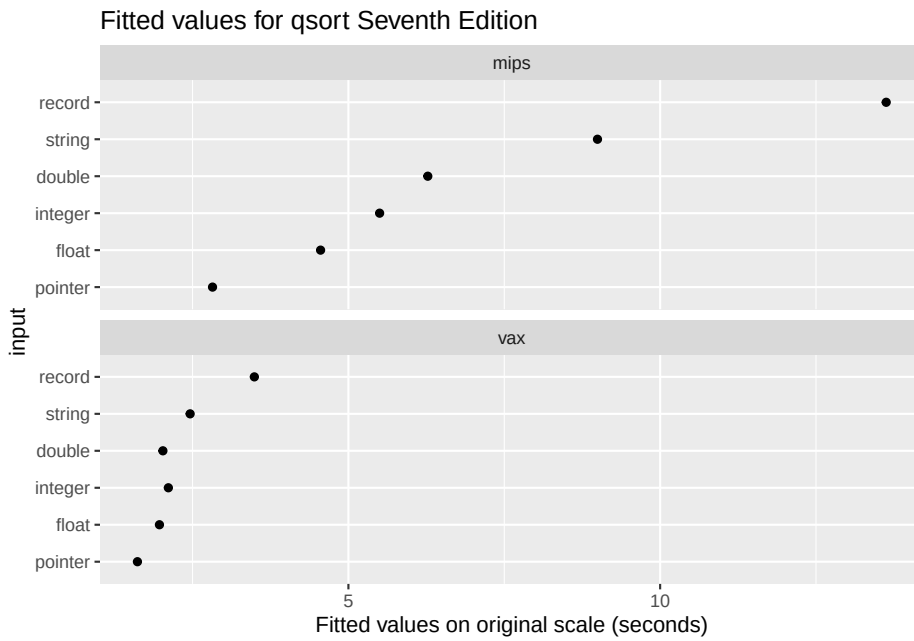
To better understand the input:machine interaction, pick an algorithm and draw a dot plot of fitted values with those two explanatories.

```
gg = ggplot(subset(runtime.lm.df, algorithm == "7th"),
  aes(x = .fitted, y = input)) + geom_point() + facet_wrap(~machine, ncol = 1)
gg + ggtitle("Fitted values for qsort Seventh Edition") +
  xlab("Fitted values (log 2 scale)")
```



We see the shape of the two sets of dots is quite different. It's not as simple as adding different effects for mips and vax. We can also replot this on the original scale:

```
gg = ggplot(subset(runtime.lm.df, algorithm == "7th"),
            aes(x = 2^(.fitted), y = input)) + geom_point()
gg + facet_wrap(~machine, ncol = 1) +
  ggtitle("Fitted values for qsort Seventh Edition") +
  xlab("Fitted values on original scale (seconds)")
```



The back-transformed fitted values give the number of times longer the old algorithms take compared to the new one. The new algorithm is always faster, but the increase in speed is highly variable. mips looks particularly bad in comparison, usually taking over five times as long.

8.3 The barley controversy

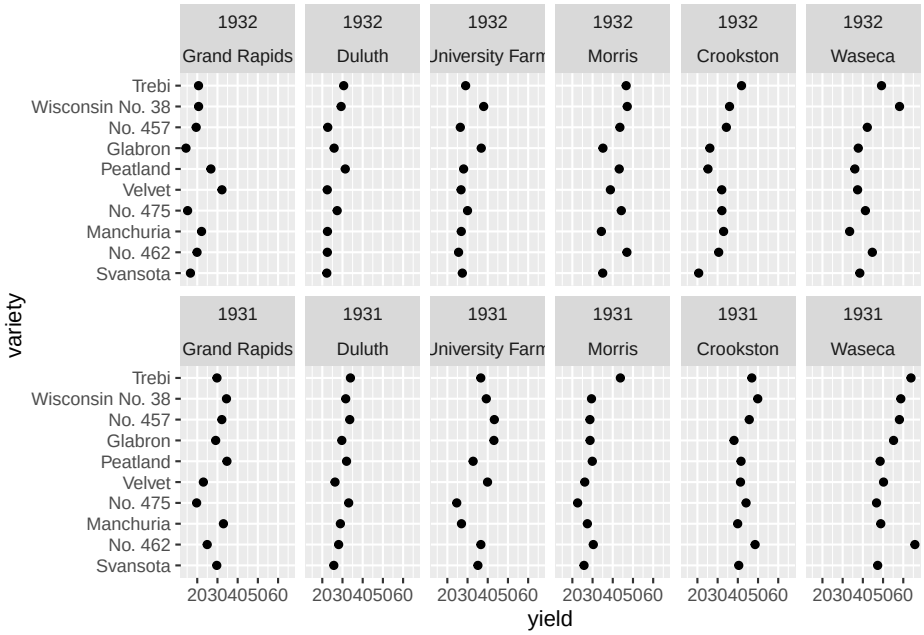
Immer's Minnesota barley data is famous as one of the first real-life applications of analysis of variance. It's also controversial due to a supposed mistake in the data. The variables in the `barley` data frame are:

- `yield`: barley crop yield in bushels per acre (a bushel is four pecks);
- `variety`: one of ten types of barley;
- `year`: 1931 or 1932;
- `site`: one of six locations in Minnesota.

So there are three categorical explanatory variables (we might as well treat year as categorical, since we only have two years) and a quantitative response. The factor variables are presorted by median.

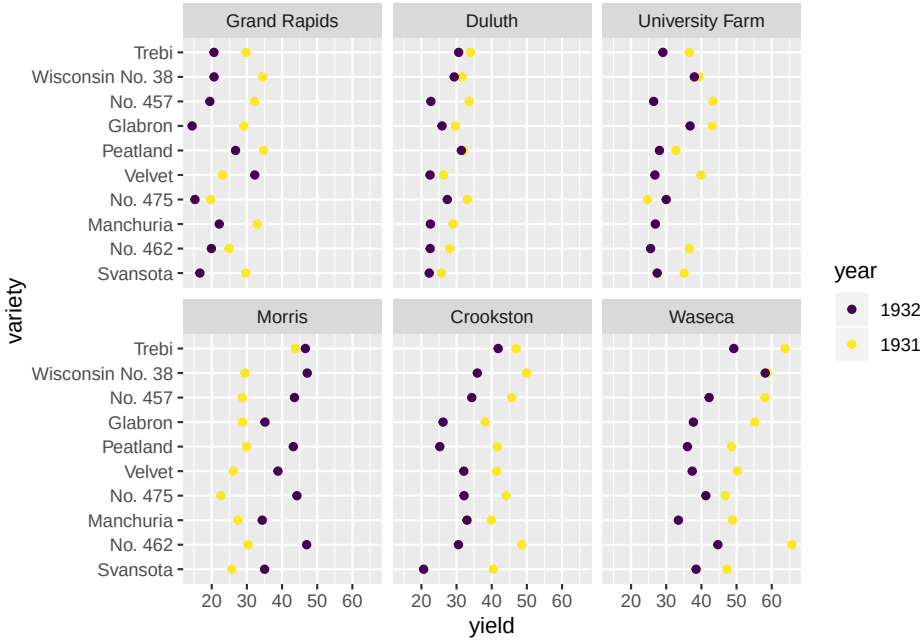
Start with a two-way faceted dot plot, with `variety` as the main explanatory:

```
ggplot(barley, aes(x = yield, y = variety)) + geom_point() +
  facet_wrap(~ year + site, ncol = 6)
```



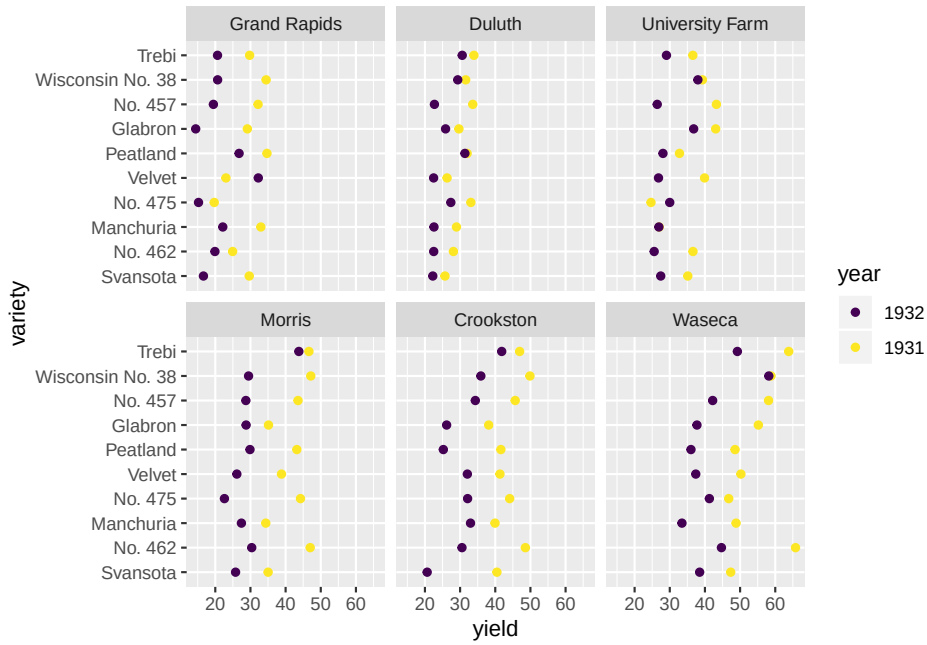
There are a few too many dots here for the patterns to be clear. Instead, let's plot one row on top of the other, distinguishing year by color.

```
ggplot(barley, aes(x = yield, y = variety, color = year)) +
  geom_point() + facet_wrap(~site)
```



In most cases, the teal dot is to the right of the red dot: that is, the 1931 yield was higher than the 1932 yield. However, the opposite is true at Morris, where every yield increased from 1931 to 1932. Cleveland claims this could have been a mistake in data entry, and we should at least entertain this possibility. Let's swap the 1931 and 1932 numbers for Morris, and see what difference that makes.

```
morris1932fixed = barley$yield[barley$site == "Morris" & barley$year == 1931]
morris1931fixed = barley$yield[barley$site == "Morris" & barley$year == 1932]
barley.fixed = barley
barley.fixed$yield[barley$site == "Morris" & barley$year == 1932] =
  morris1932fixed
barley.fixed$yield[barley$site == "Morris" & barley$year == 1931] =
  morris1931fixed
ggplot(barley.fixed, aes(x = yield, y = variety, color = year)) +
  geom_point() + facet_wrap(~site)
```



On one hand, it looks nicer – the pattern is now consistent apart from the odd exception (Grand Rapids–Velvet.) On the other hand, perhaps the pattern is too good to be true: we know that sometimes outliers happen, and maybe Morris was hit by a plague of locusts in 1931 or something. Personally, the data doesn't entirely convince me, so I'll err on the side of sticking with the original data.

We can simplify the structure by looking at the difference between the 1931 and 1932 yields. To do this, we:

- Put the data into "wide" form by using the `spread()` function in `tidyr`.

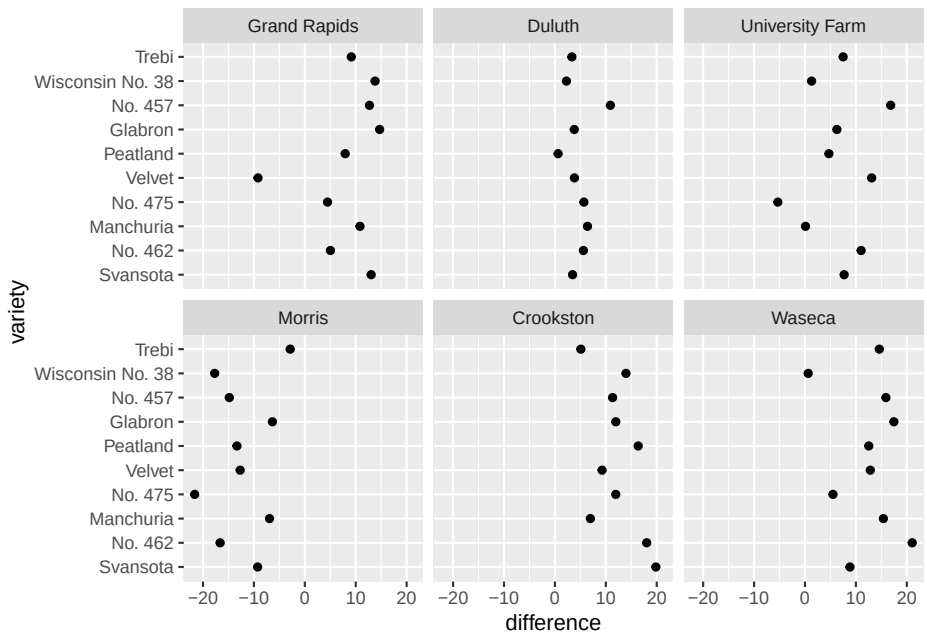
This will create columns called "1931" and "1932" containing the yields for those years, and halve the number of rows.

- Create a new data frame column called `difference` with the 1931 yield minus the 1932 yield for each variety and site. (We do the subtraction this way around because people like positive numbers.)

```
barley.wide = barley %>% spread(year, yield)
barley.wide$difference = barley.wide$"1931" - barley.wide$"1932"
```

Now we draw a dot plot faceting on site:

```
ggplot(barley.wide, aes(x = difference, y = variety)) +
  geom_point() + facet_wrap(~site)
```



The main modeling question we try to answer here is whether some interaction between variety and the other variables is necessary. If the variety-by-variety variation here is small (i.e. each panel's dot are scattered close to vertical line), then perhaps we can do without such interactions. It's a judgment call, but to me there doesn't seem to be much pattern. On the other hand, we probably need a year:site interaction, for Morris if for no other reason. Because of the outliers, we'll use `rlm()` with bisquare for the fit.

```
barley.rlm = rlm(yield ~ variety + year * site, psi = psi.bisquare, data = barley)
```

Before we move on, let's check that we've captured an informative amount of variation.

```
var(fitted.values(barley.rlm)) / var(barley$yield)
```

```
## [1] 0.8422786
```

```
var(fitted.values(barley.rlm)) /
  (var(fitted.values(barley.rlm)) + var(residuals(barley.rlm)))
```

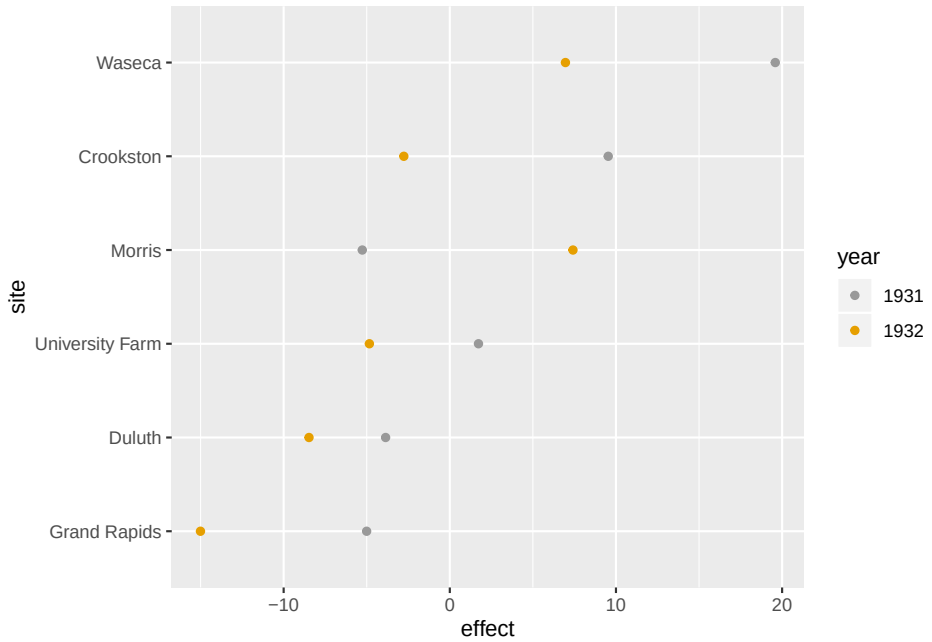
```
## [1] 0.8339532
```

Our model has captured over 80% of the variation in the yields, so let's proceed.

8.3.1 Viewing the model

Because we have a year:site interaction, we should view the effects of these two variables in tandem. Draw a dotplot of the effects, with color indicating year:

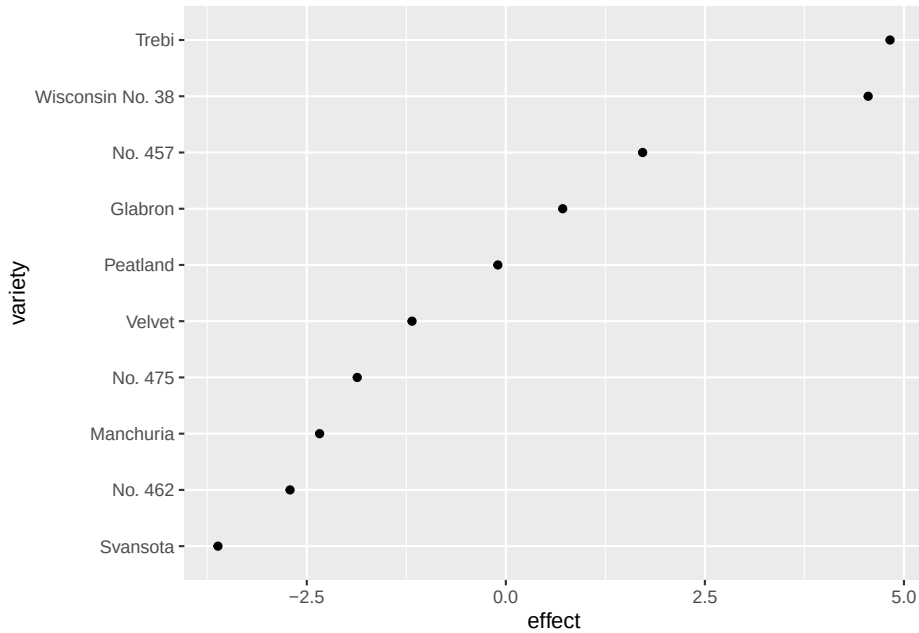
```
barley.effects = dummy.coef(barley.rlm)
year.site.main = outer(barley.effects$year, barley.effects$site, "+")
year.site.inter = barley.effects$"year:site"
year.site.effect = year.site.inter + as.vector(year.site.main)
years = rep(row.names(year.site.main), 6)
sites = rep(colnames(year.site.main), each = 2)
sites = factor(sites, levels = names(barley.effects$site))
year.site.df = data.frame(year = years,
  site = sites,
  effect = year.site.effect)
ggplot(year.site.df, aes(x = effect, y = site, col = year)) +
  geom_point() + scale_color_manual(values = cb_palette)
```



As we know, Morris doesn't fit the pattern. Even excluding Morris, there's still quite a bit of variation between sites, both within years and in the difference from 1932 to 1931.

Let's now look at the variety effects:

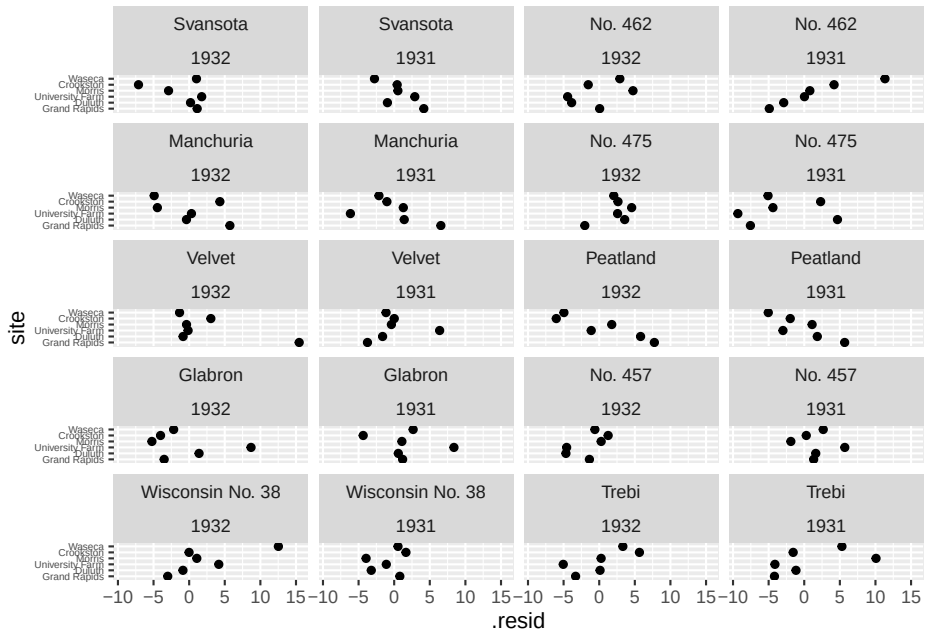
```
variety.effects = sort(barley.effects$variety)
varieties = factor(names(barley.effects$variety),
                   levels = names(barley.effects$variety))
variety.df = data.frame(effect = variety.effects, variety = varieties)
ggplot(variety.df, aes(x = effect, y = variety)) + geom_point()
```



You'd need to know more about pre-World War II strains of barley than I do to get the most out of this graph. Generally though we see that the effects are fairly well-behaved. (You might at least think about fitting a multilevel model, though there's not much data and robust multilevel models are a pain in R.)

Let's now check the residuals for anomalies. Facet by variety and year:

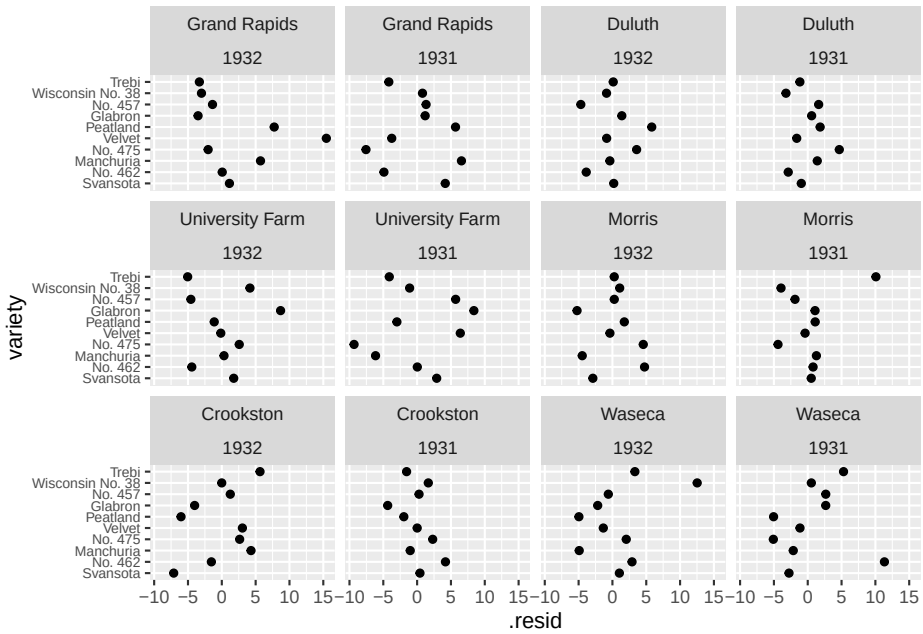
```
barley.rlm.df = augment(barley.rlm)
ggplot(barley.rlm.df, aes(x = .resid, y = site)) +
  geom_point() + facet_wrap(~ variety + year, ncol = 4) +
  theme(axis.text.y = element_text(size=5))
```



Looking carefully, there probably is a bit of a site:variety interaction: the Peatland plots look fairly similar for both 1931 and 1932, for example. But this effect is pretty small, so we ignore it. Otherwise, there's not much of note here aside from an outlier or two.

Now facet by year and location:

```
ggplot(barley.rlm.df, aes(x = .resid, y = variety)) +
  geom_point() + facet_wrap(~ site + year) +
  theme(axis.text.y = element_text(size=7))
```



Again, not much pattern, which is good. Note that Morris looks unremarkable now: including the interaction gets rid of the anomaly.

8.3.2 Was Morris a mistake?

The best way to find out is to look at more data. In fact, (incomplete) data from 1927 to 1936 is available in the file `minnesota.barley.yield.txt`. Sounds like a good topic for a problem set, doesn't it…

Chapter 9

Count data

Optional reading: Gelman & Hill pp. 110–116, 118–119.

9.1 Stop and frisk

Gelman and Hill have data on police stops in New York City in 1998–1999, during Giuliani's mayoralty. There have been accusations that some ethnic groups have been stopped at rates not justified by either their arrest rate or their location (as measured by precinct.) We'll model this data using (at first) **Poisson regression**, another form of GLM. In a standard Poisson regression, the response has a Poisson distribution with the *log* of the expected value given by a linear function of the predictors. In the single-variable case:

$$\log(E[Y|x]) = \beta_0 + \beta_1 x$$

The data, with noise added for confidentiality, is at

http://www.stat.columbia.edu/~gelman/arm/examples/police/frisk_with_noise.dat

The first few rows of this file are a description, so we tell R to skip these when reading the data.

```
frisk = read.table("http://www.stat.columbia.edu/~gelman/arm/examples/police/frisk_with_noise.dat
   skip = 6, header = TRUE)
nrow(frisk)
```

```
## [1] 900
```

```
summary(frisk)
```

```
## stops pop past.arrests precinct eth
## Min.  : 0 Min.  : 321 Min.  : 0.0 Min.  : 1 Min.  :1
## 1st Qu.: 26 1st Qu.: 6844 1st Qu.: 53.0 1st Qu.:19 1st Qu.:1
## Median : 72 Median : 18004 Median : 124.0 Median :38 Median
:2
## Mean : 146 Mean : 30105 Mean : 262.8 Mean :38 Mean :2
## 3rd Qu.: 173 3rd Qu.: 46669 3rd Qu.: 287.5 3rd Qu.:57 3rd
Qu.:3
## Max.  :1755 Max.  :184345 Max.  :2655.0 Max.  :75 Max.  :3
## crime
## Min.  :1.00
## 1st Qu.:1.75
## Median :2.50
## Mean :2.50
## 3rd Qu.:3.25
## Max.  :4.00
```

The data gives counts of police stops for all combinations of 75 precincts, three ethnicities of the person stopped (1 = black, 2 = Hispanic, 3 = white), and four types of crime (violent, weapons, property, and drug,) for a total of $75 \times 3 \times 4 = 900$ rows. The other two variables are population of the ethnic group within the precinct and the number of arrests of people in that ethnic group in that precinct for that type of crime in 1997.

Having numerical ethnicity is annoying, so recode. Based on prior knowledge that suggests that if there's discrimination, black and Hispanic people would likely be treated more similarly than white people, I'll make white the baseline level of the factor (though this is purely cosmetic.)

```
frisk$ethnicity = recode_factor(frisk$eth, `3` = "white", `1` = "black", `2` = "Hispan
```

Ensure we don't treat `precinct` as numeric:

```
frisk$precinct = factor(frisk$precinct)
```

To simplify matters, we'll ignore the type of crime, and aggregate the number of stops and past arrests over all four types.

```
frisk.sum = summarize(group_by(frisk, precinct, ethnicity), past.arrests = sum(past.ar
nrow(frisk.sum)
```

```
## [1] 225
```

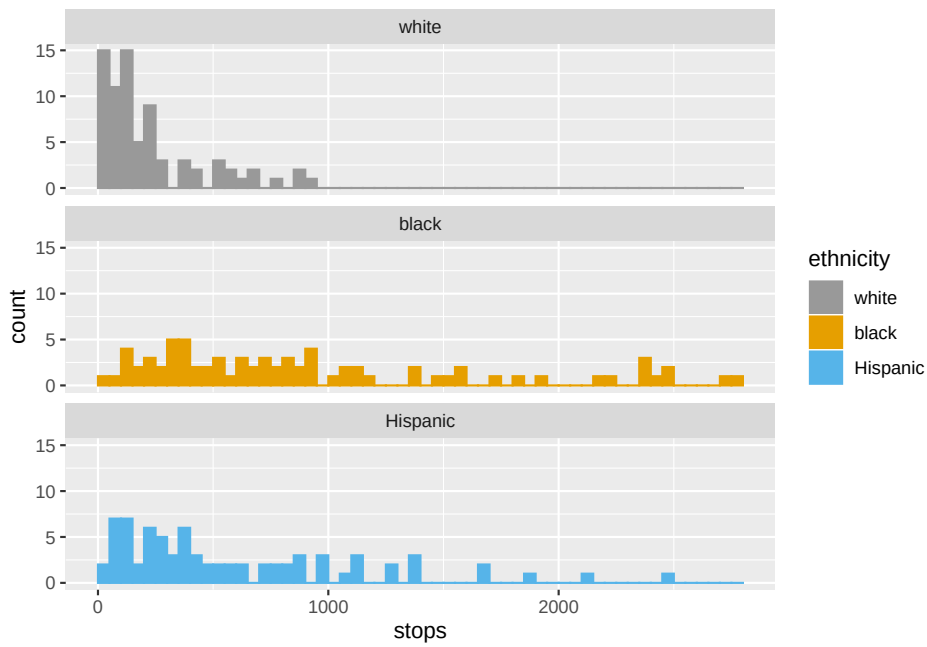
```
summary(frisk.sum)
```

```
##     precinct      ethnicity   past.arrests      stops
##  1      :  3   white   :75   Min.   :  16   Min.   :   7.0
##  2      :  3   black   :75   1st Qu.: 312   1st Qu.: 133.0
##  3      :  3   Hispanic:75   Median : 571   Median : 385.0
##  4      :  3                 Mean   :1051   Mean   : 584.1
```

```
##  5       :  3                    3rd Qu.:1467   3rd Qu.: 824.0
##  6       :  3                    Max.   :5667   Max.   :2771.0
##  (Other):207
```

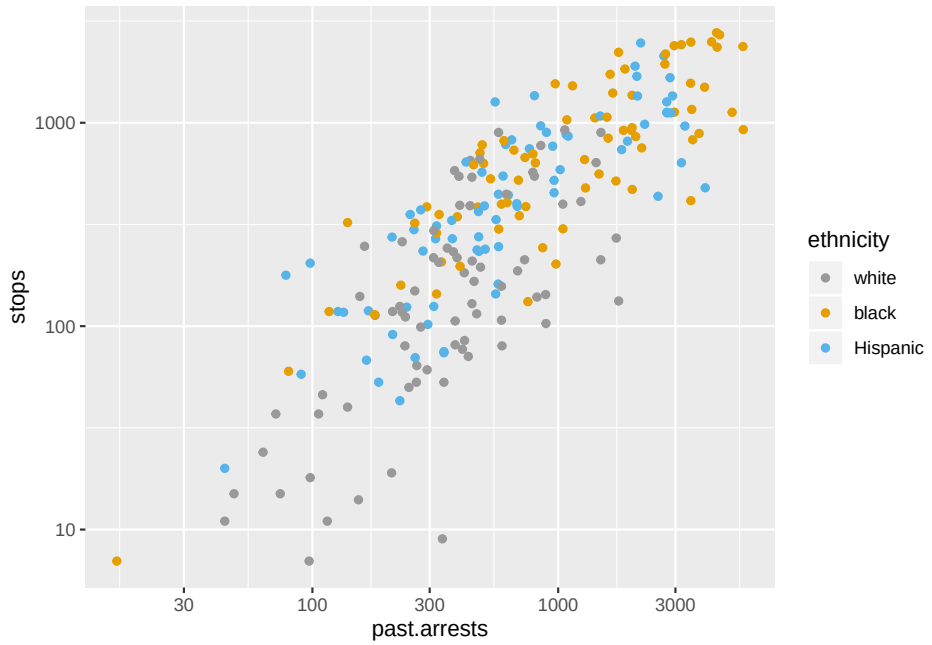
We now have 225 rows (75 precincts × 3 ethnic groups.) Let's first draw some pictures.

```
ggplot(frisk.sum, aes(x = stops, color = ethnicity, fill = ethnicity)) +
  geom_histogram(breaks = seq(0, 2800, 50)) + facet_wrap(~ethnicity, ncol = 1) +
  scale_color_manual(values = cb_palette) +
  scale_fill_manual(values = cb_palette)
```



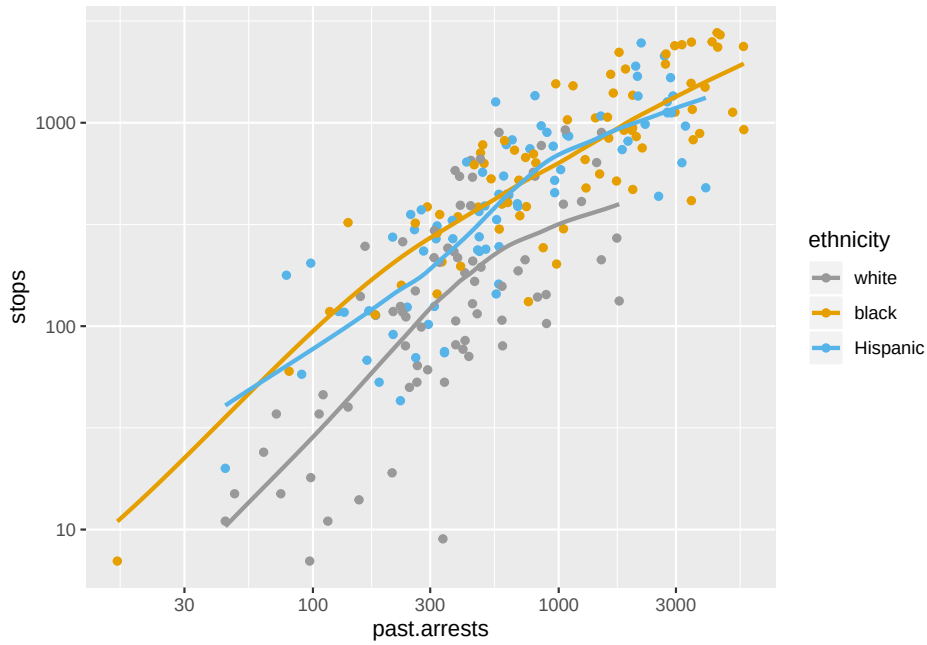
Quite clearly, the distributions of stops for black and Hispanic people are very different from the distribution for white people, though there may be multiple explanations for this. Let's look at the relationship of stops with past arrests. Because of skewness, we log both variables.

```
ggplot(frisk.sum, aes(x = past.arrests, y = stops, color = ethnicity)) +
  geom_point() +
  scale_x_log10() + scale_y_log10() +
  scale_color_manual(values = cb_palette)
```



There's certainly a relationship. The question is whether the relationship between the two variables is sufficient to explain the differences between the stops of the three ethnic groups. You could get at this just by adding smoothers for the three groups:

```
ggplot(frisk.sum, aes(x = past.arrests, y = stops, group = ethnicity, color = ethnicity
  geom_point() +
  geom_smooth(method.args = list(degree = 1), se = FALSE) +
  scale_x_log10() + scale_y_log10() +
  scale_color_manual(values = cb_palette)
```



Since this is an important topic, however, we should be a bit more careful and construct a model.

9.1.1 Poisson regression

We'll start off with a Poission regression model that's much too simple, and build up to a more useful one.

The simplest model just treats each number of stops as a realization of a Poisson random variable.

```
constant.glm = glm(stops ~ 1, family = poisson, data = frisk.sum)
summary(constant.glm)
```

```
##
## Call:
## glm(formula = stops ~ 1, family = poisson, data = frisk.sum)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -33.049  -22.552   -8.788    9.343   65.227
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 6.370053   0.002758    2309   <2e-16 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 123333  on 224  degrees of freedom
## Residual deviance: 123333  on 224  degrees of freedom
## AIC: 125041
##
## Number of Fisher Scoring iterations: 5
```

By now you might be sick of all the cruft that gets displayed when we use `summary()` on a GLM. Let's use Gelman et al.'s `display()` function in package `arm` instead.

```r
# install.packages(arm)
library(arm)
display(constant.glm)
```

```
## glm(formula = stops ~ 1, family = poisson, data = frisk.sum)
## coef.est coef.se
## (Intercept) 6.37 0.00
## ---
## n = 225, k = 1
## residual deviance = 123332.5, null deviance = 123332.5
(difference = 0.0)
```

This pares away most of the low value information. We see the coefficent estimate (on the log scale) is 6.37, which gives $e^{6.37} = 584$ on the original scale. That is, the number of stops for each ethnic group within each precinct is modeled as a random variable with distribution

$$\text{Poisson}(584).$$

The other number to keep track of is the (residual) *deviance*, which is a version of the log-likelihood rescaled to be equivalent to the residual sum of squares in the linear regression cases. (As you recall from S350/520, theoretical statisticians used to really like sums of squares.) Low deviance is good, as long as you're not overfitting. In particular, every time you add a degree of freedom, you should expect to reduce the deviance by 1 if you're just adding random noise. If the degree of freedom will help with (out-of-sample) prediction, then as a rough guide, it should reduce the deviance by more than 2. So if you're worried about overfitting when you fit a more complex model, you should want to reduce the deviance by more than twice the increase in the model degrees of freedom. (Note that if you're using this as a decision criterion, this is equivalent to using AIC.) However, this can lead to a high false positive rate, so for exploratory purposes you might want to be a little more strict than this.

Our constant model is obviously inadequate. We might, for example, think that the number of stops for an ethnic groups in a precinct should be proportional to the number of arrests for that ethnicity-precinct (though this is controversial.) In a GLM, we can model this using an **offset**:

```
offset.glm = glm(stops ~ 1, family = poisson, offset = log(past.arrests),
  data = frisk.sum)
display(offset.glm)
```

```
## glm(formula = stops ~ 1, family = poisson, data = frisk.sum,
## offset = log(past.arrests))
## coef.est coef.se
## (Intercept) -0.59 0.00
## ---
## n = 225, k = 1
## residual deviance = 46120.3, null deviance = 46120.3
(difference = 0.0)
```

Since the linear predictor is on the log scale, the offset also has to be logged. This gives the following model for each precinct/race combination:

$$\log[E(\text{stops}|\text{past arrests})] = -0.59 + \log(\text{past arrests})$$

or (taking the exponential of both sides)

$$E(\text{stops}|\text{past arrests}) = e^{-0.59+\log(\text{past arrests})} = 0.56 \times \text{past arrests}$$

To check this, we look at the predicted number of stops for precinct/race combinations with 10, 100, and 1000 past arrests respectively:

```
predict(offset.glm, newdata = data.frame(past.arrests = c(10, 100, 1000)),
  type = "response")
```

```
##          1          2          3
##   5.555955  55.559548 555.595483
```

Our model has a much lower deviance than the constant model, so we've improved the fit by a lot.

Now we want to see what happens if we add ethnic group as a predictor. Ethnic group is categorical, so we use it as a factor.

```
eth.glm = glm(stops ~ ethnicity, family = poisson, offset = log(past.arrests),
  data = frisk.sum)
display(eth.glm)
```

```
## glm(formula = stops ~ ethnicity, family = poisson, data =
frisk.sum,
## offset = log(past.arrests))
```

```
## coef.est coef.se
## (Intercept) -0.75 0.01
## ethnicityblack 0.16 0.01
## ethnicityHispanic 0.23 0.01
## ---
## n = 225, k = 3
## residual deviance = 45437.4, null deviance = 46120.3
(difference = 682.9)
```

Note that "past arrests" doesn't have a coefficient: the model assumes that expected stops are proportional to past arrests (where the constant of proportionality may depend on other stuff.) The deviance has dropped substantially again. On the log scale, we have additive terms for the offset and for ethnicity (relative to black, which is taken as the baseline due to alphabetical order.) On the original scale, the terms are multiplicative, and we can combine the offset and ethnicity terms to get a coefficient for each ethnicity. That is, the model is now

$$E(\text{stops}) = \text{multiplier for ethnic group} \times \text{past arrests}$$

where the multipliers are

```r
eth.co = coefficients(eth.glm)
multipliers = exp(c(eth.co[1], eth.co[1] + eth.co[2], eth.co[1] + eth.co[3]))
print(multipliers)
```

```
## (Intercept) (Intercept) (Intercept)
##   0.4725238   0.5553894   0.5957836
```

for black, Hispanic, and white respectively. We can check this using `predict()`:

```r
predict(eth.glm, newdata = data.frame(past.arrests = 1000,
  ethnicity = c("black", "Hispanic", "white")), type = "response")
```

```
##        1        2        3
## 555.3894 595.7836 472.5238
```

So far we have shown that black and Hispanic people were stopped at a proportionately higher fraction of their arrest rate compared to white people. However, as the data isn't from a randomized experiment, there may be confounding. For example, black and Hispanic people generally live in precincts with higher stop rates. (Whether this is in itself evidence of bias is again, controversial.) Since this is exploratory work, we won't attempt to prove cause-and-effect, but we'll see whether we can simply explain the results by including a precinct variable. If we can, then the NYPD might argue that minorities are only stopped more often because they, perhaps coincidentally, tend to live in precincts with high stop rates.

```
precinct.glm = glm(stops ~ ethnicity + precinct, family = poisson,
  offset = log(past.arrests), data = frisk.sum)
```

We won't print out the full results because we now have a coefficient for each precinct. Let's just first check the deviance has gone down meaningfully:

```
deviance(eth.glm)
```

```
## [1] 45437.35
```

```
deviance(precinct.glm)
```

```
## [1] 3427.14
```

Now look at the first few coefficients (and their standard errors):

```
coefficients(summary(precinct.glm))[1:6, 1:2]
```

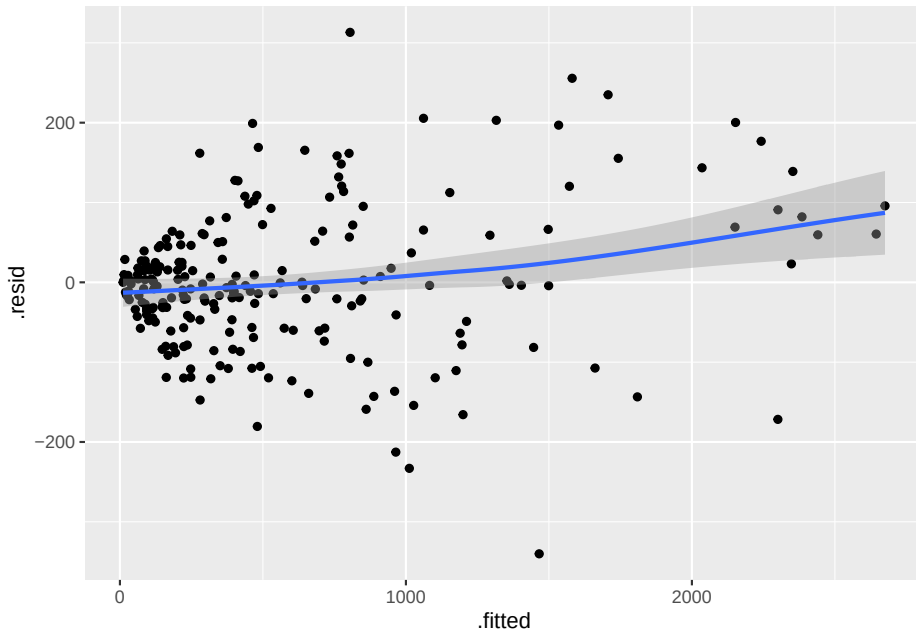
```
##                     Estimate  Std. Error
## (Intercept)       -1.7978692 0.051520069
## ethnicityblack     0.4190012 0.009434996
## ethnicityHispanic  0.4291892 0.009587547
## precinct2         -0.1490496 0.074030344
## precinct3          0.5599550 0.056758425
## precinct4          1.2106361 0.057548994
```

After controlling for precinct, the differences between the white and minority coefficients becomes even bigger.

9.1.2 Checking the model

We first plot the residuals against the fitted values on the response (original) scale, and see what happens.

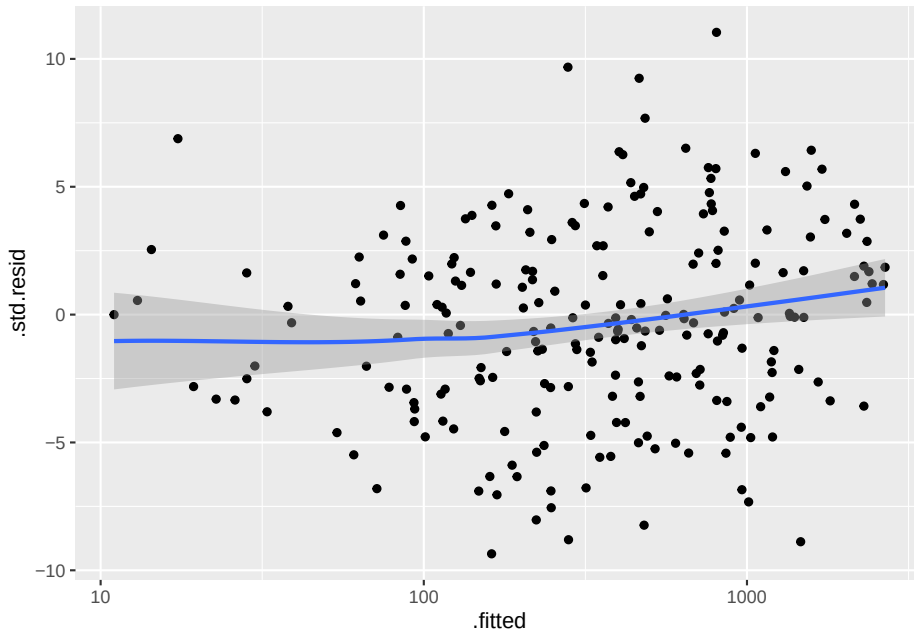
```
precinct.fitted = fitted.values(precinct.glm)
precinct.resid = residuals(precinct.glm, type = "response")
precinct.glm.df = data.frame(frisk.sum, .fitted = precinct.fitted,
  .resid = precinct.resid)
ggplot(precinct.glm.df, aes(x = .fitted, y = .resid)) + geom_point() +
  geom_smooth(span = 1, method.args = list(degree = 1))
```



The smoother we added isn't flat, but that could just be because the residuals are heteroskedastic: they spread out dramatically. The heteroskedasticity is not a bug: Poissons are supposed to be heteroskedastic. Recall that a Poisson(λ) random variable has variance λ and standard deviation $\sqrt{\lambda}$. So the typical size of the residuals should go up as the square root of the fitted value.

To hopefully remove this effect, we create **standardized residuals** by dividing the raw residuals by the square root of the fitted value. We plot these against the log fitted values to reduce the distortions caused by skewness.

```
precinct.std.resid = precinct.resid / sqrt(precinct.fitted)
precinct.glm.df$.std.resid = precinct.std.resid
ggplot(precinct.glm.df, aes(x = .fitted, y = .std.resid)) +
  geom_point() +
  geom_smooth(span = 1, method.args = list(degree = 1)) +
  scale_x_log10()
```



This is better, though far from perfect. There's still some nonlinearity left in the smoother, though the amount is relatively small. If prediction was the sole goal, we could think about transformations other than the log for the linear predictor scale, though this would make the interpretation much hazier.

9.1.3 Overdispersion

If you care about more than just the conditional expectation, however, we find a bigger problem. If the Poisson model were correct, the standardized residuals should be on a similar scale to the standard normal – that is, the vast majority should be within ± 2. From the previous graph, that's clearly not the case.

We need to measure the **overdispersion** in the data. We could do a formal χ^2 test for overdispersion, but instead, let's calculate the typical size of the squared residuals. (When we "average", we divide the sum by the residual degrees of freedom.) If the Poisson model is correct, this should be close to 1. If it's much more than 1, we need a better model.

```
overdispersion = sum(precinct.std.resid^2) / df.residual(precinct.glm)
overdispersion
```

```
## [1] 21.88505
```

This is much more than 1. In fact, this happens most of the time with count data – the data is usually more dispersed than the Poisson model.

9.1.4 How bad is it?

We know there are problems with our model. But are they so bad that we can't draw conclusions from it?

One simple way of checking is to simulate a fake set of data, and see if it closely resembles the actual set. For a Poisson model, this is easy. We know according to the model, each observation is a realization of a Poisson random variable, whose parameter is given by the fitted value. Then we can use `rpois()` to do simulation and do numerical summaries and plots. (Of course, we could repeat the simulation multiple times if necessary, but we'll just do it once.)

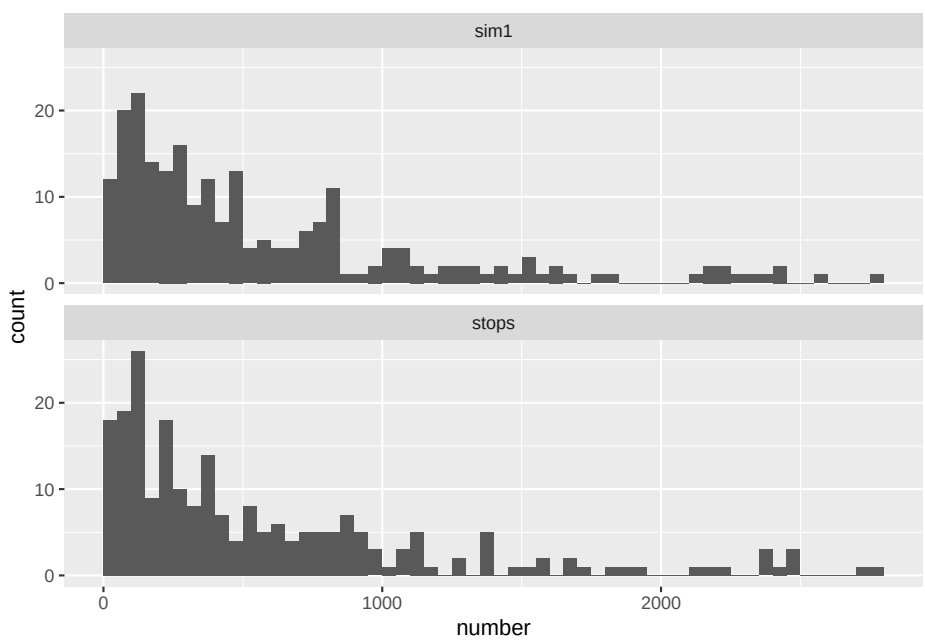
```
sim1 = rpois(nrow(frisk.sum), lambda = fitted.values(precinct.glm))
summary(frisk.sum$stops)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     7.0   133.0   385.0   584.1   824.0  2771.0
```

```
summary(sim1)
```

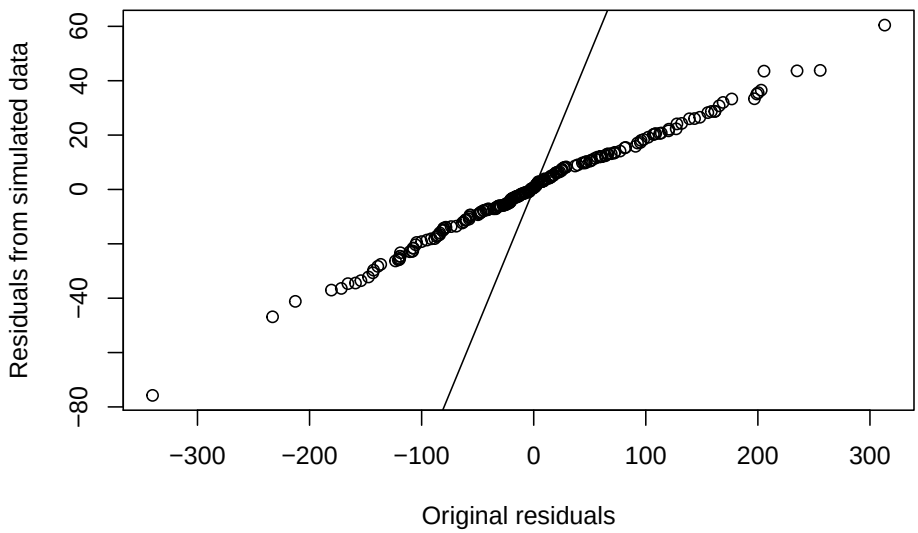
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    10.0   166.0   391.0   585.5   803.0  2786.0
```

```
sim.df = data.frame(frisk.sum, sim1)
sim.long = sim.df %>% gather(type, number, stops:sim1)
ggplot(sim.long, aes(x = number)) + geom_histogram(breaks = seq(0, 2800, 50)) +
  facet_wrap(~type, ncol = 1)
```



If we look at the histograms, there doesn't seem to be much difference. But what happens if we fit a model to the simulated data and look at its residuals? We'll find these and do a two-sample QQ plot of them against the original residuals (out of laziness we'll just draw the plot in base R.)

```
precinct.sim = glm(sim1 ~ ethnicity + precinct, family = poisson,
  offset = log(past.arrests), data = sim.df)
qqplot(residuals(precinct.glm, type = "response"),
       residuals(precinct.sim, type = "response"),
       xlab = "Original residuals",
       ylab = "Residuals from simulated data")
abline(0, 1)
```



If the model were correct, this QQ plot should be close to a line through the origin with slope 1. It ain't.

The simulation here is overkill, since we understand the Poisson fairly well and already know the data is overdispersed. However, the more complicated your model gets, the more useful this kind of simulation is as a sanity check.

9.1.5 Fixing overdispersion

The quickest fix is to use the **quasipoisson** family instead of the Poisson.

```
precinct.quasi = glm(stops ~ ethnicity + precinct, family = quasipoisson,
  offset = log(past.arrests), data = frisk.sum)
coefficients(summary(precinct.quasi))[1:6, 1:2]
```

```
##                     Estimate Std. Error
## (Intercept)       -1.7978692 0.24101846
## ethnicityblack     0.4190012 0.04413830
```

```
## ethnicityHispanic  0.4291892 0.04485195
## precinct2         -0.1490496 0.34632483
## precinct3          0.5599550 0.26552425
## precinct4          1.2106361 0.26922265
```

Note that the coefficients look the same as they were in the standard Poisson case. However, their standard errors have been inflated by the square root of their overdispersion. We can confirm that the fitted values haven't changed:

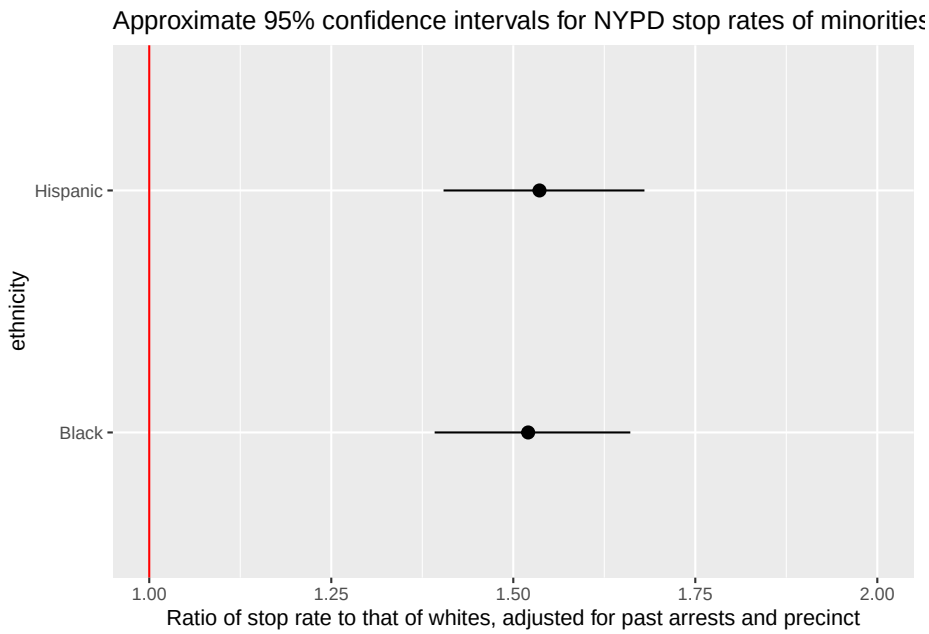
```
quasi.fitted = fitted.values(precinct.quasi)
summary(quasi.fitted - precinct.fitted)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0       0       0       0       0       0
```

So the quasipoission doesn't change the fit, only the variance and the standard errors.

We now back-transform to get intervals for the stop rates of blacks and Hispanics relative to whites, after adjusting for arrest rates and precinct.

```
eth.co = coefficients(summary(precinct.quasi))[1:3, 1:2]
ethnicity = c("Black", "Hispanic")
estimate = exp(eth.co[2:3, 1])
lower = exp(eth.co[2:3, 1] - 2 * eth.co[2:3, 2])
upper = exp(eth.co[2:3, 1] + 2 * eth.co[2:3, 2])
eth.co.df = data.frame(ethnicity, estimate, lower, upper)
ggplot(eth.co.df, aes(x = ethnicity, y = estimate, ymin = lower, ymax = upper)) +
  geom_pointrange() + ylim(1, 2) +
  geom_abline(intercept = 1, slope = 0, color = "red") +
  ylab("Ratio of stop rate to that of whites, adjusted for past arrests and precinct")
  ggtitle("Approximate 95% confidence intervals for NYPD stop rates of minorities") +
  coord_flip()
```



The confidence intervals don't include 1. This would be consistent with a hypothesis of bias against minorities, though we should think very carefully about other confounding variables before drawing a firm conclusion (e.g. type of crime, which we ignored.) You should check your model very thoroughly. A definitive answer here requires subject matter knowledge in conjunction with statistics.

9.1.6 Other models

There are lots of alternative approaches:

- **Negative binomial regression** is an alternative to the quasipoisson when the count data is overdispersed. It has the advantage of being a proper probabilistic model, unlike the quasipoisson.

```
library(MASS)
precinct.nb = glm.nb(stops ~ ethnicity + precinct + offset(log(past.arrests)), data = frisk.sum)
coefficients(summary(precinct.nb))[1:6, 1:2]
```

```
##                     Estimate Std. Error
## (Intercept)       -1.7130435 0.14413514
## ethnicityblack     0.4858489 0.03932041
## ethnicityHispanic  0.4944202 0.03949455
## precinct2         -0.2384622 0.20112759
## precinct3          0.5809872 0.19482896
## precinct4          0.9599195 0.19555008
```

- Nonparametric approaches like GAM can give you a better fit for the

conditional expectation, at the cost of making inference much more complicated. This isn't useful here, as apart from the offset, everything is categorical.

- A (frequentist) **mixed model** with precinct treated as a random intercept has appeal here, because of the large number of precincts. Such a model can deal with overdispersion as well as regularize the estimates for the precincts.

```
library(lme4)
precinct.glmer = glmer(stops ~ ethnicity + (1 | precinct), family = poisson,
  offset = log(past.arrests), data = frisk.sum)
summary(precinct.glmer)
```

```
## Generalized linear mixed model fit by maximum likelihood
(Laplace
## Approximation) [glmerMod]
## Family: poisson ( log )
## Formula: stops ~ ethnicity + (1 | precinct)
## Data: frisk.sum
## Offset: log(past.arrests)
##
## AIC BIC logLik deviance df.resid
## 5690.4 5704.1 -2841.2 5682.4 221
##
## Scaled residuals:
## Min 1Q Median 3Q Max
## -9.3553 -2.9260 -0.2423 2.1736 11.0036
##
## Random effects:
## Groups Name Variance Std.Dev.
## precinct (Intercept) 0.3653 0.6044
## Number of obs: 225, groups: precinct, 75
##
## Fixed effects:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.864209 0.070260 -12.30 <2e-16 ***
## ethnicityblack 0.418300 0.009432 44.35 <2e-16 ***
## ethnicityHispanic 0.428675 0.009585 44.73 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
## (Intr) ethnct
## ethnctyblck -0.102
## ethnctyHspn -0.096 0.744
```

- A **Bayesian model** allows you to put prior distributions on parameters. Packages like `rstanarm` and `brms` make such models much, much easier to fit in R than they used to be. However, you should think pretty hard about priors before rushing in to fit such a model.

Chapter 10

Categorical responses

10.1 Ordered categorical responses: polr()

Optional reading: Gelman & Hill pp. 119–123.

With categorical regression, the main distinction is between models with **ordered** categories and models with **unordered** categories. Let's start with the ordered case.

10.1.1 Fake data: Grad school

Let's use the (simulated) data on the potential grad school application of college students at

http://stats.idre.ucla.edu/r/dae/ordinal-logistic-regression/

(I think this is the only fake data example left in this text. Let me know if you have a better real data set. I know there's the diamonds data set but everyone uses that one.) We'll read in the Stata data using `import()` in the `rio` package:

```
library(rio)
gradschool = import("https://stats.idre.ucla.edu/stat/data/ologit.dta")
summary(gradschool)
```

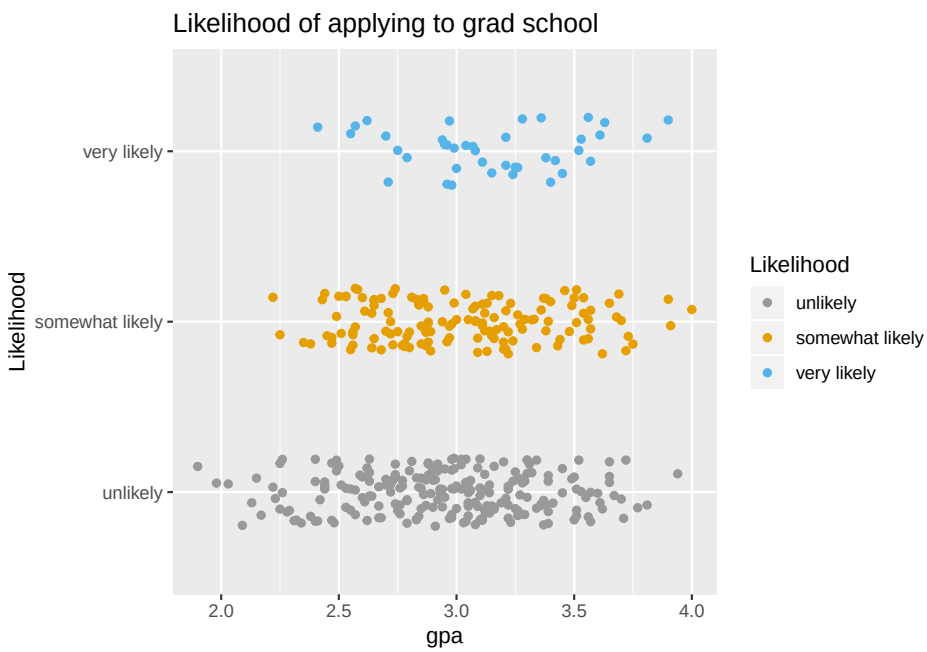
```
## apply pared public gpa
## Min.  :0.00 Min.  :0.0000 Min.  :0.0000 Min.  :1.900
## 1st Qu.:0.00 1st Qu.:0.0000 1st Qu.:0.0000 1st Qu.:2.720
## Median :0.00 Median :0.0000 Median :0.0000 Median :2.990
## Mean :0.55 Mean :0.1575 Mean :0.1425 Mean :2.999
## 3rd Qu.:1.00 3rd Qu.:0.0000 3rd Qu.:0.0000 3rd Qu.:3.270
## Max.  :2.00 Max.  :1.0000 Max.  :1.0000 Max.  :4.000
```

The data purports to be for 400 juniors asked how likely they are to apply to grad school. `apply` gives a student's intention to apply to grad school, where 0 means unlikely, 1 means somewhat likely, and 2 means very likely. That is, `apply` is an ordered categorical response. Let's explicitly it make a factor:

```
gradschool$Likelihood = recode_factor(gradschool$apply, `0` = "unlikely",
   `1` = "somewhat likely", `2` = "very likely")
```

We'll use `gpa` as our initial explanatory variable. Draw a jittered dot plot using a colorblind-friendly palette:

```
ggplot(gradschool, aes(x = gpa, y = Likelihood, color = Likelihood)) +
  geom_jitter(width = 0, height = 0.2) +
  ggtitle("Likelihood of applying to grad school") +
  scale_color_manual(values = cb_palette)
```

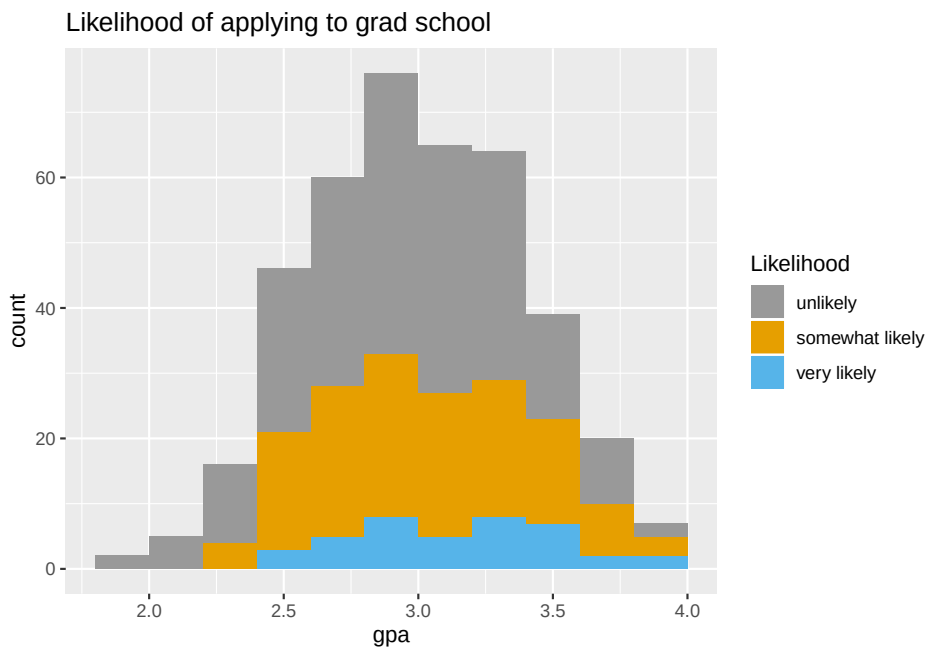


Most students are unlikely to apply to grad school. However, a higher GPA does mean a student is more likely to apply.

It's easy to get a sense of the conditional distribution of `gpa` given `Likelihood`. What we want, however, is the conditional distribution of `Likelihood` given `gpa`. We can get at this by discretizing `gpa`, drawing a histogram, then coloring it by the levels of `Likelihood`:

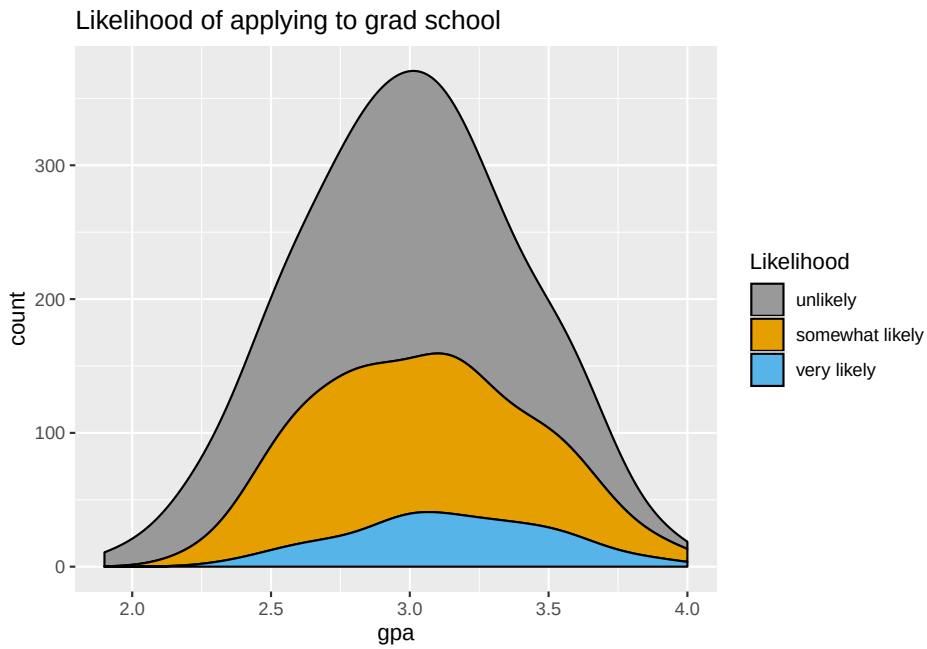
```
ggplot(gradschool, aes(x = gpa, fill = Likelihood)) +
  geom_histogram(breaks = seq(1.8, 4, 0.2)) +
  ggtitle("Likelihood of applying to grad school") +
```

```
  scale_fill_manual(values = cb_palette)
```



Discretizing is arbitrary. If you prefer smooth estimates at the cost of some transparency, you can use stacked density estimates instead:

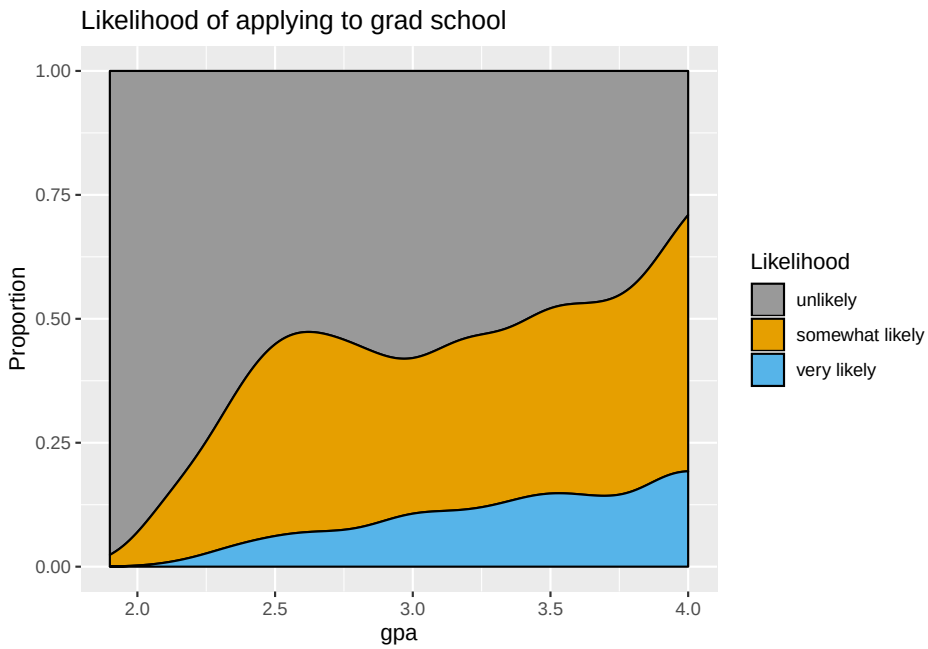
```
ggplot(gradschool, aes(x = gpa, ..count.., fill = Likelihood)) +
  geom_density(position = "stack") +
  ggtitle("Likelihood of applying to grad school") +
  scale_fill_manual(values = cb_palette)
```



The y-axis scale is a bit confusing here, since it's neither a true density nor a count. Instead, the total area is scaled to be equal to the number of observations (400.)

One question these graph don't answer directly: for a given GPA, what proportions of students are unlikely, somewhat likely, and very likely to apply to grad school? Here, instead of *joint* probabilities, we want *conditional* probabilities (conditional on GPA and possibly other variables.) We can use `position = "fill"` to plot conditional density estimates:

```
ggplot(gradschool, aes(x = gpa, ..count.., fill = Likelihood)) +
  geom_density(position = "fill") +
  ggtitle("Likelihood of applying to grad school") +
  scale_fill_manual(values = cb_palette) +
  ylab("Proportion")
```



What if we'd prefer to fit a model? The option we'll pursue is **proportional odds logistic regression**, fitted in R using the `polr()` function in `MASS`. Let's first fit the model, then explain what it means.

```
library(MASS)
gpa.polr = polr(Likelihood ~ gpa, data = gradschool)
library(arm)
display(gpa.polr)
```

```
## polr(formula = Likelihood ~ gpa, data = gradschool)
## coef.est coef.se
## gpa 0.72 0.25
## unlikely|somewhat likely 2.37 0.76
## somewhat likely|very likely 4.40 0.78
## ---
## n = 400, k = 3 (including 2 intercepts)
## residual deviance = 732.6, null deviance is not computed by
polr
```

The model gives us both a **linear predictor** (on a logit scale) and **cutpoints**. The linear predictor is

$$0.72 \times \text{GPA}$$

(Note that the form of the model fitted by `polr()` has no intercept.) To get deterministic predictions, we compare the linear predictor to the cutpoints. The

boundary between group 0 (unlikely) and group 1 (somewhat likely) is 2.37, while the boundary between group 1 and group 2 (very likely) is 4.4. (These can be extracted with `gpa.polr$zeta`.) Division tells us that this means a GPA below 3.28 gives a deterministic prediction of "unlikely", a GPA between 3.28 and 6.07 gives a prediction of "somewhat likely", and a GPA of above 6.07 gives a prediction of "very likely." Of course, no one has a GPA above 6.07.

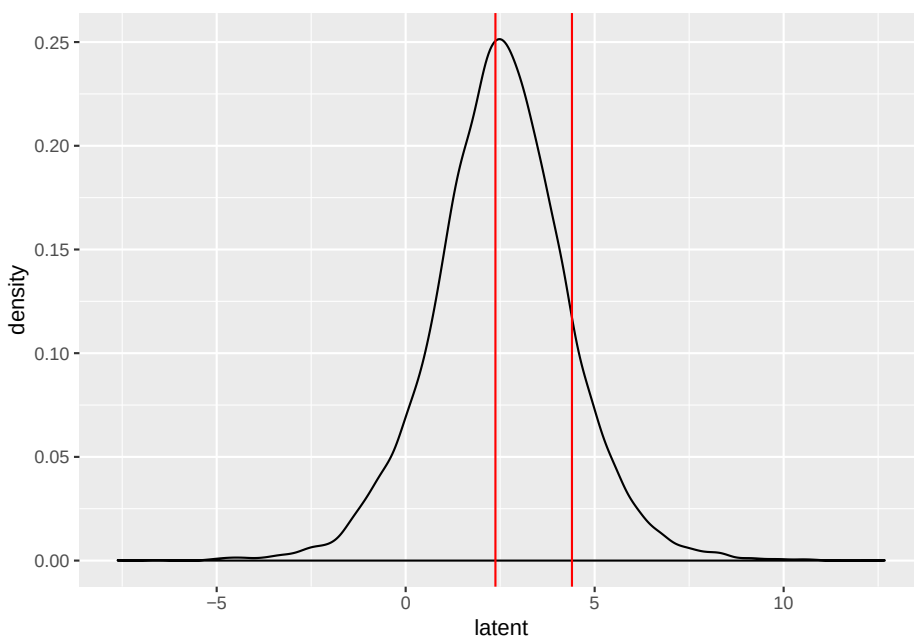
10.1.2 polr() and probability

Let's move on to probabilistic predictions. The errors in this model have a standard logistic distribution (i.e. with mean zero and scale parameter 1.) To make a prediction for an individual:

- Find the linear predictor based on their GPA;
- Add random logistic noise;
- Compare this "latent" variable to the cutpoints;
- Repeat lots of times.

Because we might not be used to the logistic distribution, let's first use simulation to estimate the distribution of the latent variable for a person with a 3.5 GPA. Their linear predictor is $0.725 \times 3.5 = 2.54$. We add logistic noise and see how often they fall in each cutpoint range.

```
prediction = coefficients(gpa.polr) * 3.5
latent = prediction + rlogis(10000)
ggplot(as.data.frame(latent), aes(x = latent)) + geom_density() +
  geom_vline(xintercept = gpa.polr$zeta, color = "red")
```



We see that the left and middle areas are bigger than the right area. This means that "unlikely" and "somewhat likely" are more probable than "very likely." Let's find the exact probabilities. The probability of being "unlikely" is

$$P(\beta x + \epsilon < z_{unlikely|somewhat})$$

where x is GPA, ϵ is standard logistic noise, and $z_{unlikely|somewhat}$ is the lower cutpoint. This is the same as

$$P(\epsilon < z_{unlikely|somewhat} - \beta x)$$

i.e., the probabilistic a standard logistic random variable is less than $z_{unlikely|somewhat} - \beta x$. We find logistic probabilities using the `inv.logit()` function in `boot`.

```
beta = coefficients(gpa.polr)
zeta = gpa.polr$zeta
library(boot)
inv.logit(zeta[1] - beta * 3.5)
```

```
## unlikely|somewhat likely
##                0.4595418
```

There's a 46% chance a person with a 3.5 GPA is "unlikely" to apply to grad school. Similarly, the probability they're "very likely" to apply to grad school is the probability a standard logistic random variable is *greater* than the difference between the second cutpoint and the linear predictor:

```
1 - inv.logit(zeta[2] - beta * 3.5)
```

```
## somewhat likely|very likely
##                   0.1343714
```

There's a 13% chance they're "very likely." That leaves a 41% chance they're "somewhat likely."

Now that we know what we're doing, we can just get these probabilities using `predict()`:

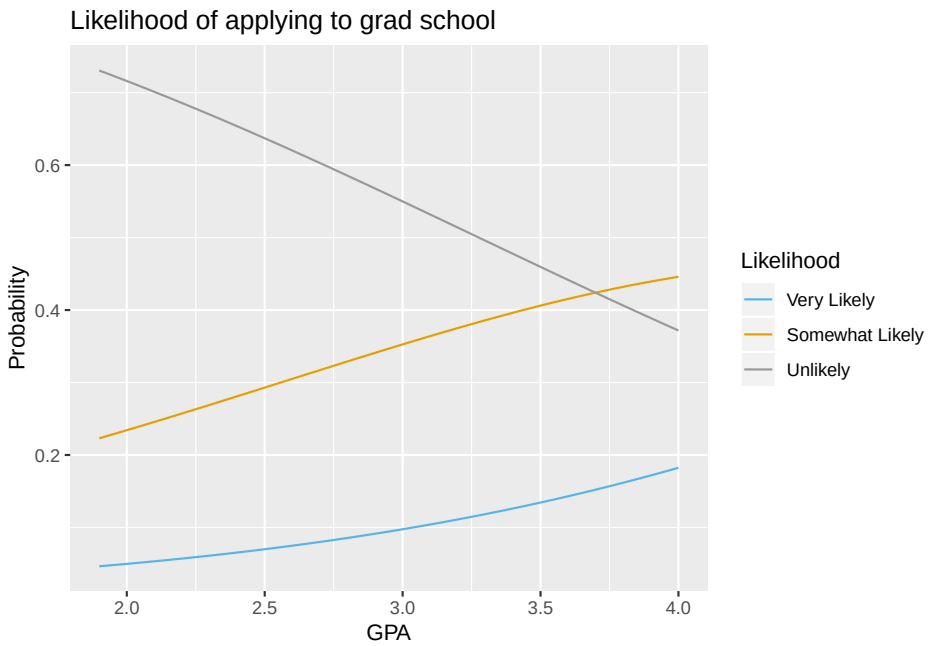
```
predict(gpa.polr, newdata=data.frame(gpa=3.5), type = "probs")
```

```
##        unlikely somewhat likely     very likely
##       0.4595418       0.4060868       0.1343714
```

10.1.3 Graphing and checking the model

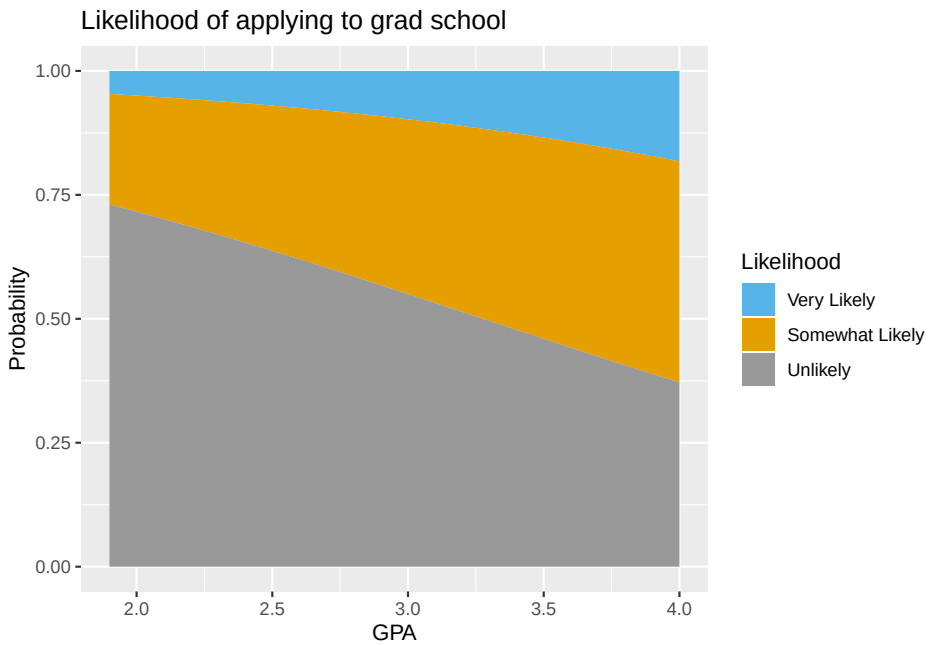
Let's display the fit as a function of GPA.

```
gpa = seq(min(gradschool$gpa), max(gradschool$gpa), 0.01)
grad.probs = predict(gpa.polr, newdata = data.frame(gpa), type = "prob")
grad.probs.df = data.frame(gpa, grad.probs)
names(grad.probs.df) = c("GPA", "Unlikely", "Somewhat Likely", "Very Likely")
grad.probs.long = grad.probs.df %>% gather(Likelihood, Probability, 2:4)
# Put levels in reverse order
grad.probs.long$Likelihood = factor(grad.probs.long$Likelihood,
  levels = c("Very Likely", "Somewhat Likely", "Unlikely"))
reverse.palette = c("#56B4E9", "#E69F00", "#999999", "#0072B2")
ggplot(grad.probs.long, aes(x = GPA, y = Probability, group = Likelihood, color = Likel
  geom_line() + ggtitle("Likelihood of applying to grad school") +
  scale_color_manual(values = reverse.palette)
```



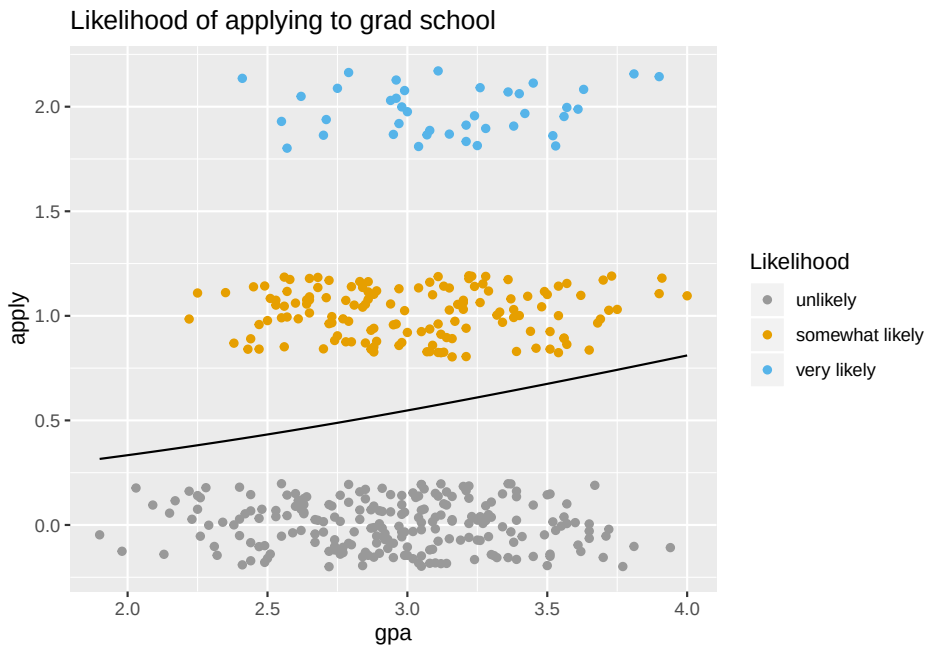
The probability of both "somewhat likely" and "very likely" increase with GPA, though "very likely" never gets very high. We can also stack the lines and use areas:

```
ggplot(grad.probs.long, aes(x = GPA, y = Probability, group = Likelihood, fill = Likeli
  geom_area() + ggtitle("Likelihood of applying to grad school") +
  scale_fill_manual(values = reverse.palette)
```



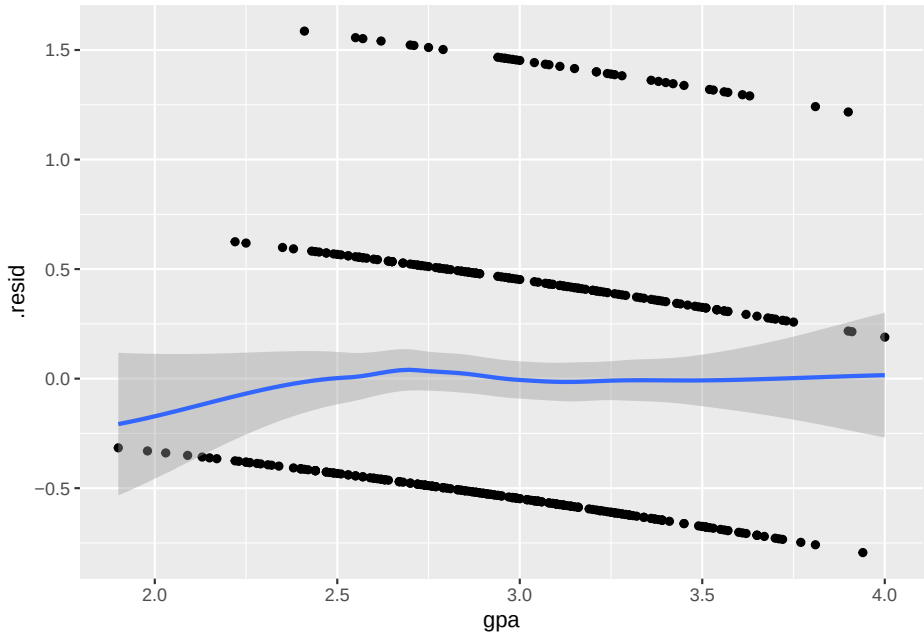
To check the fit, we could check every category separately. Instead we'll return to using `apply` as a quantitative variable: 0 for unlikely, 1 for somewhat likely, and 2 for very likely. That allows us to find means.

```
apply.fitted = fitted.values(gpa.polr)[,2] + 2 * fitted.values(gpa.polr)[,3]
apply.resid = gradschool$apply - apply.fitted
gpa.polr.df = data.frame(gradschool, .fitted = apply.fitted, .resid = apply.resid)
# Sort
gpa.polr.df = gpa.polr.df[order(gradschool$gpa),]
ggplot(gpa.polr.df, aes(x = gpa, y = apply)) +
  geom_jitter(width = 0, height = 0.2, aes(color = Likelihood)) +
  geom_line(aes(x = gpa, y = .fitted)) +
  ggtitle("Likelihood of applying to grad school") +
  scale_color_manual(values = cb_palette)
```



Now look at the residuals:

```
ggplot(gpa.polr.df, aes(x = gpa, y = .resid)) +
  geom_point() +
  geom_smooth(method.args = list(degree = 1))
```



This looks fine – there's a kink for GPAs below 2.25, but that could just be because there are few people with GPAs that low.

10.1.4 Multiple predictors

Let's now include two other variables in the model: `pared` is a binary variable indicating whether a parent has a grad degree, and `public` is a binary variable indicating whether the student goes to a public college.

```
grad.polr = polr(Likelihood ~ gpa + pared + public, data = gradschool)
display(grad.polr)
```

```
## polr(formula = Likelihood ~ gpa + pared + public, data =
gradschool)
## coef.est coef.se
## gpa 0.62 0.26
## pared 1.05 0.27
## public -0.06 0.30
## unlikely|somewhat likely 2.20 0.78
## somewhat likely|very likely 4.30 0.80
## ---
## n = 400, k = 5 (including 2 intercepts)
## residual deviance = 717.0, null deviance is not computed by
polr
```

The deviance has gone down by about 16 and the coefficients are in the direction in you'd expect – your parents going to grad school means it's more probable you'll go to grad school, while going to a public college means it's slightly less probable.

As for numerical responses, we can study the fit by using `expand.grid()` to get a data frame of explanatories and making predictions.

```
grad.grid = expand.grid(gpa = seq(min(gradschool$gpa), max(gradschool$gpa), 0.01),
    pared = 0:1, public = 0:1)
grad.predict = as.data.frame(predict(grad.polr, newdata = grad.grid, type = "probs"))
grad.polr.df = data.frame(grad.grid, grad.predict)
names(grad.polr.df) = c("gpa", "pared", "public", "Unlikely", "Somewhat Likely", "Very Likely")
```

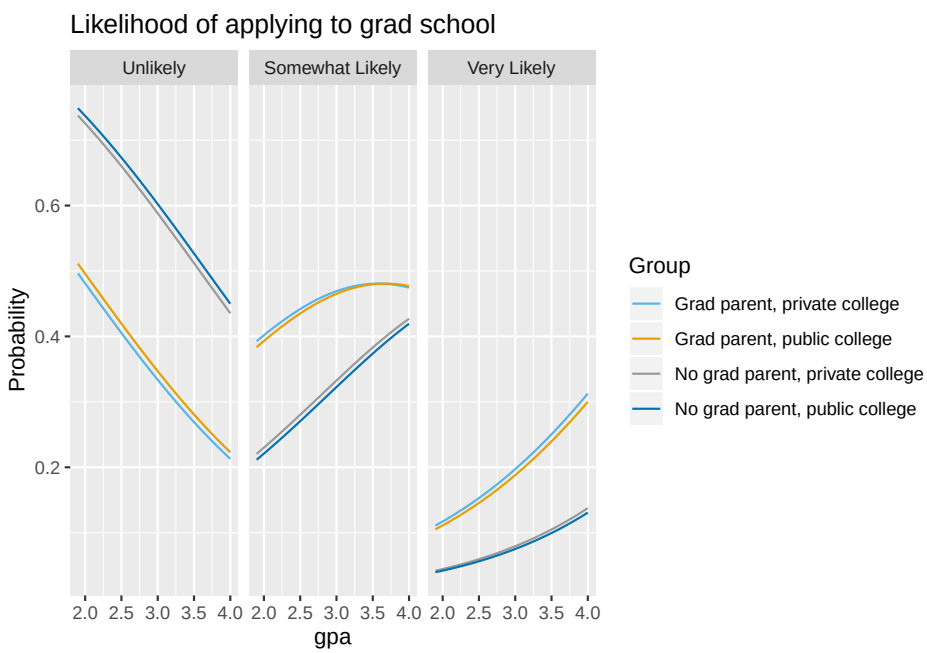
We'll append a new variable that gives the combination of `pared` and `public`.

```
group = 2 * grad.polr.df$pared + grad.polr.df$public
group[group == 3] = "Grad parent, public college"
group[group == 2] = "Grad parent, private college"
group[group == 1] = "No grad parent, public college"
group[group == 0] = "No grad parent, private college"
grad.polr.df$Group = factor(group)
```

There are a few ways to view this data frame, but probably the clearest is to

draw a panel for each category.

```
grad.polr.long = grad.polr.df %>% gather(Likelihood, Probability, 4:6)
grad.polr.long$Likelihood = factor(grad.polr.long$Likelihood,
    levels = c("Unlikely", "Somewhat Likely", "Very Likely"))
ggplot(grad.polr.long, aes(x = gpa, y = Probability, group = Group, color = Group)) +
  geom_line() +
  facet_grid(~Likelihood) +
  ggtitle("Likelihood of applying to grad school") +
  scale_color_manual(values = reverse.palette)
```



We see that private or public college makes almost no difference, so we should consider dropping that from the model.

10.2 Unordered categorical responses

Optional reading: Agresti, Categorical Data Analysis, section 8.1 (3rd edition pp. 294–297.)

With categorical regression, the main distinction is between models with **ordered** categories and models with **unordered** categories. Let's start with the ordered case.

10.2.1 Alligator food

What do alligators like to eat? Researcher captured 219 alligators in four Florida lakes, and categorized them by the primary contents of their stomach.

```
alligator = read.table("alligator.txt", header = TRUE)
head(alligator)
```

```
##      lake  sex  size    food count
## 1 Hancock male small    fish     7
## 2 Hancock male small  invert     1
## 3 Hancock male small reptile     0
## 4 Hancock male small    bird     0
## 5 Hancock male small   other     5
## 6 Hancock male large    fish     4
```

- `lake` gives the lake where the alligator was captured;
- `sex` is male or female;
- `size` is small or large;
- `food` is fish, invertebrate, reptile, bird, or other;
- `count` is how many of the 219 alligators had that combination of lake, sex, size, and food.

Check that there are $4 \times 2 \times 2 \times 5 = 80$ rows:

```
nrow(alligator)
```

```
## [1] 80
```

Check that there are 219 alligators:

```
sum(alligator$count)
```

```
## [1] 219
```

One annoying thing about categorical data is that different R functions often require the data to be in different formats. To get it over with, let's put the data in wide form. This will also let us print out a table with fewer rows that gives all the data.

```
alligator.wide = alligator %>% spread(food, count)
alligator.wide
```

```
##        lake    sex  size bird fish invert other reptile
## 1    George female large    0    8      1     1       0
## 2    George female small    0    3      9     1       1
## 3    George   male large    1    9      0     2       0
## 4    George   male small    2   13     10     2       0
## 5   Hancock female large    2    3      0     3       1
## 6   Hancock female small    2   16      3     3       2
## 7   Hancock   male large    1    4      0     2       0
```

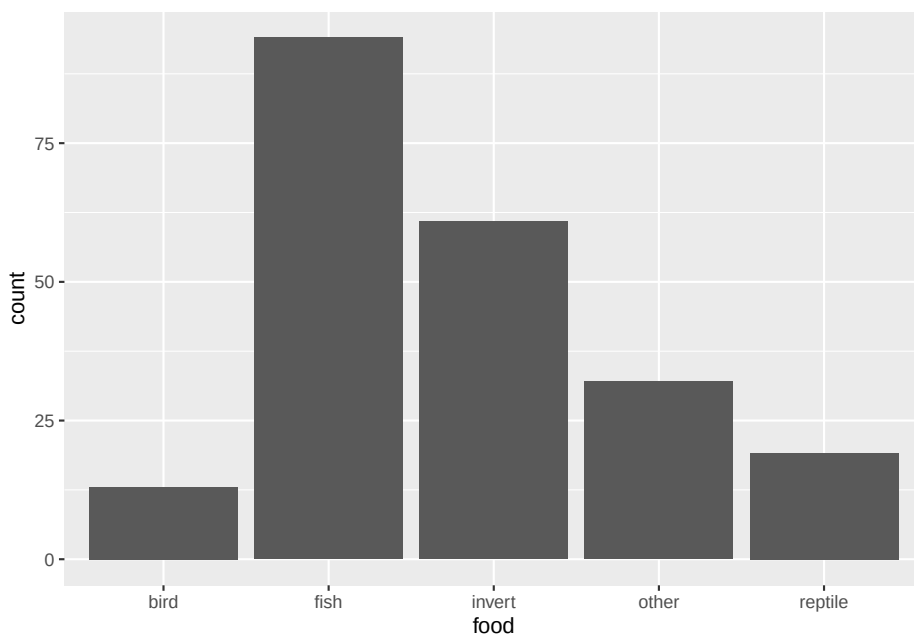
```
## 8   Hancock   male small    0    7      1     5       0
## 9  Oklawaha female large    1    0      1     0       0
## 10 Oklawaha female small    0    3      9     2       1
## 11 Oklawaha   male large    0   13      7     0       6
## 12 Oklawaha   male small    0    2      2     1       0
## 13 Trafford female large    0    0      1     0       0
## 14 Trafford female small    1    2      4     4       1
## 15 Trafford   male large    3    8      6     5       6
## 16 Trafford   male small    0    3      7     1       1
```

Just by looking at the numbers we see that fish are relatively popular, while birds and reptiles are unpopular. Our eventual goal will be to build a model that gives the probability an alligator prefers each type of food, based on the predictors we have.

10.2.2 Stacking vs. faceting

We can display the data using dots or bars. (I prefer the solidity of bars to points for graphs of raw count data, but this is idiosyncratic.)

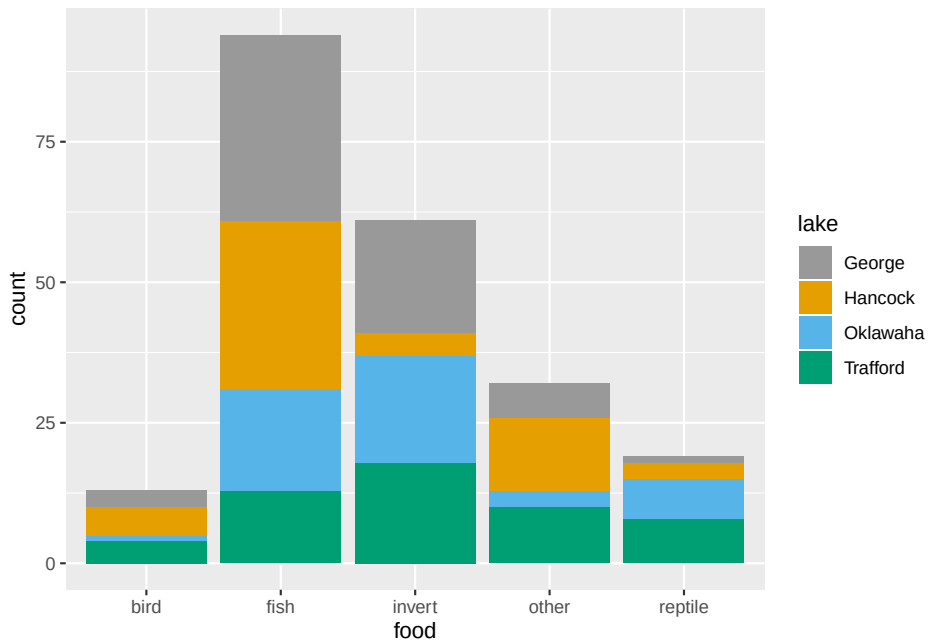
```
ggplot(alligator, aes(x = food, y = count)) + geom_bar(stat = "identity")
```



One way of including an additional categorical predictor is by stacking. This is easily achieved using `fill`:

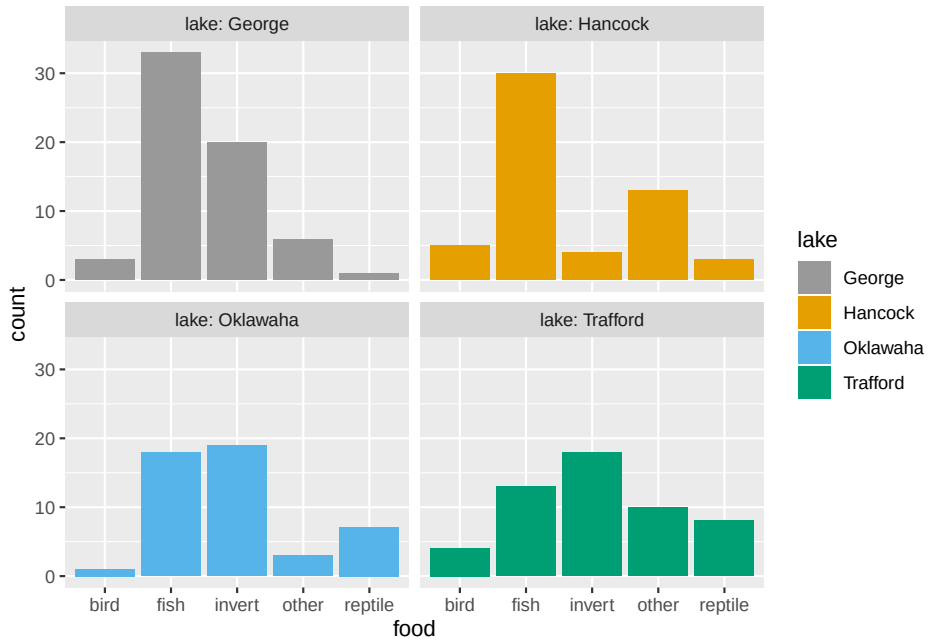
```
ggplot(alligator, aes(x = food, y = count, fill = lake)) +
  geom_bar(stat = "identity") +
```

```
  scale_fill_manual(values = cb_palette)
```



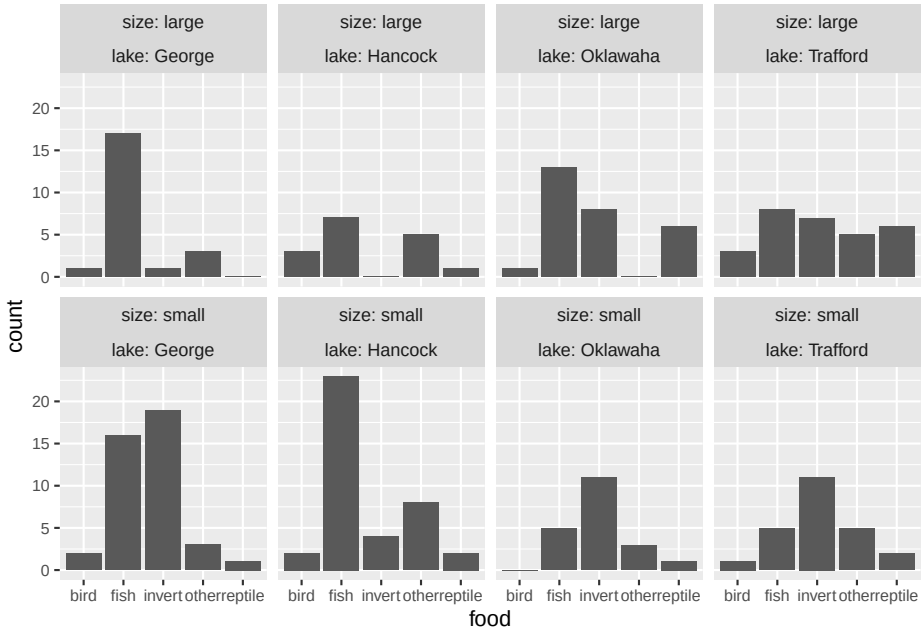
However, it's hard to clearly compare the distributions for the four lakes in the above graph. Faceting is usually better:

```
ggplot(alligator, aes(x = food, y = count, fill = lake)) +
  geom_bar(stat = "identity") +
  facet_wrap(~lake, labeller = label_both) +
  scale_fill_manual(values = cb_palette)
```



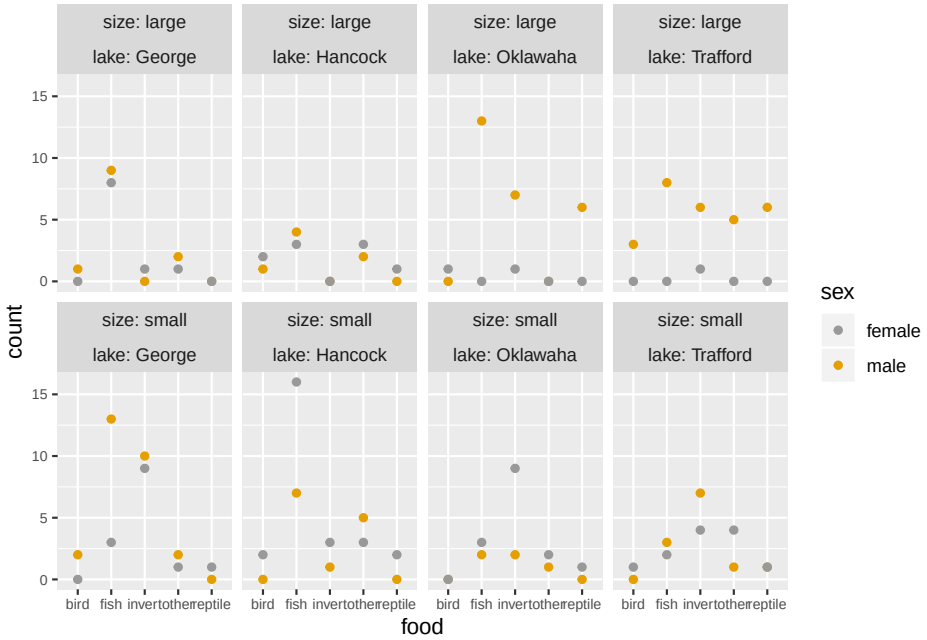
As in the continuous case, it's easy to facet two-ways, though you need to make sure the labeling is clear.

```
ggplot(alligator, aes(x = food, y = count)) +
  geom_bar(stat = "identity") +
  facet_wrap(~size + lake, ncol = 4, labeller = label_context) +
  theme(axis.text=element_text(size=8))
```



If you really want to get another variable in, you can switch back to dots and use color:

```
ggplot(alligator, aes(x = food, y = count, color = sex)) +
  geom_point() +
  facet_wrap(~size + lake, ncol = 4, labeller = label_context) +
  theme(axis.text = element_text(size = 7)) +
  scale_color_manual(values = cb_palette)
```

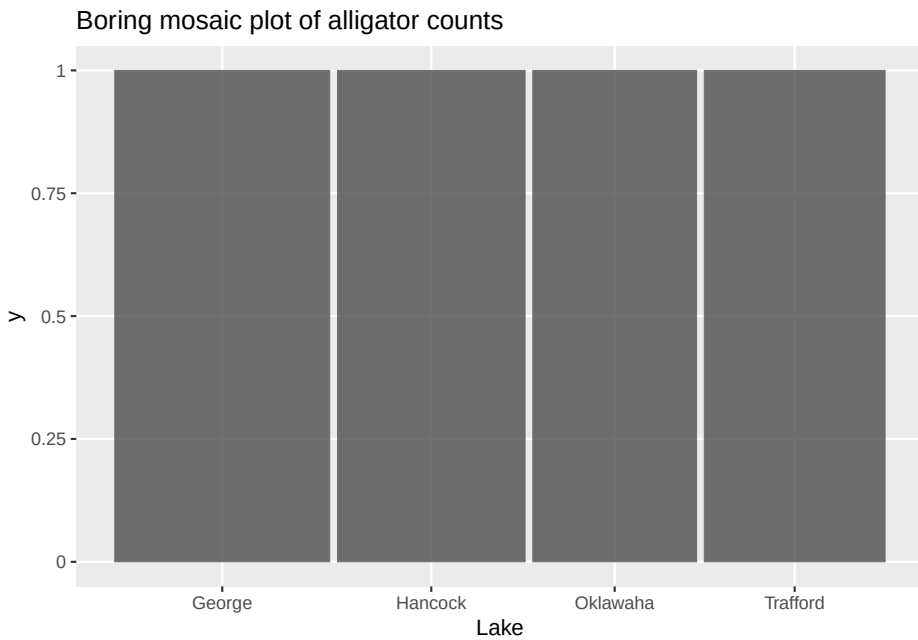


10.2.3 Mosaic plots

A (slightly controversial) alternative to the aobve is **mosaic plots**. Alternatives are the `mosaic()` function in package `vcd` and `geom_mosaic()` in the `ggmosiac` package; we'll use the latter because it allows us to combine mosaic plots with faceting.

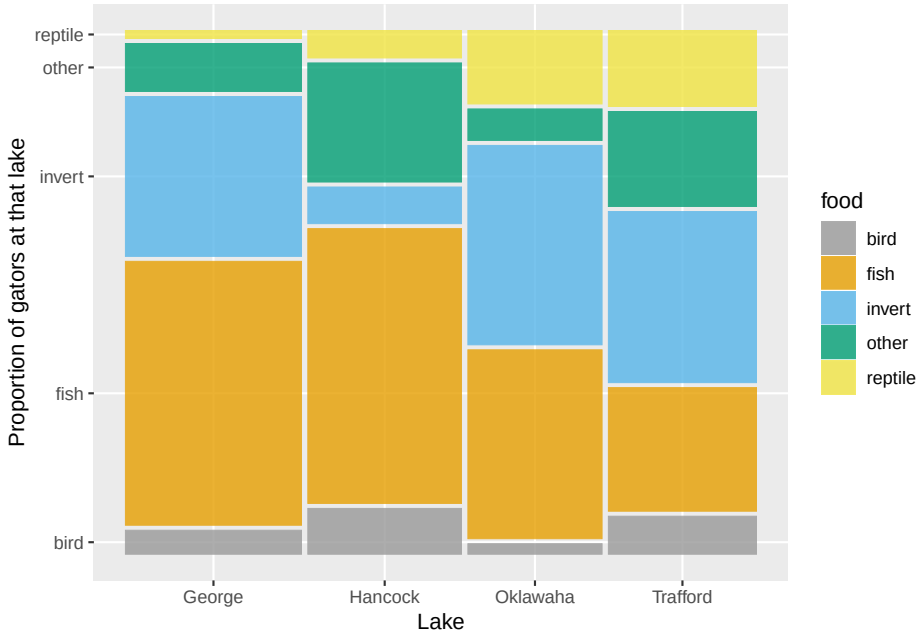
`ggmosaic` requires the data in "product" format (easily achieved with the `product()` function) and a `weight` variable. Here our weights are the counts of alligators in each combination of categories. Let's first draw a mosaic plot breaking up the total sample of alligators by lake.

```
# install.packages('ggmosaic')
library(ggmosaic)
ggplot(alligator) +
  geom_mosaic(aes(product(lake), weight = count)) +
  xlab("Lake") +
  ggtitle("Boring mosaic plot of alligator counts")
```



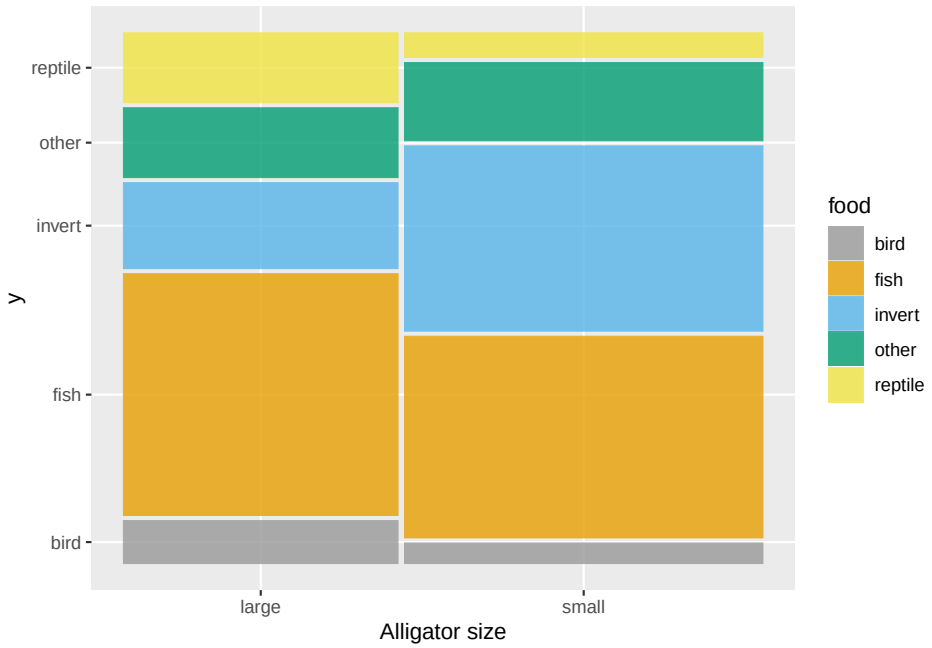
Here, the width of each block indicates the relative frequency of the variable. This isn't very interesting, but we can also divide up the area two ways:

```
ggplot(alligator) + geom_mosaic(aes(product(food, lake), weight = count, fill = food)) +
  xlab("Lake") + ylab("Proportion of gators at that lake") +
  scale_fill_manual(values = cb_palette)
```

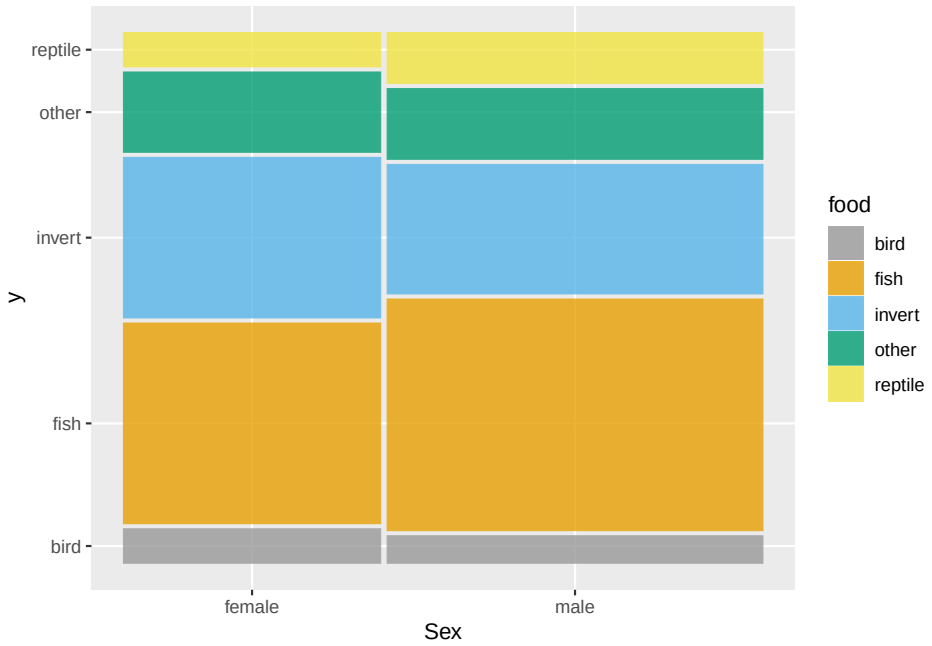


The above plot shows us the *conditional* distribution of each type of food, given the lake, as well as the joint relative frequency of each lake/food combination. The preferred types of food do seem to differ a lot by lake.

```
ggplot(alligator) +
  geom_mosaic(aes(product(food, size), weight = count, fill = food)) +
  xlab("Alligator size") +
  scale_fill_manual(values = cb_palette)
```



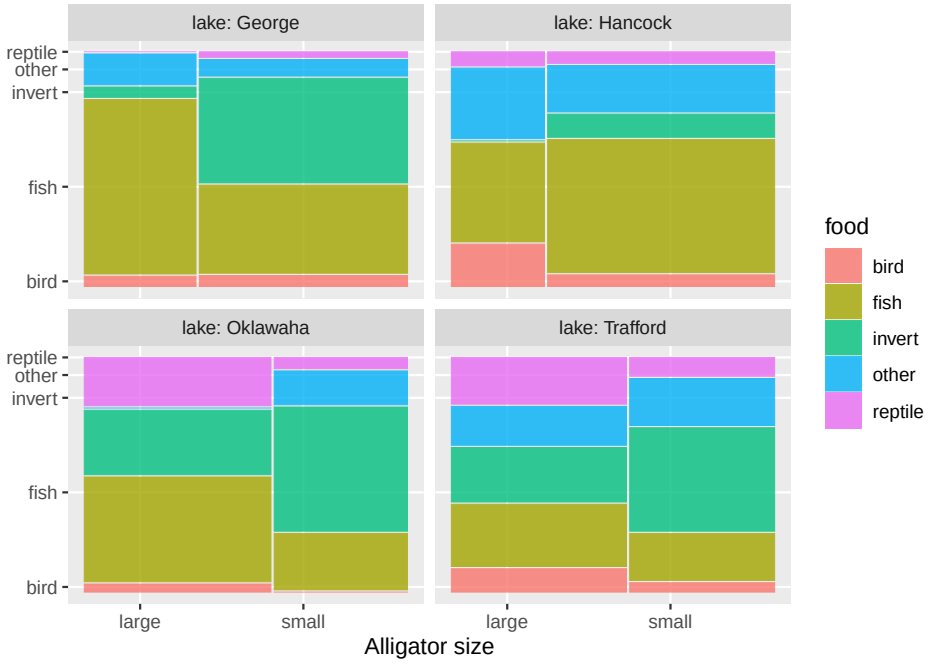

```
ggplot(alligator) +
  geom_mosaic(aes(product(food, sex), weight = count, fill = food)) +
  xlab("Sex") +
  scale_fill_manual(values = cb_palette)
```



We see that in the sample, there are more small gators than large ones, and more males than females. More importantly, the conditional distribution of food looks quite different between big and small gators, but quite similar comparing males and females.

We could keep on subdividing the bars in an attempt to look for interactions, but this is messy. Combining mosaics with faceting is preferable. In this case, because the data consists of a sample from each lake, it makes sense to facet by lake.

```
ggplot(alligator) +
  geom_mosaic(aes(product(food, size), weight = count, fill = food)) +
  facet_wrap(~lake, ncol = 2, labeller = label_both) +
  xlab("Alligator size") + ylab("")
```




```
  scale_fill_manual(values = cb_palette)
```

```
## <ggproto object: Class ScaleDiscrete, Scale, gg>
## aesthetics: fill
## axis_order: function
## break_info: function
## break_positions: function
## breaks: waiver
## call: call
## clone: function
## dimension: function
```

```
## drop: TRUE
## expand: waiver
## get_breaks: function
## get_breaks_minor: function
## get_labels: function
## get_limits: function
## guide: legend
## is_discrete: function
## is_empty: function
## labels: waiver
## limits: NULL
## make_sec_title: function
## make_title: function
## map: function
## map_df: function
## n.breaks.cache: NULL
## na.translate: TRUE
## na.value: NA
## name: waiver
## palette: function
## palette.cache: NULL
## position: left
## range: <ggproto object: Class RangeDiscrete, Range, gg>
## range: NULL
## reset: function
## train: function
## super: <ggproto object: Class RangeDiscrete, Range, gg>
## reset: function
## scale_name: manual
## train: function
## train_df: function
## transform: function
## transform_df: function
## super: <ggproto object: Class ScaleDiscrete, Scale, gg>
```

10.2.4 Multinomial regression

Let's fit a model using `lake` and `size` as predictors. For categorical responses, we want the conditional distribution given the predictors to be `multinomial`. I use the `vglm()` function (vector GLM) in package `VGAM` to fit multinomial regressions. The syntax is similar to that of `glm()` with family `multinomial`, except you need to specify a matrix of responses (one column for each category.) This can be done using `cbind()` with the data in wide format.

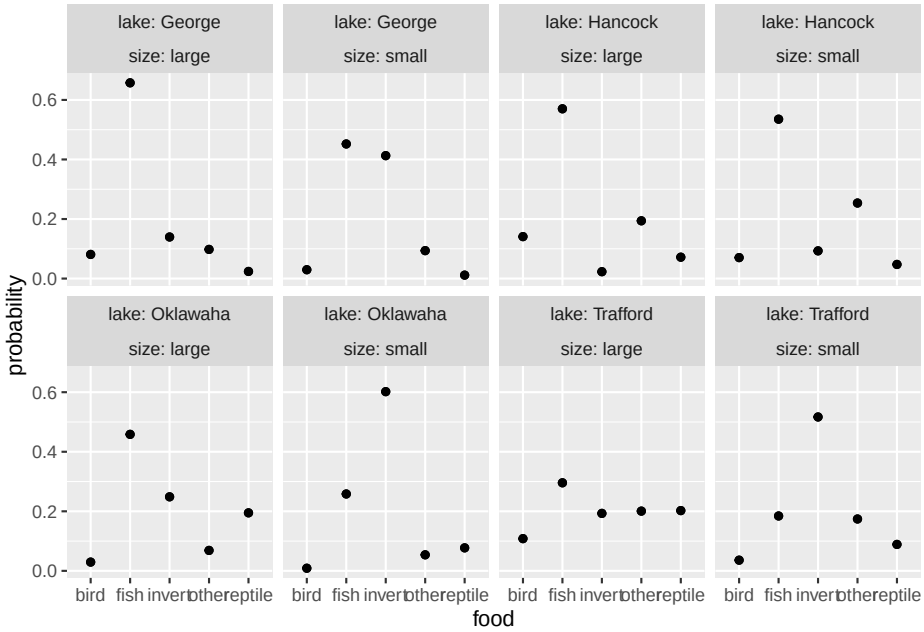
```r
# install.packages('VGAM')
library(VGAM)
```

```
alligator.mlogit = vglm(cbind(bird, fish, invert, other, reptile) ~ lake + size,
  family = multinomial, data = alligator.wide)
alligator.mlogit
```

```
##
## Call:
## vglm(formula = cbind(bird, fish, invert, other, reptile) ~
lake +
## size, family = multinomial, data = alligator.wide)
##
##
## Coefficients:
## (Intercept):1 (Intercept):2 (Intercept):3 (Intercept):4
lakeHancock:1
## 1.2214559 3.3145327 1.7655141 1.4102610 -0.5476591
## lakeHancock:2 lakeHancock:3 lakeHancock:4 lakeOklawaha:1
lakeOklawaha:2
## -1.2427766 -2.9011352 -0.4165804 -3.1120797 -2.4588720
## lakeOklawaha:3 lakeOklawaha:4 lakeTrafford:1 lakeTrafford:2
lakeTrafford:3
## -1.5216526 -2.4532189 -1.8474865 -2.9352533 -1.8132685
## lakeTrafford:4 sizesmall:1 sizesmall:2 sizesmall:3
sizesmall:4
## -1.4188846 -0.2793969 0.3512628 1.8094675 0.6828131
##
## Degrees of Freedom: 64 Total; 44 Residual
## Residual deviance: 52.47849
## Log-likelihood: -74.42948
##
## This is a multinomial logit model with 5 levels
```

There are lots of coefficients here! These can be interpreted in terms of log odds, but instead we'll examine the model fit graphically.

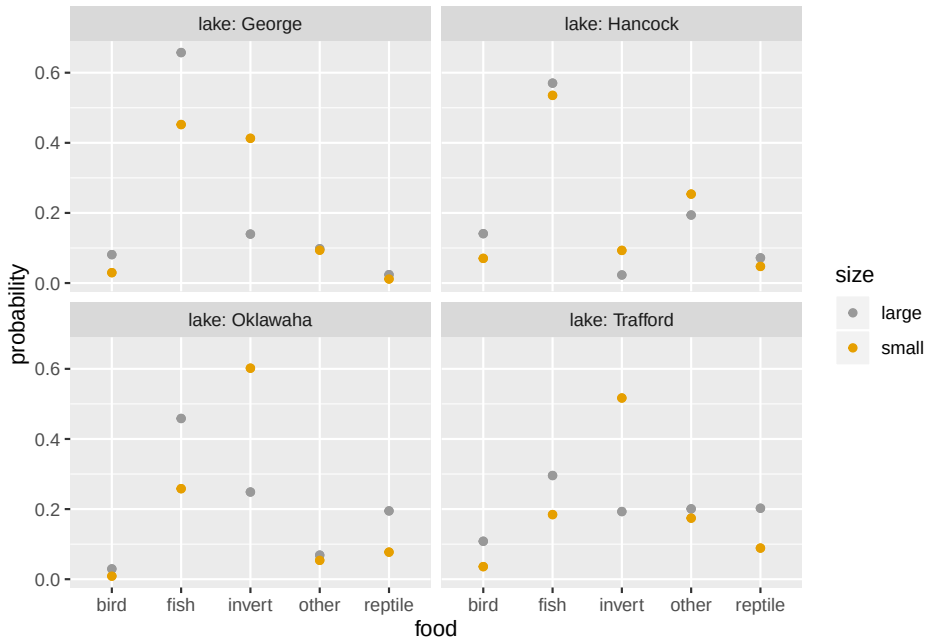
```
alligator.mlogit.df = data.frame(model.frame(alligator.mlogit),
  fitted.values(alligator.mlogit))
alligator.mlogit.long = alligator.mlogit.df %>%
  gather(food, probability, bird:reptile)
ggplot(alligator.mlogit.long, aes(x = food, y = probability)) +
  geom_point() +
  facet_wrap(~lake + size, ncol = 4, labeller = label_both)
```



This is basically a standardized and smoothed version of our faceted bar graphs above. The smoothing might have some advantages over just taking raw data, e.g. there are no zero probabilities.

We could also collapse the large and small rows and color-code:

```
ggplot(alligator.mlogit.long, aes(x = food, y = probability, col = size)) +
  geom_point() +
  facet_wrap(~ lake, labeller = label_both) +
  scale_color_manual(values = cb_palette)
```



Let's check the deviance of a few alternatives:

```
deviance(alligator.mlogit)
```

```
## [1] 52.47849
```

```
deviance(vglm(cbind(bird, fish, invert, other, reptile) ~ lake + size + sex,
    family = multinomial, data = alligator.wide))
```

```
## [1] 50.26369
```

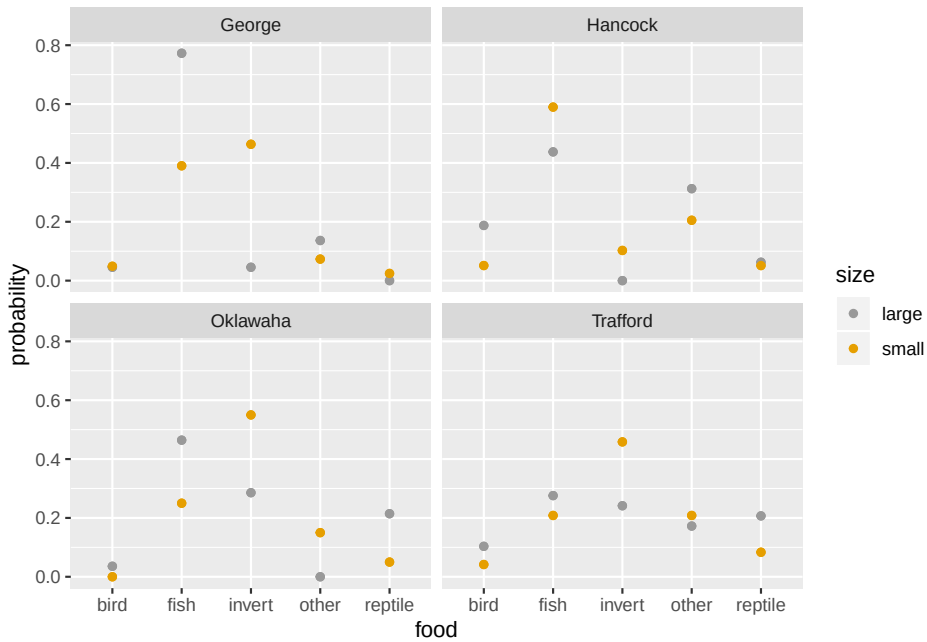
```
deviance(vglm(cbind(bird, fish, invert, other, reptile) ~ lake * size,
    family = multinomial, data = alligator.wide))
```

```
## [1] 35.39866
```

Adding sex only reduces deviance by a trivial amount (less than the 4 extra degrees of freedom), and so is unlikely to be worth it. Adding an interaction between lake and size reduces deviance by a resonable amount but not AIC and makes the model much more complicated, so that's a judgment call. If you decided to keep the interaction:

```
alligator.int = vglm(cbind(bird, fish, invert, other, reptile) ~ lake * size,
    family = multinomial, data = alligator.wide)
alligator.int.df = data.frame(model.frame(alligator.int),
    fitted.values(alligator.int))
alligator.int.long = alligator.int.df %>%
  gather(food, probability, bird:reptile)
```

```
ggplot(alligator.int.long, aes(x = food, y = probability, col = size)) +
  geom_point() + facet_wrap(~ lake) + scale_color_manual(values = cb_palette)
```



But when we add an interaction between categorical predictors, this is just equivalent to taking the raw proportions for each two-way combination of predictors. So we're back to the two-way faceted bar graph, only standardized so that each panel adds up to probability 1.

10.2.5 Quantitative predictors

We can also fit multinomial models with quantitative predictors. In the file `gator2.txt`, the numerical predictor is the length of the alligator in meters.

```
gator2 = read.table("gator2.txt", header = TRUE)
summary(gator2)
```

```
##      length                 food
##  Min.   :1.240   Fish         :31
##  1st Qu.:1.575   Invertebrates:20
##  Median :1.850   Other        : 8
##  Mean   :2.130
##  3rd Qu.:2.450
##  Max.   :3.890
```

```
gator2.mlogit = vglm(food ~ length, family = multinomial, data = gator2)
gator2.mlogit
```

```
##
## Call:
## vglm(formula = food ~ length, family = multinomial, data =
gator2)
##
##
## Coefficients:
## (Intercept):1 (Intercept):2 length:1 length:2
## 1.617731 5.697444 -0.110109 -2.465446
##
## Degrees of Freedom: 118 Total; 114 Residual
## Residual deviance: 98.34124
## Log-likelihood: -49.17062
##
## This is a multinomial logit model with 3 levels
```

As we did in the ordered categories case, let's start making predictions to understand the fit. First, on the linear predictor (i.e. transformed) scale:

```
log.ratios = predict(gator2.mlogit, newdata = data.frame(length = 2))
log.ratios
```

```
##   log(mu[,1]/mu[,3]) log(mu[,2]/mu[,3])
## 1           1.397513          0.7665519
```

This gives us the log probability ratios for one type of food to another. The log of the probability ratio for fish to other is $1.62 - 0.11 \times 2 \approx 1.4$ and for invertebrates to other is $5.7 - 2.47 \times 2 \approx 0.77$. (Note that `vglm()` take the *last* level of the factor as the baseline, which is weird but is what it is.)

Now, on the probability scale:

```
twometerprobs = predict(gator2.mlogit, newdata = data.frame(length = 2),
    type = "response")
twometerprobs
```

```
##        Fish Invertebrates     Other
## 1 0.5620216     0.2990405 0.1389379
```

To go from the probability scale to the linear predictor scale:

```
log(twometerprobs[1:2] / twometerprobs[3])
```

```
## [1] 1.3975134 0.7665519
```

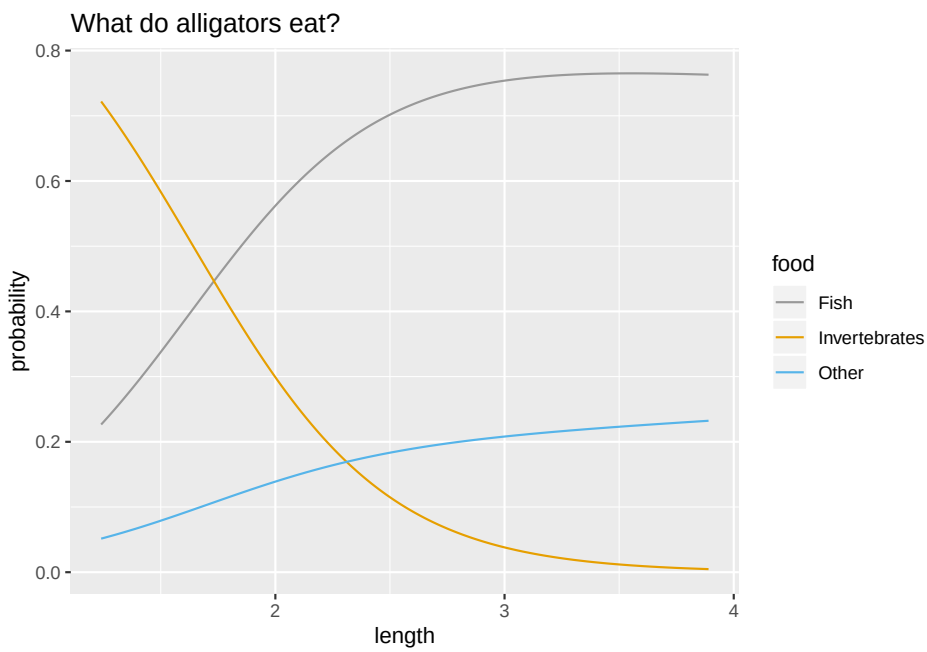
To go from the linear predictor scale to the probability scale:

```
exp(c(log.ratios, 0)) / sum(exp(c(log.ratios, 0)))
```

```
## [1] 0.5620216 0.2990405 0.1389379
```

Now let's look at how these probabilities vary with length:

```
length = data.frame(length = seq(1.24, 3.89, 0.01))
gator2.pred = predict(gator2.mlogit, newdata = length, type = "response")
gator2.pred.df = data.frame(length, gator2.pred)
gator2.pred.long = gator2.pred.df %>%
  gather(food, probability, Fish:Other)
ggplot(gator2.pred.long, aes(x = length, y = probability, group = food, color = food)) +
  geom_line() +
  ggtitle("What do alligators eat?") +
  scale_color_manual(values = cb_palette)
```



Bigger alligators prefer fish and, to a lesser extend, "other." Smaller alligators prefer invertebrates.

We finally note that just as with the Poisson, multinomial data is often overdispersed, so be careful of taking standard errors literally.

Chapter 11

High-dimensional data

11.1 Regression with lots of predictors

Optional reading: Ch. 3.4 of the Elements of Statistical Learning.

In this section, we'll consider situations with "lots" of predictors, where lots might mean ten or twenty (but not the "$p > n$" situation with more predictors than observations, which presents a distinctive set of challenges.) In this situation, where prediction is not the goal (or at least not the only goal), there may be no single "best" model. Rather, different models might give you different insights about the data.

11.1.1 Prostate cancer

The `prostate` data set in the `ElemStatLearn` package contains measurements of 97 men suffering from prostate cancer.

```r
library(tidyverse)
# install.packages("ElemStatLearn")
library(ElemStatLearn)
data(prostate)
summary(prostate)
```

```
## lcavol lweight age lbph
## Min.  :-1.3471 Min.  :2.375 Min.  :41.00 Min.  :-1.3863
## 1st Qu.: 0.5128 1st Qu.:3.376 1st Qu.:60.00 1st Qu.:-1.3863
## Median : 1.4469 Median :3.623 Median :65.00 Median : 0.3001
## Mean : 1.3500 Mean :3.629 Mean :63.87 Mean : 0.1004
## 3rd Qu.: 2.1270 3rd Qu.:3.876 3rd Qu.:68.00 3rd Qu.: 1.5581
## Max.  : 3.8210 Max.  :4.780 Max.  :79.00 Max.  : 2.3263
## svi lcp gleason pgg45
```

```
## Min.   :0.0000 Min.   :-1.3863 Min.   :6.000 Min.   : 0.00
## 1st Qu.:0.0000 1st Qu.:-1.3863 1st Qu.:6.000 1st Qu.: 0.00
## Median :0.0000 Median :-0.7985 Median :7.000 Median : 15.00
## Mean :0.2165 Mean :-0.1794 Mean :6.753 Mean : 24.38
## 3rd Qu.:0.0000 3rd Qu.: 1.1787 3rd Qu.:7.000 3rd Qu.: 40.00
## Max.   :1.0000 Max.   : 2.9042 Max.   :9.000 Max.   :100.00
## lpsa train
## Min.   :-0.4308 Mode :logical
## 1st Qu.: 1.7317 FALSE:30
## Median : 2.5915 TRUE :67
## Mean : 2.4784
## 3rd Qu.: 3.0564
## Max.   : 5.5829
```

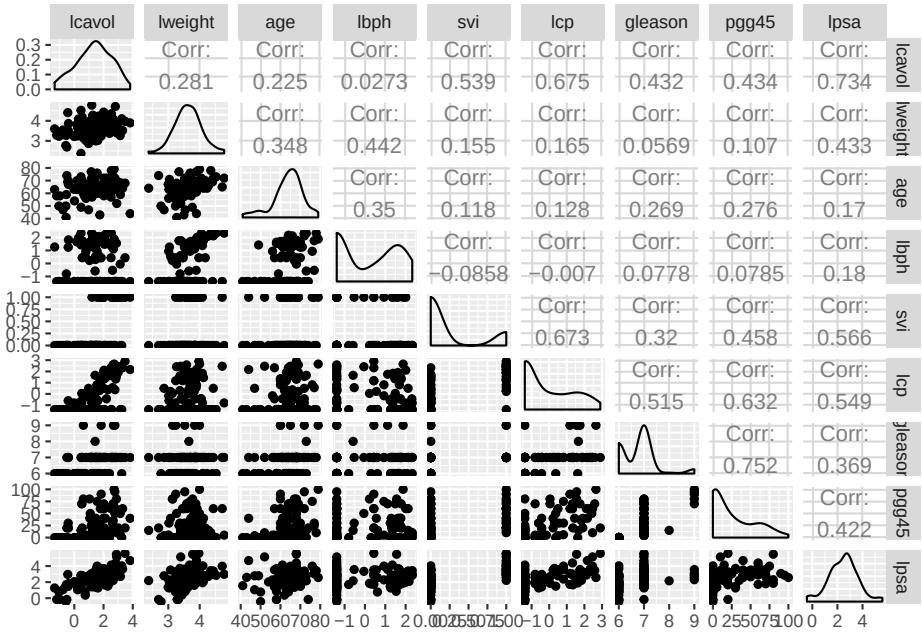
```r
apply(prostate, 2, sd)
```

```
## lcavol lweight age lbph svi lcp gleason
## 1.1786249 0.4284112 7.4451171 1.4508066 0.4139949 1.3982496
0.7221341
## pgg45 lpsa train
## 28.2040346 1.1543291 0.4645972
```

The response is `lpsa`, the log of "prostate specific antigen," a protein which is often at an elevated level in prostate cancer sufferers. There are also eight predictor variables (and an indicator `train` which we'll disregard.)

Draw a pairs plot (omitting `train`):

```r
library(GGally)
ggpairs(prostate[, 1:9])
```



`lpsa` seems to be related to most of the other variables. It seems to have a reasonably symmetric distribution, so we won't transform further. (You could make arguments for transforming some of the predictors,but for simplicity, we won't.)

Eight predictors is getting toward the borderline where we need to use variable selection or regularization to avoid overfitting. But first, let's start by throwing all the predictors into a linear model.

11.1.2 Full linear model

Because there are lots of variables and I don't know much about prostate cancer, I'll rescale the variables so they all have mean 0 and SD 1, to allow easy comparison of their coefficients. (I prefer to not standardize the response.) Then we'll fit a full linear model, with all eight predictors.

```
predictors.scaled = scale(prostate[, 1:8])
lpsa = prostate$lpsa
prostate.lm = lm(lpsa ~ predictors.scaled)
```

As we've seen, the `summary()` function gives a bit of a mess, and that only gets worse as the number of predictors increases. Instead, we can use the `tidy()` function in `broom` to create a nice data frame of coefficients and their confidence intervals.

```
library(broom)
prostate.lm.tidy = tidy(prostate.lm, conf.int = TRUE)
```

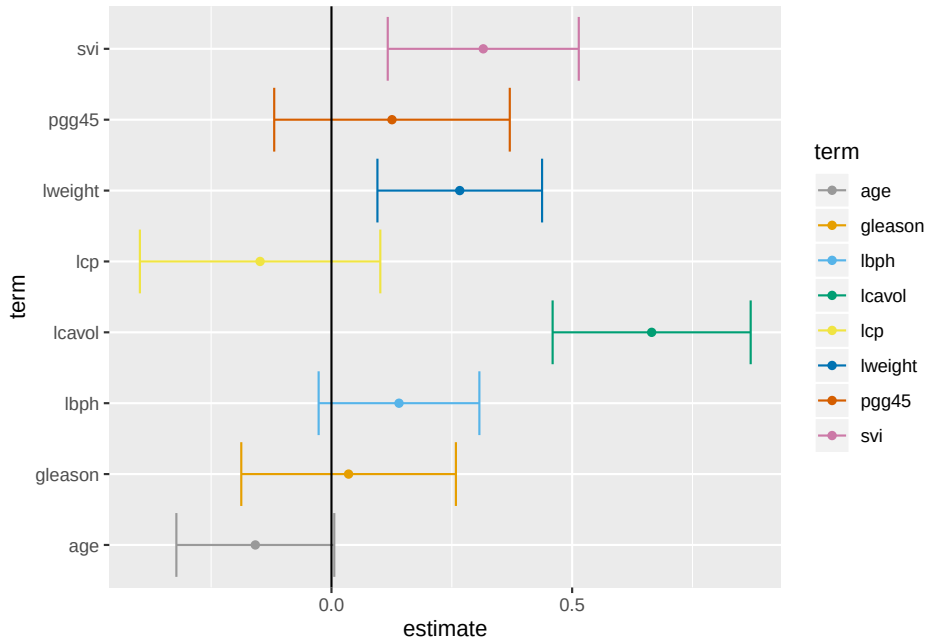
```
# Go back to original names
prostate.lm.tidy[2:9, 1] = names(prostate)[1:8]
data.frame(prostate.lm.tidy[,1], round(prostate.lm.tidy[,-1], 2))
```

```
## term estimate std.error statistic p.value conf.low conf.high
## 1 (Intercept) 2.48 0.07 34.90 0.00 2.34 2.62
## 2 lcavol 0.67 0.10 6.43 0.00 0.46 0.87
## 3 lweight 0.27 0.09 3.10 0.00 0.10 0.44
## 4 age -0.16 0.08 -1.92 0.06 -0.32 0.01
## 5 lbph 0.14 0.08 1.67 0.10 -0.03 0.31
## 6 svi 0.32 0.10 3.16 0.00 0.12 0.51
## 7 lcp -0.15 0.13 -1.18 0.24 -0.40 0.10
## 8 gleason 0.04 0.11 0.32 0.75 -0.19 0.26
## 9 pgg45 0.13 0.12 1.02 0.31 -0.12 0.37
```

The standard errors of the coefficients are of similar magnitude. The biggest coefficient is for `lcavol` (log cancer volume.) Since the predictors are scaled and response is a natural log, small coefficients (of the order of 0.1 or 0.2) can be interpreted as the proportion change in the prediction unlogged response if the predictor is one SD higher. So if age were one SD higher (about 7.5 years) and everything else was constant, the model's predicted PSA would be about 16% lower.

We can also draw a **TIE fighter plot** of the coefficients (that might be a name I just made up):

```
gg = ggplot(prostate.lm.tidy[-1,], aes(x = estimate, y = term,
    xmin = conf.low, xmax = conf.high, color = term)) +
  geom_point() + geom_errorbarh() + geom_vline(xintercept = 0)
cb_palette = c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55
cbp = function(){scale_color_manual(values = cb_palette)}
gg + cbp()
```



Without worrying too much about significance, we see that some of the coefficients are very small compared to their margins of error. We should thus consider a model with fewer predictors.

11.2 All subsets

There are lots of ways to do variable selection. An old-school method is to do **stepwise** selection, but this has the fairly major weakness that it doesn't always choose the best model of a given size, let alone the right model. In fact, given a half-decent laptop, there's no difficulty in looking at *all* subsets of eight predictor variables, and finding the best (i.e. lowest squared error) subset for each size. `regsubsets()` in the `leaps` package does this for you. We run it on the unscaled data (excluding the `train` variable.)

```
# install.packages("leaps")
library(leaps)
prostate.leaps = regsubsets(lpsa ~ . - train, data = prostate)
summary(prostate.leaps)$which
```

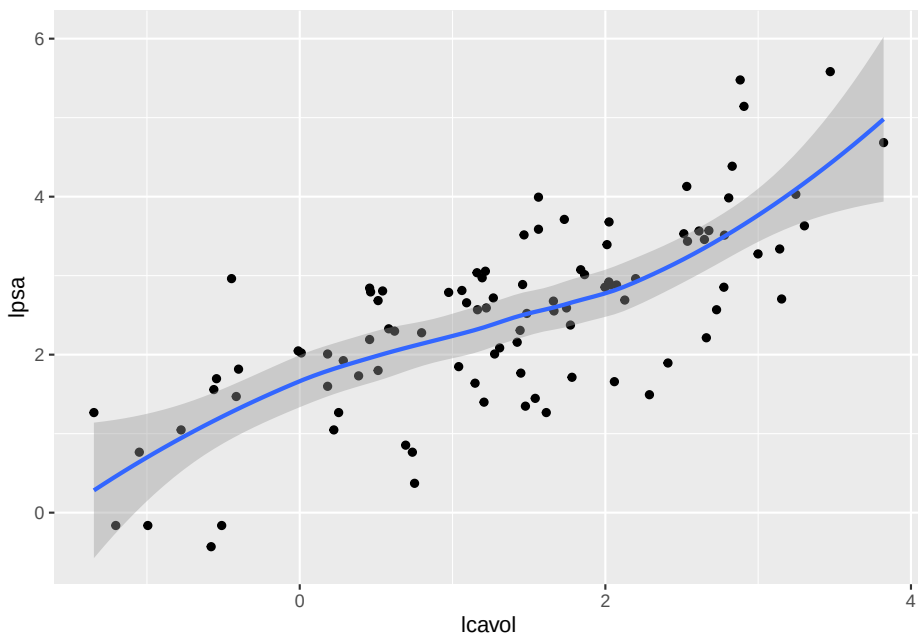
```
## (Intercept) lcavol lweight age lbph svi lcp gleason pgg45
## 1 TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 2 TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## 3 TRUE TRUE TRUE FALSE FALSE TRUE FALSE FALSE FALSE
## 4 TRUE TRUE TRUE FALSE TRUE TRUE FALSE FALSE FALSE
## 5 TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE
```

```
## 6 TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE TRUE
## 7 TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE
## 8 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

So we see that for example, the best 3-predictor model includes `lcavol`, `lweight`, and `svi`. (Note that if we only want to find the best models, it doesn't matter whether we run `regsubsets()` with the scaled or unscaled predictors.) We see that the best models for each size form a nested sequence, so a greedy algorithm would've worked here.

Now we can build up our understanding of the data by drawing plots, adding variables sequentially and checking for nonlinearity and interactions as we've done throughout the course. Start with log cancer volume as a single predictor:

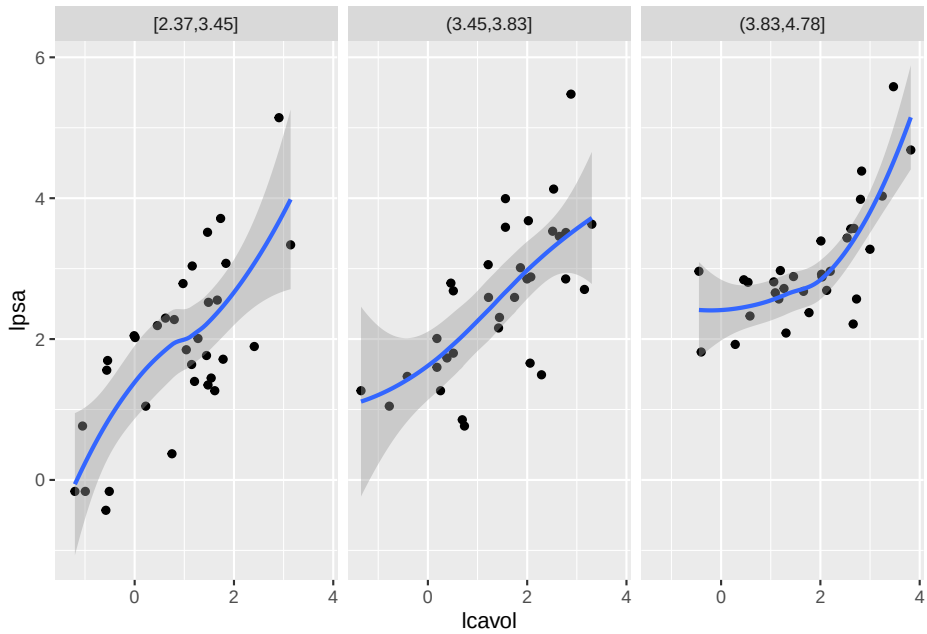
```
ggplot(prostate, aes(x = lcavol, y = lpsa)) + geom_point() + geom_smooth()
```



The relationship seems reasonably close to linear here – we could draw a straight line entirely within the confidence band. If there was more curvature, I'd throw the model out and start from scratch using a nonparametric approach.

Now facet by log weight:

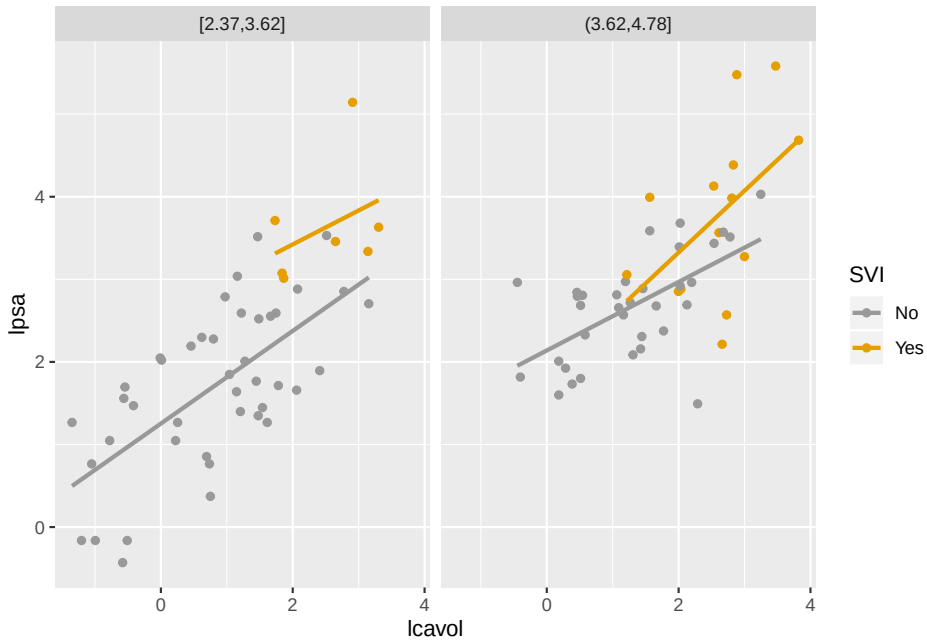
```
ggplot(prostate, aes(x = lcavol, y = lpsa)) + geom_point() +
  geom_smooth(span = 1) + facet_grid(~ cut_number(lweight, n = 3))
```



There's an intriguing hint of nonlinearity here – the log PSA prediction increases more and more quickly with log cancer volume, but only for the heavy patients. With lots of data, we could switch to a multipredictor loess here, but since our sample is small, let's first see if this pattern shows up in other graphs.

Next, condition on `svi` (seminal vesicle invasion) which if present, indicates the cancer is advanced. Since this is a binary variable, we can make it a factor and distinguish it with color. However, as only 21 individuals in the data set have SVI, the number of facets we can show becomes limited, and it's hard to fit curves instead of lines.

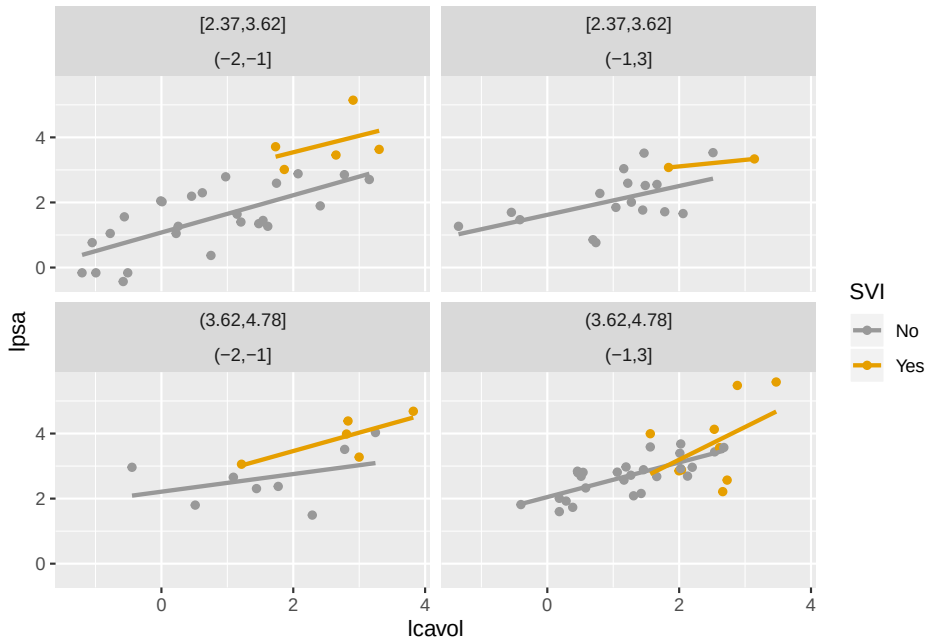
```
prostate$SVI = recode(prostate$svi, `1` = "Yes", `0` = "No")
ggplot(prostate, aes(x = lcavol, y = lpsa, group = SVI, color = SVI)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE) +
  facet_wrap(~ cut_number(lweight, n = 2)) + cbp()
```



There's not much hint of nonlinearity here. However, the lines on the right panel are far from parallel, so we should keep the possibility of an interaction in mind.

Now facet again on `lbph`, "log benign prostatic hyperplasia amount":

```
ggplot(prostate, aes(x = lcavol, y = lpsa, group = SVI, color = SVI)) +
  geom_point() + geom_smooth(method = "lm", se = FALSE) +
  facet_wrap(~ cut_number(lweight, n=2) +
      cut(lbph, breaks = c(-2, -1, 3))) +
  cbp()
```



It's getting harder to be confident about an interaction: the orange lines vary in slope, but there's based on very small samples. The grey lines have different heights but are similar in slope.

In EDA we're not always required to find a "best" model, and even if we were we can decide on what best means subjectively. So if you wanted to fit a linear model with `lcavol`, `lweight`, and `svi` as predictors plus interactions, you're free to do so and then call that "best" because of the complexity you can get out of a relatively small number of variables. On the other hand, if you want a (somewhat) objective decision for "best", you can just find the model that optimizes your favorite criterion. For example, if you like Mallow's C_p:

```
summary(prostate.leaps)$cp
```

```
## [1] 27.406210 14.747299 6.173546 6.185065 5.816804 6.466493
7.100428
## [8] 9.000000
```

the best model is the five-predictor one:

```
tidy(lm(lpsa ~ lcavol + lweight + svi + lbph + age, data = prostate))
```

```
## # A tibble: 6 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   0.495     0.877      0.564 5.74e- 1
## 2 lcavol        0.544     0.0746     7.29  1.11e-10
```

```
## 3 lweight       0.588     0.198      2.97  3.78e- 3
## 4 svi           0.715     0.207      3.46  8.20e- 4
## 5 lbph          0.101     0.0576     1.76  8.22e- 2
## 6 age          -0.0164    0.0107    -1.54  1.27e- 1
```

Note, however, that the standard errors and P-values are wrong – we selected the model *because* it fitted the data well, so the goodness-of-fit will be overestimated. If in doubt, bootstrap.

11.3 If prediction is a major goal

If prediction is a major goal (in addition to interpretation), there are lots of modern techniques to consider, such as principal components regression, ridge regressions, and Lasso. These are technically beyond the scope of this course and you should really go read *Elements of Statistical Learning*, but we briefly discuss some of them.

11.3.1 Principal components regression

Model selection methods of the kind shown above can be unstable and noisy, and generally give suboptimal prediction. A method that works very well for prediction is **principal components regression (PCR)**: fit a PCA on the predictors (usually scaled), then regress the response on the first k principal components.

```r
predictors.pc = prcomp(predictors.scaled)
prostate.pcr = lm(lpsa ~ predictors.pc$x[, 1:4])
tidy(prostate.pcr)
```

```
## # A tibble: 5 x 5
## term estimate std.error statistic p.value
## <chr> <dbl> <dbl> <dbl> <dbl>
## 1 (Intercept) 2.48 0.0762 32.5 3.17e-52
## 2 predictors.pc$x[, 1:4]PC1 0.432 0.0418 10.3 4.61e-17
## 3 predictors.pc$x[, 1:4]PC2 0.0714 0.0597 1.20 2.35e- 1
## 4 predictors.pc$x[, 1:4]PC3 0.387 0.0776 4.98 2.94e- 6
## 5 predictors.pc$x[, 1:4]PC4 0.133 0.0964 1.38 1.72e- 1
```

(If you really care about prediction, choose k by cross-validation.) While the predictive accuracy of PCR is often shockingly good (provided the data is well-behaved), it's extremely finicky to interpret – once you transform back to the original predictors, there's no sparsity. So we don't use it for EDA much. But if you ever want to, the implementation people seem to like is in the `caret` package.

11.3.2 Ridge regression

Ridge regression is closely related to (a smoothed version of) PCR, but it's easier to introduce as a form of *penalized* regression. We find coefficients β_j that minimize

$$\sum_i \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

where λ is a tuning parameter. In practice, this means the coefficients **shrink** toward zero, reducing variance at the expense of a little bias. Note that this criterion usually only makes sense if the predictors are scaled.

The `linearRidge` function in package `ridge` fits ridge regression, choosing λ via a method by Cule and De Iorio.

```
# install.packages("ridge")
library(ridge)
prostate.ridge = linearRidge(lpsa ~ predictors.scaled)
summary(prostate.ridge)
```

```
##
## Call:
## linearRidge(formula = lpsa ~ predictors.scaled)
##
##
## Coefficients:
## Estimate Scaled estimate Std. Error (scaled)
## (Intercept) 2.47839 NA NA
## predictors.scaledlcavol 0.57430 5.62697 0.82949
## predictors.scaledlweight 0.25801 2.52794 0.73697
## predictors.scaledage -0.12219 -1.19722 0.71803
## predictors.scaledlbph 0.12347 1.20975 0.72474
## predictors.scaledsvi 0.28180 2.76111 0.81276
## predictors.scaledlcp -0.04981 -0.48805 0.92867
## predictors.scaledgleason 0.04660 0.45656 0.85639
## predictors.scaledpgg45 0.09470 0.92782 0.90517
## t value (scaled) Pr(>|t|)
## (Intercept) NA NA
## predictors.scaledlcavol 6.784 1.17e-11 ***
## predictors.scaledlweight 3.430 0.000603 ***
## predictors.scaledage 1.667 0.095441 .
## predictors.scaledlbph 1.669 0.095073 .
## predictors.scaledsvi 3.397 0.000681 ***
## predictors.scaledlcp 0.526 0.599208
## predictors.scaledgleason 0.533 0.593945
```

```
## predictors.scaledpgg45 1.025 0.305350
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Ridge parameter: 0.07978994, chosen automatically, computed
using 6 PCs
##
## Degrees of freedom: model 6.925 , variance 6.053 , residual
7.797
```

As with PCR, no predictors are dropped, but we can check that the sum of squared coefficients (excluding the intercept) has shrunk compared to the full model:

```
sum(coef(prostate.lm)[-1]^2)
```

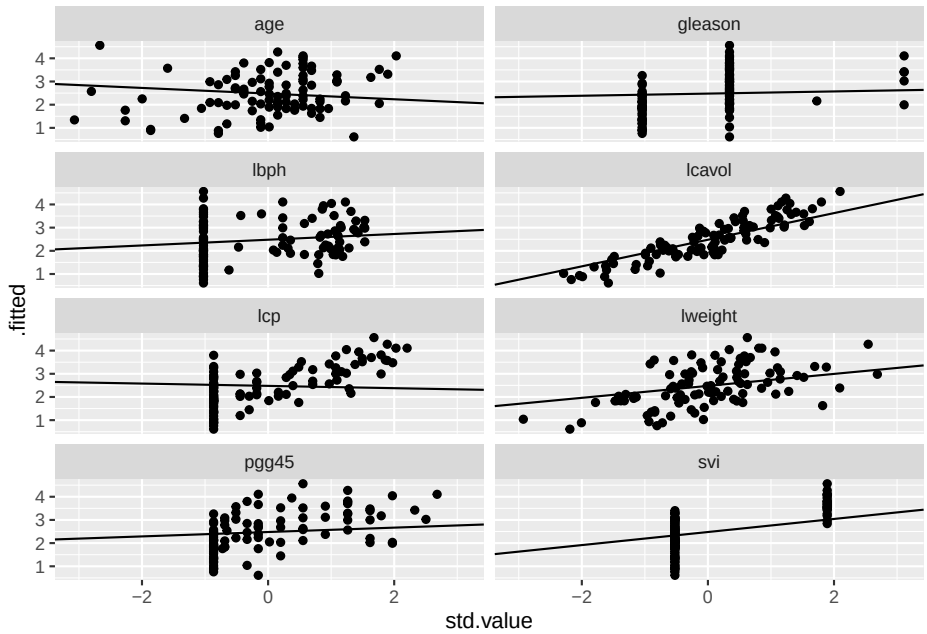
```
## [1] 0.6966349
```

```
sum(coef(prostate.ridge)[-1]^2)
```

```
## [1] 0.5195971
```

The model is still hard to understand, but we can draw pictures of the fitted values and data.

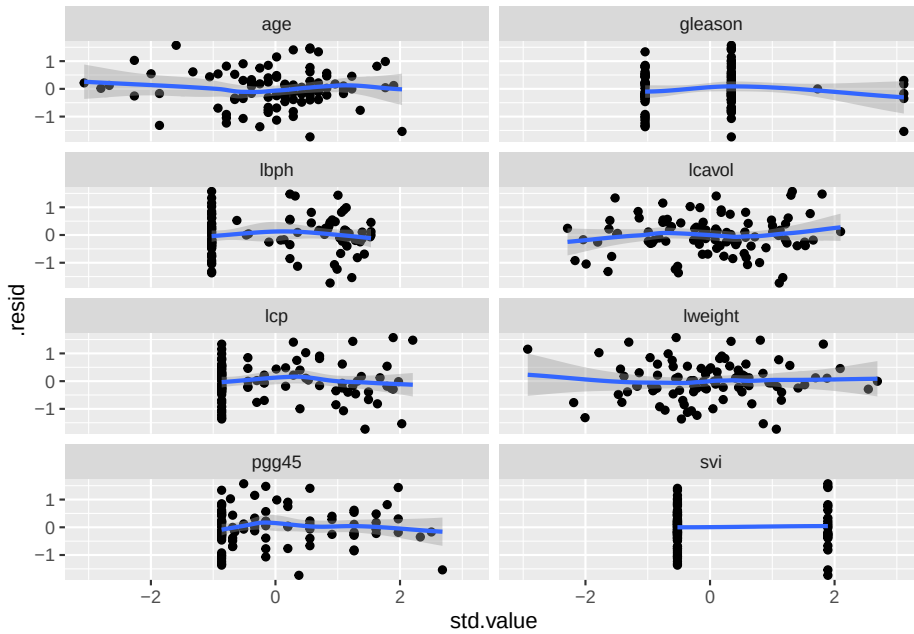
```
prostate.ridge.fitted = predict(prostate.ridge)
prostate.ridge.resid = lpsa - predict(prostate.ridge)
prostate.ridge.df = data.frame(predictors.scaled, prostate.ridge.fitted, prostate.ridge
names(prostate.ridge.df)[9:10] = c(".fitted", ".resid")
library(tidyr)
prostate.ridge.fit.long = prostate.ridge.df %>% gather(variable, std.value, 1:8)
prostate.ridge.int = coef(prostate.ridge)[1]
prostate.ridge.coef.df = data.frame(variable = names(prostate)[1:8],
    coef = coef(prostate.ridge)[-1])
ggplot(prostate.ridge.fit.long, aes(x = std.value, y = .fitted)) +
  geom_point() + facet_wrap(~variable, ncol = 2) +
  geom_abline(data = prostate.ridge.coef.df,
    aes(intercept = prostate.ridge.int, slope = coef))
```



Note that the lines are *not* the least squares lines for each variable; instead, they give the predicted log PSA by value of that variable if all other variables are fixed at their means. Log cancer volume has the steepest slope, while the Gleason score (an ordinal on a ten point scale giving the risk of the cancer spreading) is almost flat.

Now look at the residuals:

```
ggplot(prostate.ridge.fit.long, aes(x = std.value, y = .resid)) +
  geom_point() + facet_wrap(~variable, ncol = 2) +
  geom_smooth(method.args = list(degree = 1))
```



There's no obvious nonlinearity. Our linear model-based methods seem adequate.

11.3.3 Lasso

The **lasso** gives you both shrinkage and sparsity. We find coefficients β_j that minimize

$$\sum_i \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

where λ is a tuning parameter. This usually requires shrinking some of the β_js to zero.

The two most-used R implementations of the lasso are `lars()` in the package of the same name, and the more general `glmnet` package, which can also fit (cross-validated) ridge regression.

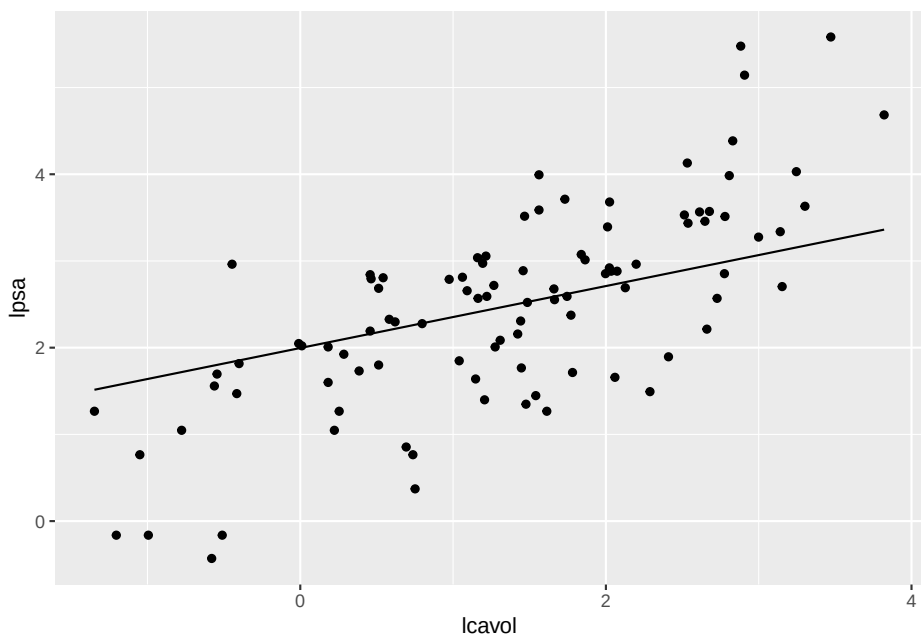
```
# install.packages("lars")
library(lars)
prostate.lasso = lars(as.matrix(predictors.scaled), lpsa)
prostate.lasso
```

```
##
## Call:
## lars(x = as.matrix(predictors.scaled), y = lpsa)
```

```
## R-squared: 0.663
## Sequence of LASSO moves:
##      lcavol svi lweight pgg45 lbph age gleason lcp
## Var       1   5       2     8    4   3       7   6
## Step      1   2       3     4    5   6       7   8
```

`lars()` gives a sequence of models that increment the number of predictors. We can look at the first few of these in turn. One predictor:

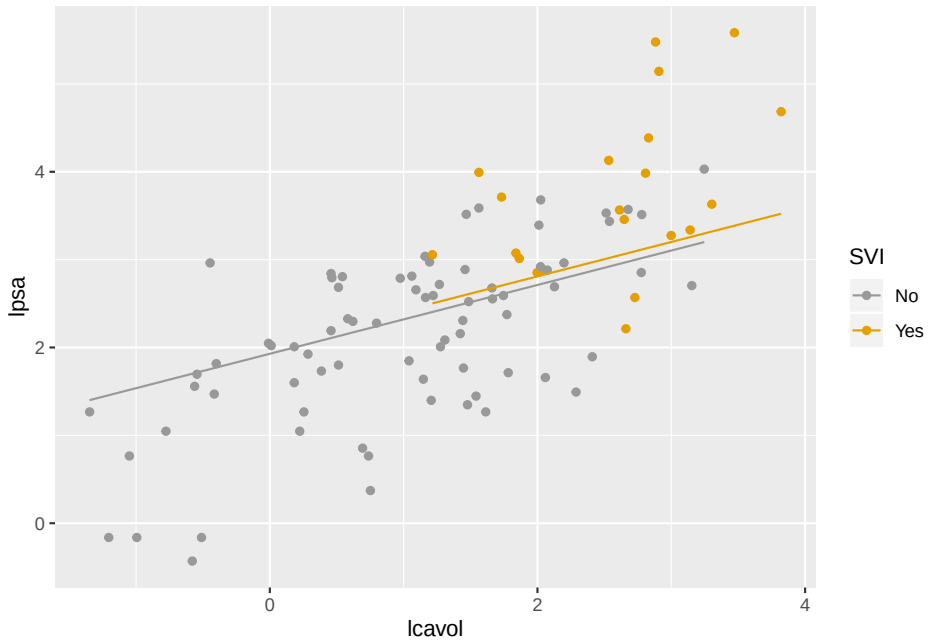
```
prostate.lasso.fit = predict.lars(prostate.lasso, newx = predictors.scaled)$fit
prostate.lasso.df = data.frame(prostate, fit1 = prostate.lasso.fit[,2],
    fit2 = prostate.lasso.fit[,3], fit3 = prostate.lasso.fit[,4])
ggplot(prostate.lasso.df, aes(x = lcavol, y = lpsa)) +
  geom_point() + geom_line(aes(y = fit1))
```



The first predictor is still log cancer volume. Once again, this isn't the least squares line – the slope is shrunk toward zero.

Two predictors:

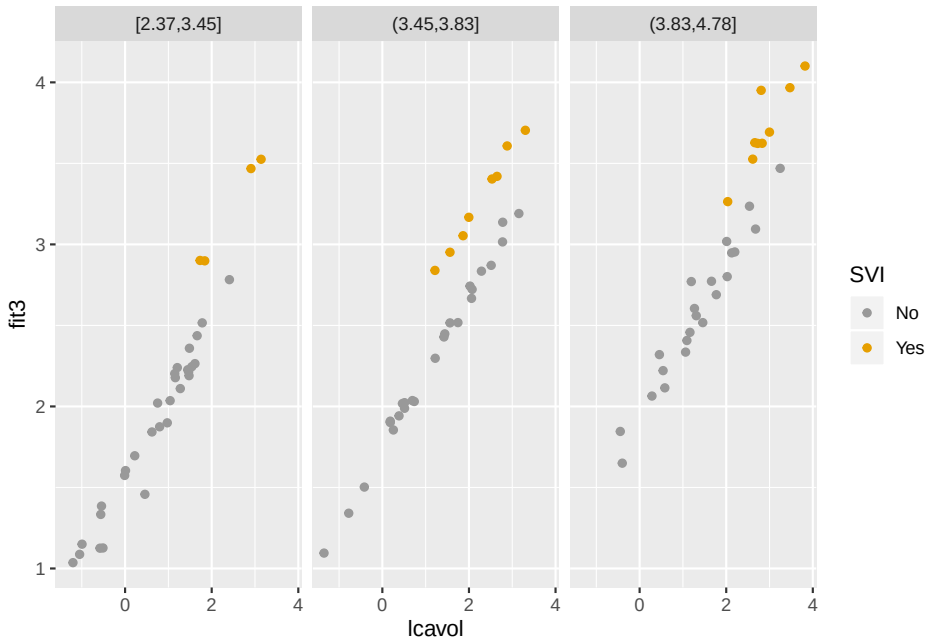
```
ggplot(prostate.lasso.df, aes(x = lcavol, y = lpsa, group = SVI, color = SVI)) +
  geom_point() + geom_line(aes(y = fit2)) + cbp()
```



This time SVI is the second variable. It doesn't look like the model fits that well at this point (the best model with these two predictors might have an interaction.)

Three predictors:

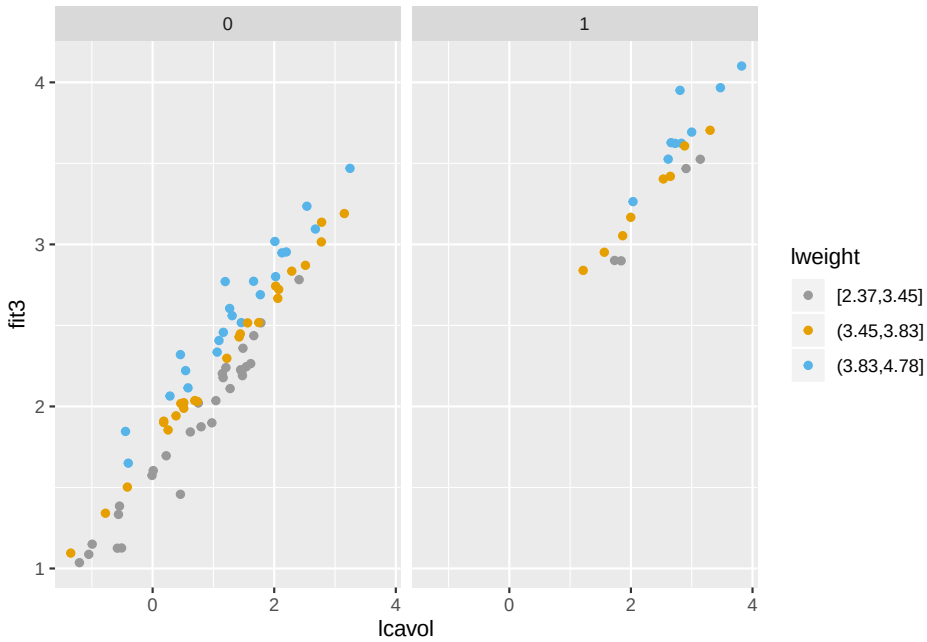
```
ggplot(prostate.lasso.df, aes(x = lcavol, y = fit3, group = SVI, color = SVI)) +
  geom_point() + facet_grid(~cut_number(lweight, n = 3)) + cbp()
```



We faceted by (log) weight and plotted fitted values (not raw data.) We see that within each panel, SVI matters but weight doesn't matter much – perhaps not surprising since each panel contains patients of similar weight.

We can facet on SVI instead:

```
ggplot(prostate.lasso.df, aes(x = lcavol, y = fit3,
    color = cut_number(lweight, n = 3))) +
  geom_point() + facet_grid(~svi) + cbp() + labs(color = "lweight")
```



The weight effect is there, since the blue dots (highest weight) are above the green dots, which are above the red dots. But the effect is pretty small.

11.3.4 Finding the "best" model using glmnet()

If you're required to find a "best" lasso model (or equivalently a "best" tuning parameter λ,) the `cv.glmnet()` function in `glmnet` can pick one by cross-validation.

```
# install.packages("glmnet")
library(glmnet)
prostate.cv = cv.glmnet(as.matrix(predictors.scaled), lpsa)
coef(prostate.cv, s = "lambda.min")
```

```
## 9 x 1 sparse Matrix of class "dgCMatrix"
##                          1
## (Intercept)  2.478386878
## lcavol       0.601019011
## lweight      0.241731323
## age         -0.082347028
## lbph         0.101357032
## svi          0.249585872
## lcp          .
## gleason      0.008064517
## pgg45        0.068282221
```

The model here is optimized for prediction. However, for exploratory work, this often results in an overly complex model. An unrigorous fix is to use the largest λ that gives error within one standard error of the minimum. (This is really just an excuse to drop terms, but some people would rather have an excuse than make a subjective decision.)

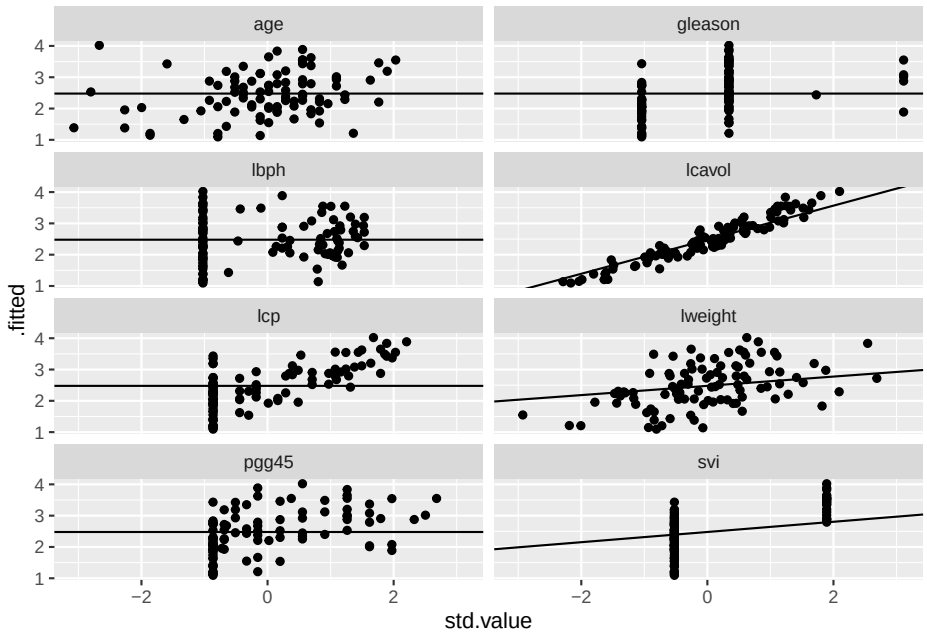
```
coef(prostate.cv, s = "lambda.1se")
```

```
## 9 x 1 sparse Matrix of class "dgCMatrix"
##                     1
## (Intercept) 2.4783869
## lcavol      0.5441106
## lweight     0.1478673
## age         .
## lbph        .
## svi         0.1630816
## lcp         .
## gleason     .
## pgg45       .
```

A dot means the coefficient for that term is set to zero. So we've now got a much simpler model.

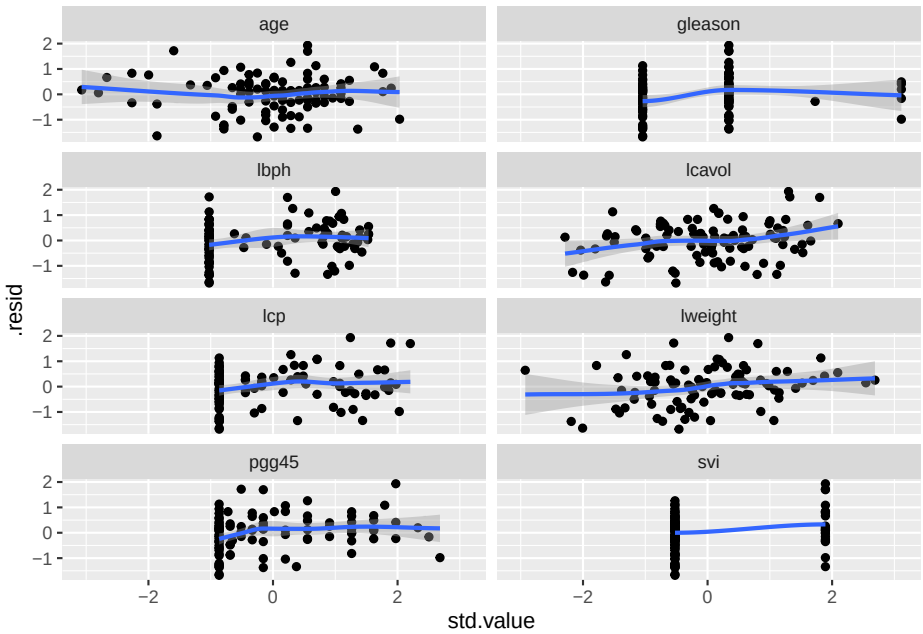
We can plot the model against each predictor, setting all other predictors to zero (this only makes sense if the predictors are standardized):

```
prostate.cv.int = coef(prostate.cv, s = "lambda.1se")[1]
prostate.cv.fitted = predict(prostate.cv, s = "lambda.1se",
    newx = as.matrix(predictors.scaled))
prostate.cv.resid = lpsa - prostate.cv.fitted
prostate.cv.df = data.frame(predictors.scaled, prostate.cv.fitted, prostate.cv.resid)
names(prostate.cv.df)[9:10] = c(".fitted", ".resid")
prostate.cv.fit.long = prostate.cv.df %>% gather(variable, std.value, 1:8)
prostate.cv.coef.df = data.frame(variable = row.names(coef(prostate.cv, s = "lambda.1se")),
    coef = as.vector(coef(prostate.cv, s = "lambda.1se")))
ggplot(prostate.cv.fit.long, aes(x = std.value, y = .fitted)) +
  geom_point() + facet_wrap(~ variable, ncol = 2) +
  geom_abline(data = prostate.cv.coef.df[-1,],
    aes(intercept = prostate.cv.int, slope = coef))
```



So even though we see the, for example, `lcp` (log capsular penetration) has a moderately strong relationship with the response, we set its coefficient to zero. That is, we hope its relationship with the response is well-explained by the variables we kept. We check this with a residual plot:

```
ggplot(prostate.cv.fit.long, aes(x = std.value, y = .resid)) +
  geom_point() + facet_wrap(~ variable, ncol = 2) +
  geom_smooth(method.args = list(degree = 1))
```



We see that `lcp`, like most of the other variables, has a flat residual plot. However, there's still some slope left in the residuals for the variables we kept in the model (`lcavol`, `lweight`, `svi`) because we shrunk the coefficients. Now to be honest, lasso isn't going to give much of an improvement in prediction over just choosing the best subset, and is going to complicate things in the sense that you have to learn a new bunch of theory to understand what's going on. But if you know the theory, it's a solid choice.

11.4 PCA

Optional reading: Ch. 16 of Shalizi's Advanced Data Analysis (http://www.stat.cmu.edu/~cshalizi/ADAfaEPoV/) is mathy but correct and useful. I don't know of a good treatment that doesn't have a bunch of linear algebra yet isn't a horrendous over-simplification; if you've seen one, let me know.

In **principal components analysis (PCA)**, we project a high-dimensional set of data on to the k-dimensional (hyper-) plane that retains the most of the original data's variance. This is done via a singular value decomposition. Choosing k can be a hard problem, but not in EDA: we use $k = 2$ because two is the best number of dimensions for graphs. So in other words, we're displaying a data set of lots of variables on a plane.

My preferred function for displaying PCA is `ggbiplot()` in the package of the same name. This isn't on CRAN, so you'll need to install it from Github:

```r
library(devtools)
install_github("vqv/ggbiplot")
```

Technical note: `ggbiplot` requires `plyr`, which clashes with `dplyr`. So load `ggbiplot` before you load `dplyr` or `tidyverse`.

11.4.1 Wine

The `ggbiplot` package contains a data set on 13 chemical measurements on the composition of a bunch of red wines from the Piedmont in Italy, which I'm visiting on holiday in May. Details are at:

https://archive.ics.uci.edu/ml/datasets/Wine

```r
library(ggbiplot)
data(wine)
nrow(wine)
```

```
## [1] 178
```

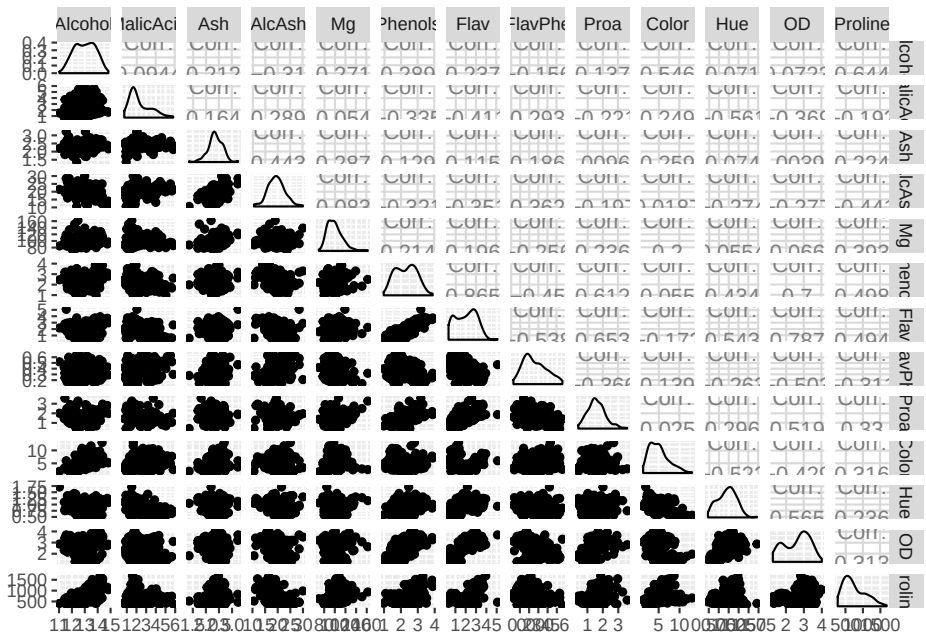
```r
summary(wine)
```

```
##     Alcohol        MalicAcid          Ash            AlcAsh      
##  Min.   :11.03   Min.   :0.740   Min.   :1.360   Min.   :10.60  
##  1st Qu.:12.36   1st Qu.:1.603   1st Qu.:2.210   1st Qu.:17.20  
##  Median :13.05   Median :1.865   Median :2.360   Median :19.50  
##  Mean   :13.00   Mean   :2.336   Mean   :2.367   Mean   :19.49  
##  3rd Qu.:13.68   3rd Qu.:3.083   3rd Qu.:2.558   3rd Qu.:21.50  
##  Max.   :14.83   Max.   :5.800   Max.   :3.230   Max.   :30.00  
##        Mg            Phenols           Flav       NonFlavPhenols
##  Min.   : 70.00   Min.   :0.980   Min.   :0.340   Min.   :0.1300  
##  1st Qu.: 88.00   1st Qu.:1.742   1st Qu.:1.205   1st Qu.:0.2700  
##  Median : 98.00   Median :2.355   Median :2.135   Median :0.3400  
##  Mean   : 99.74   Mean   :2.295   Mean   :2.029   Mean   :0.3619  
##  3rd Qu.:107.00   3rd Qu.:2.800   3rd Qu.:2.875   3rd Qu.:0.4375  
##  Max.   :162.00   Max.   :3.880   Max.   :5.080   Max.   :0.6600  
##       Proa           Color             Hue               OD      
##  Min.   :0.410   Min.   : 1.280   Min.   :0.4800   Min.   :1.270  
##  1st Qu.:1.250   1st Qu.: 3.220   1st Qu.:0.7825   1st Qu.:1.938  
##  Median :1.555   Median : 4.690   Median :0.9650   Median :2.780  
##  Mean   :1.591   Mean   : 5.058   Mean   :0.9574   Mean   :2.612  
##  3rd Qu.:1.950   3rd Qu.: 6.200   3rd Qu.:1.1200   3rd Qu.:3.170  
##  Max.   :3.580   Max.   :13.000   Max.   :1.7100   Max.   :4.000  
##     Proline      
##  Min.   : 278.0  
##  1st Qu.: 500.5  
##  Median : 673.5  
##  Mean   : 746.9  
```

```
##  3rd Qu.: 985.0
##  Max.   :1680.0
```

We'd expect most of the variables to be associated with most of the other variables. 13 is a pretty large number of variable. We can still draw a pairs plot:

```
library(GGally)
ggpairs(wine)
```



but it's too crowded to see much (plus it has a non-trivial chance of crashing RStudio.) If I knew about the chemistry of wine I might try to choose variables I thought were important, but I don't, so let's do dimension reduction via PCA.

The `prcomp()` function runs PCA. The main decision to make is whether to center and scale the data: the function's default is to center but not scale. However, rescaling is almost the right thing to do when the variables are not all measuring the same thing.

```
wine.pca = prcomp(wine, scale. = TRUE)
```

(Note the dot after scale.) The `wine.pca` object we've created contains two main things of interest. One is a `rotation` matrix that give a new coordinate system for the data, the **principal components**, such that all the components are orthogonal.

```
wine.pca$rotation[, 1:3]
```

```
##                              PC1           PC2           PC3
```

```
## Alcohol        -0.144329395  0.483651548 -0.20738262
## MalicAcid       0.245187580  0.224930935  0.08901289
## Ash             0.002051061  0.316068814  0.62622390
## AlcAsh          0.239320405 -0.010590502  0.61208035
## Mg             -0.141992042  0.299634003  0.13075693
## Phenols        -0.394660845  0.065039512  0.14617896
## Flav           -0.422934297 -0.003359812  0.15068190
## NonFlavPhenols  0.298533103  0.028779488  0.17036816
## Proa           -0.313429488  0.039301722  0.14945431
## Color           0.088616705  0.529995672 -0.13730621
## Hue            -0.296714564 -0.279235148  0.08522192
## OD             -0.376167411 -0.164496193  0.16600459
## Proline        -0.286752227  0.364902832 -0.12674592
```

Roughly, out of all the 13-dimensional directions through the origin you could project the scaled data on to, PC1 is the direction that results in the most variance retained, i.e. in the projected data with the highest variance. PC2 is the direction out of all those orthogonal to PC1 that results in the most variance retained, and so on. (More rigorously, the PCs are the eigenvectors of the covariance matrix of the scaled data, sorted by decreasing eigenvalue.) Thus if we want a two-dimensional representation of the data, then projecting the data on to PC1 and PC2 gives the representation that retains the most variance in this sense. Note that different implementations of PCA may flip the signs of the vectors; these are arbitrary. In many uses of PCA, we wish to reduce the dimensionality of the data, and thus "how many principal components should we keep?" becomes a question to agonize over. However, in this course we mostly want to draw graphs, so our answer will simply be "two, or very occasionally three."

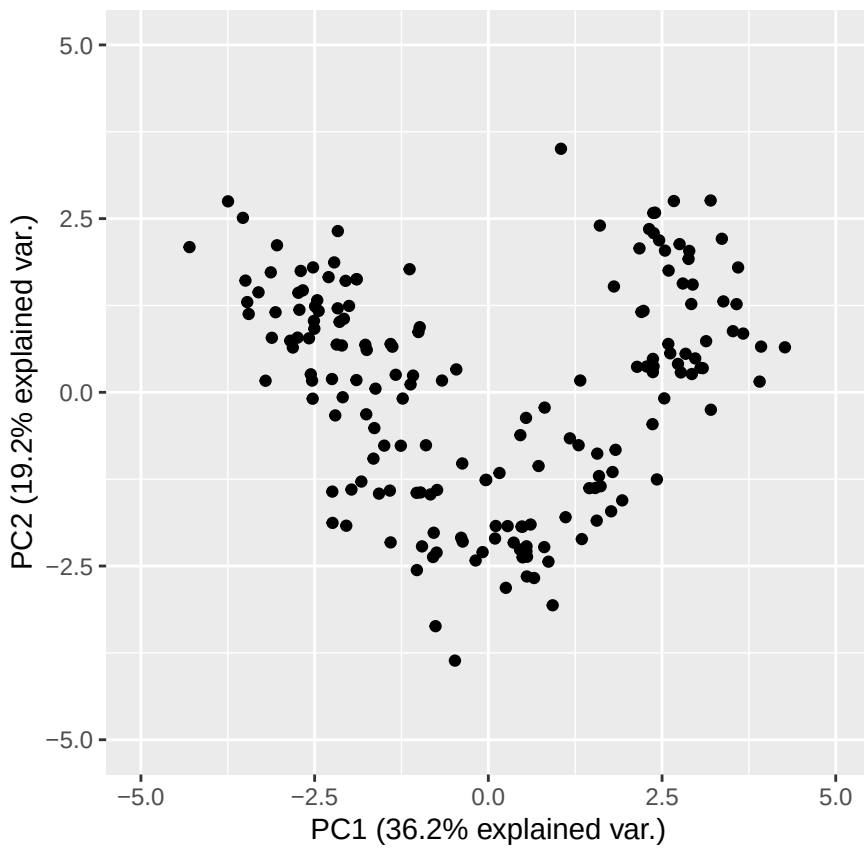
The coordinates of the observations in this new system are stored in the `x` matrix inside `wine.pca`:

```
wine.pca$x[1:10, 1:2]
```

```
##             PC1        PC2
##  [1,] -3.307421  1.4394023
##  [2,] -2.203250 -0.3324551
##  [3,] -2.509661  1.0282507
##  [4,] -3.746497  2.7486184
##  [5,] -1.006070  0.8673840
##  [6,] -3.041674  2.1164309
##  [7,] -2.442201  1.1715453
##  [8,] -2.053644  1.6044371
##  [9,] -2.503811  0.9154885
## [10,] -2.745882  0.7872170
```

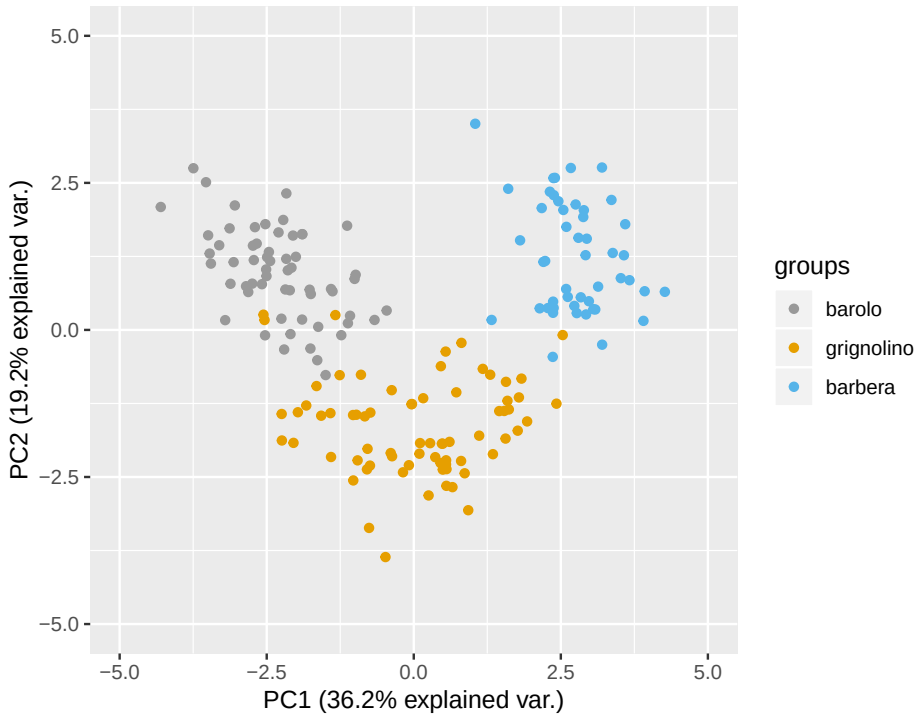
We now use `ggbiplot()` to visualize the data projected on the first two principal components. We first leave out the "biplot" part and just plot the points.

```
library(ggbiplot)
ggbiplot(wine.pca, obs.scale = 1, var.axes = FALSE) + xlim(-5, 5) + ylim(-5, 5)
```



`obs.scale = 1` prevent the rescaling of the principal components and `var.axes = FALSE` suppresses the biplot. We see that on this two-dimensional representation, there's a U-shape. For this to be useful, we need to relate this back to the wines. There's a data variable called `wine.class` that gives the varietal of the wines. We can group according to this variable:

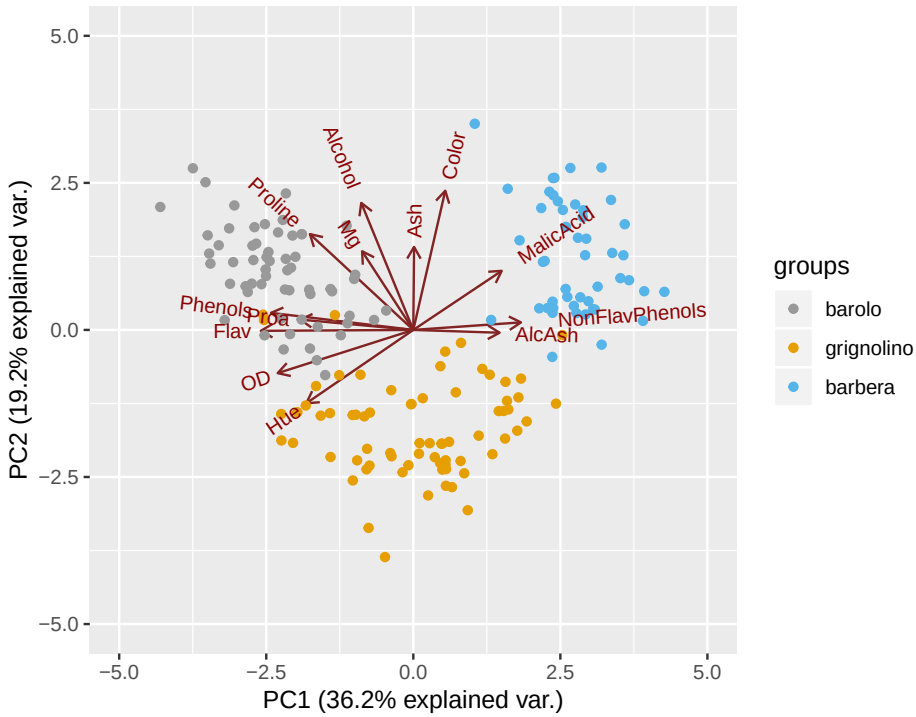
```
ggbiplot(wine.pca, obs.scale = 1, var.axes = FALSE, groups = wine.class) +
  xlim(-5, 5) + ylim(-5, 5) + cbp()
```



We can now see an ordering: Barolo, Grignolino, Barbera. Barolo is a full-bodied red wine and Barbera is medium-bodied. I don't know anything about Grignolino but it looks like it's not simply in between the other two. The separation is fairly clear, with only a small amount of overlap between Barolo and Grignolino and between Grignolino and Barbera. (Note that if our main goal was classification, *linear discriminant analysis (LDA)* or one of its many variants might be a better approach than PCA.)

In addition to displaying the data, we also want to interpret how this new coordinate system relates to the original variables. This is already encoded in our `rotation` matrix. Where the columns of the rotation matrix give the meanings of each principal components, the rows tell us the *directions* associated with each of the original variables in the new system. That is, if some wines were average on all variables besides alcohol, the alcohol row (sometimes called a *loading*) would give the line along which all these wines would fall in the new system. To show these directions, we add the **biplot**:

```
ggbiplot(wine.pca, obs.scale = 1, groups = wine.class) +
  xlim(-5, 5) + ylim(-5, 5) + cbp()
```

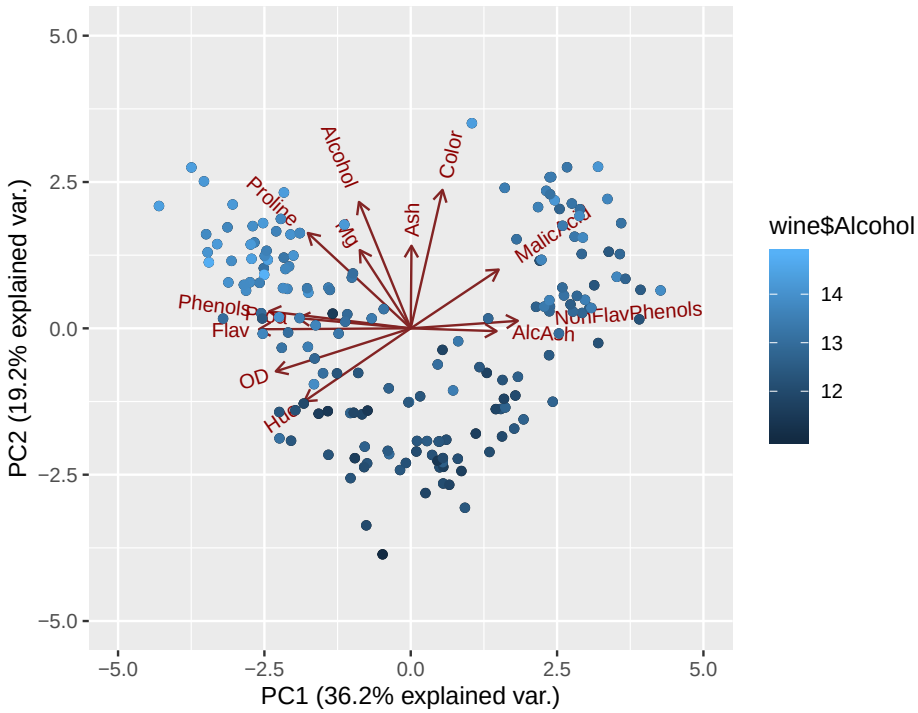


The direction of the arrows tells you in which direction values of that variable are generally increasing. So `AlcAsh` (the alkalinity of the ash in the wine) increases as you go right. `Flav` (flavanoids) increase as you go left, `Ash` increases as you go up, and so on.

The length of the arrows gives the magnitude correlation of the variables with the two principal components. So `Flav` has a relatively strong relationship with the two principal components, compared to, say, `Ash`. (Note that the arrows are not on the same numerical scale as the observations.)

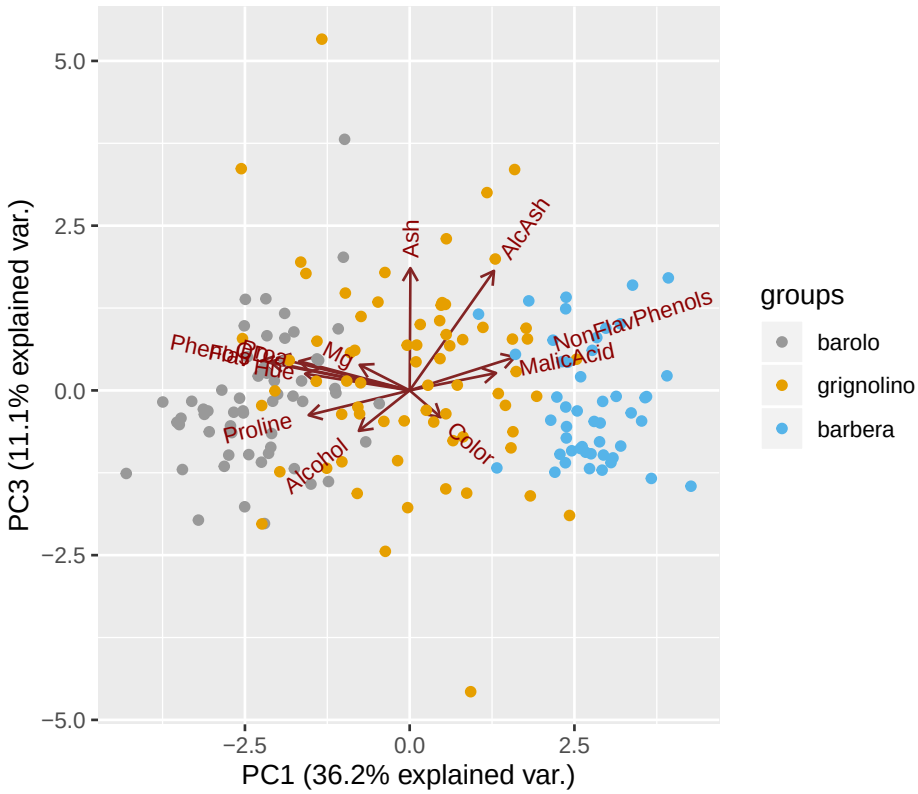
You can also use `geom_point()` and so on to control the appearance of the graph:

```
ggbiplot(wine.pca, obs.scale = 1) + geom_point(aes(color = wine$Alcohol)) +
  xlim(-5, 5) + ylim(-5, 5)
```



Finally, you could also plot principal components other than the first two:

```
ggbiplot(wine.pca, obs.scale = 1, groups = wine.class, choices = c(1,3)) + cbp()
```



We see that using the third principal component doesn't give much help with separation (over and above what the first component gives.)

11.5 PCA vs. factor analysis

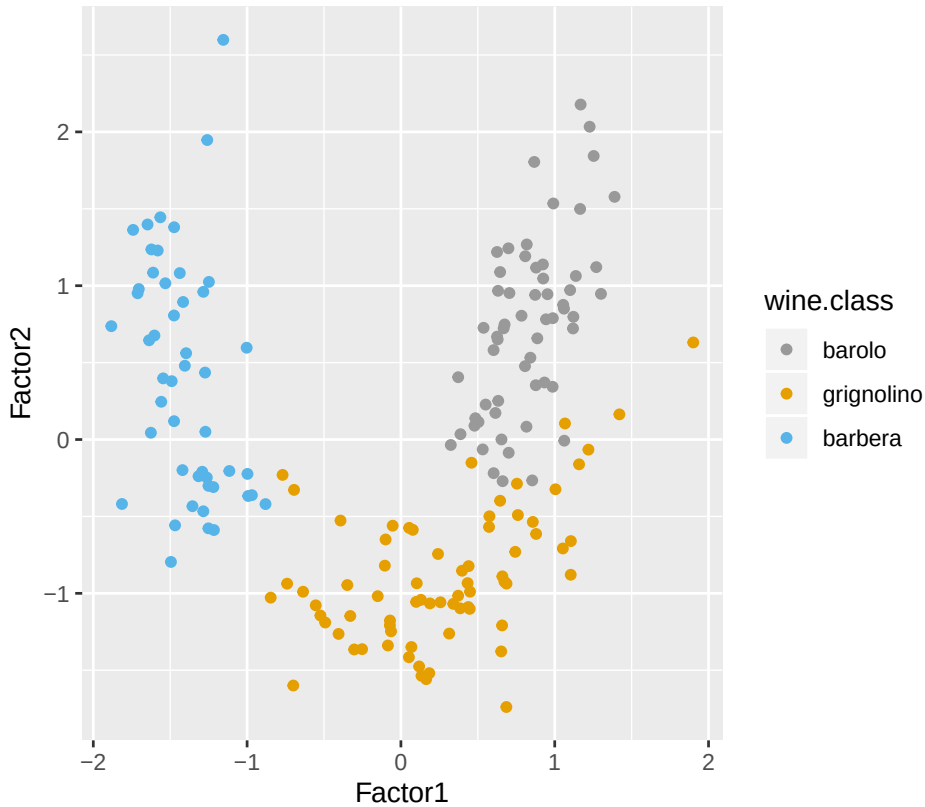
Factor analysis is a technique closely related to PCA, with some subtle differences. The conceptual difference between the two can be summarized as follows:

- *PCA:* Determine a sequence of orthogonal directions such that each direction explains the maximum amount of variance not yet explained by the directions you've already chosen.
- *Factor analysis:* Assume a model such that the data consists of points on a k-dimensional subspace plus some probabilistic error term. Estimate that k-dimensional subspace. (Since we're estimating a subspace rather than directions, the subspace can be rotated arbitrarily, though there are conventions that can make the subspace line up with PCA to some extent.)

Sometimes, but not always, these two methods give similar results (conditional on the number of components/factors,) but the interpretation can be quite different.

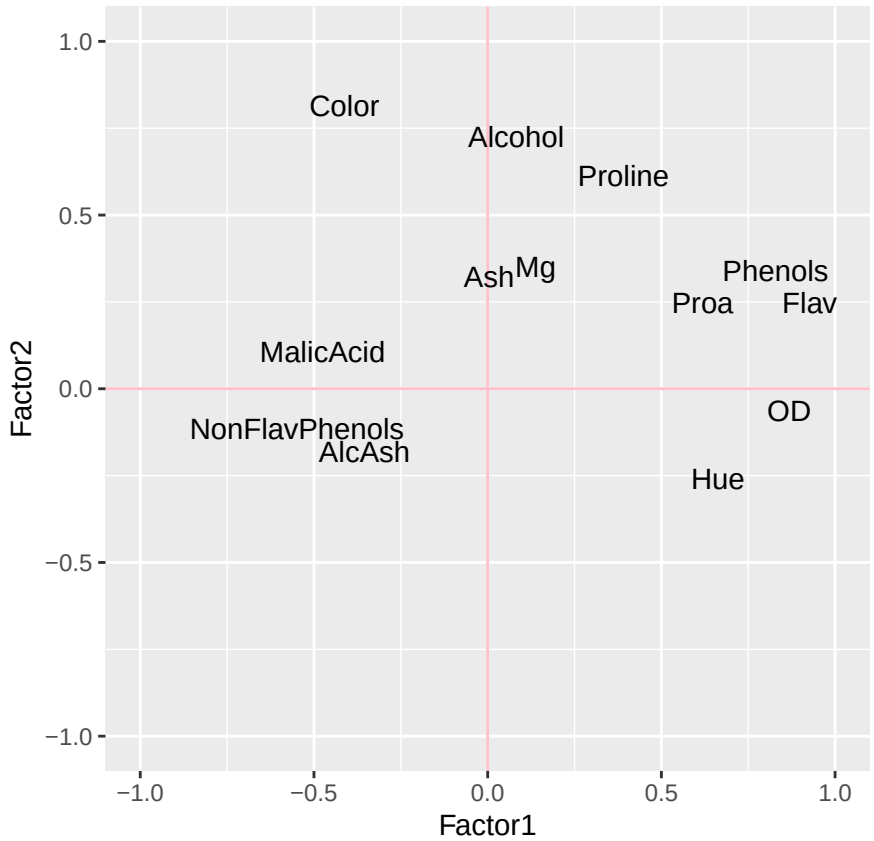
The base R implementation of factor analysis is the unfortunately named `factanal()` function. Below, we fit a model with two orthogonal "factors" and specifying we want "regression" scores, which give us the locations of the individuals in the new two-dimensional coordinate system.

```
wine.2factor = factanal(wine, factors = 2, scores = "regression")
wine.2factor.df = data.frame(wine.2factor$scores, wine.class)
ggplot(wine.2factor.df, aes(x = Factor1, y = Factor2, color = wine.class)) +
  geom_point() + coord_fixed() + cbp()
```



This looks kind of the same as our plot of the wines in the PCA co-ordinate system (with the first component flipped), but a closer look shows the points are not in the same relative positions. This becomes a bit clearer if we look at the variable loadings:

```
wine.loadings = loadings(wine.2factor)[1:13,]
wine.loadings.df = data.frame(variable = row.names(wine.loadings), wine.loadings)
ggplot(wine.loadings.df, aes(x = Factor1, y = Factor2, label = variable)) +
  geom_vline(xintercept = 0, color = "pink") +
  geom_hline(yintercept = 0, color = "pink") +
  geom_text() + coord_fixed() + xlim(-1, 1) + ylim(-1, 1)
```



Here we see that "Ash" and "Alcohol" are pointing in the same direction, whereas in the PCA biplot they were pointing in quite different directions.

In both cases, for EDA purposes all we're doing is representing a high-dimensional data set in a small number of dimensions, so it's hard to say whether PCA or factor analysis is "right" or "wrong." A big practical difference is that PCA doesn't require you to specify a number of components in advance, whereas factor analysis does make you decide on the number of factors in advance. Of course, you could just try out every possible number of factors and see which number gives you the best results, but this feels like cheating in a way that it doesn't in PCA. Remember, the assumption behind factor analysis is that the true pattern really does lie on a low-dimensional subspace, and it seems heroic to make this assumption if you don't even know how many dimensions this subspace has.

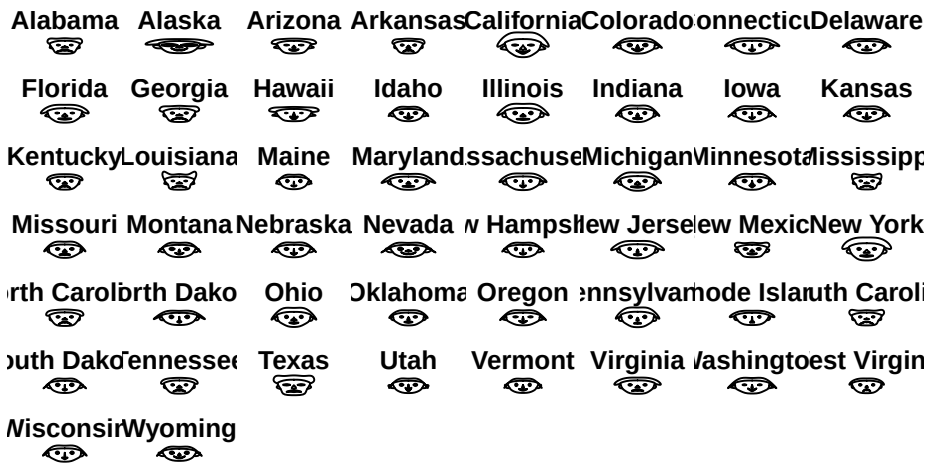
Interpretations of factor analysis and, to some extent, PCA too) also seem to run into the issue of – here's a word I learned from the Department of History and Philosophy of Science across the hall in Ballantine – *reification*, where the factors are taken to be actually existing things. In personality studies, for example, factor analysis led to the Big Five model, which is an efficient 5-dimensional

summary of personality. The temptation, however, is to then immediately conclude "personality really is five dimensional, and these are the five dimensions." However, this isn't good reasoning unless you've done a confirmatory analysis to back it up. (From my not-thorough-at-all understanding of the psychometric literature, the Big Five model failed confirmatory analyses, which the researchers took to indicate that you shouldn't do confirmatory analysis.)

11.6 Chernoff faces

These aren't actually useful but are amusing.

```
library(TeachingDemos)
faces(state.x77)
```



11.7 Example: The 88th Congress

The 88th Congress, which met in 1963–1964, passed many important pieces of legislation, including the Civil Rights Act, the Economic Opportunity Act, the Food Stamp Act, the Clean Air Act, and the Tonkin Gulf Resolution. It's especially interesting because many of these key votes were not along party lines. Data on all of the roll call votes held is on the site

http://www.voteview.com/house88.htm

We can download the Stata file, import it, and reformat it as a data frame (or a tibble if you prefer):

```
library(rio)
house88 = import("hou88kh.dta")
```

There were 232 roll call votes, coded 0 to 9 (see the webpage for details.) We can recode as follows:

- Let 1 mean some kind of Yes;
- Let −1 mean some kind of No;
- Everything else is a 0.

(This could be a bit misleading, as the zeroes don't distinguish between people who abstained from a vote and people who couldn't vote because they weren't in Congress or were dead, but we'll keep it simple.) Let's create a new matrix called `votes6364` to contain this new encoding.

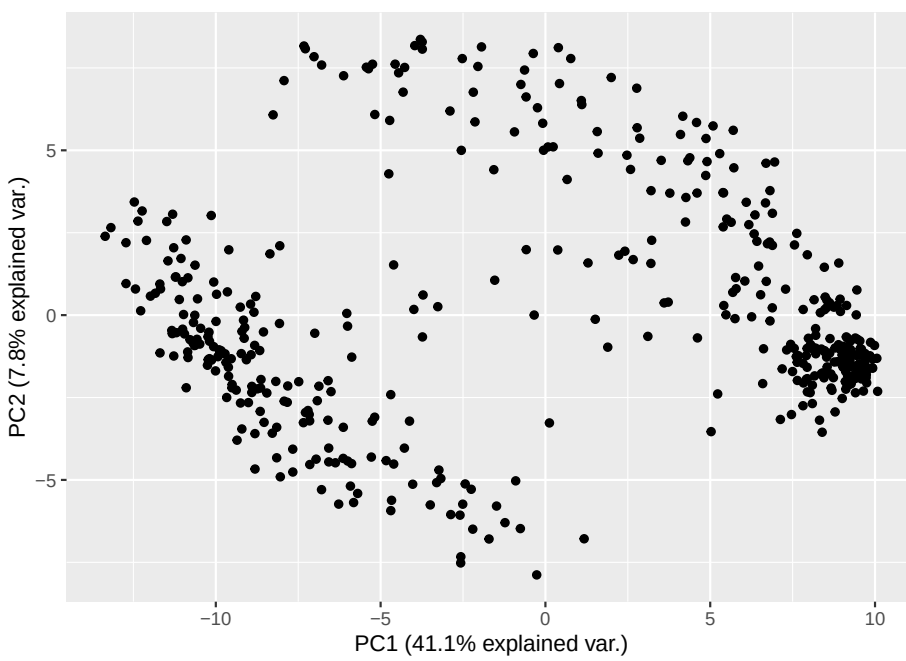
```
votes6364 = matrix(0, nrow = 445, ncol = 232)
votes6364[house88[,10:232] == 1] = 1
votes6364[house88[,10:232] == 2] = 1
votes6364[house88[,10:232] == 3] = 1
votes6364[house88[,10:232] == 4] = -1
votes6364[house88[,10:232] == 5] = -1
votes6364[house88[,10:232] == 6] = -1
```

Everything's now on the same scale, so rescaling isn't necessary. Run PCA:

```
votes.pca = prcomp(votes6364)
```

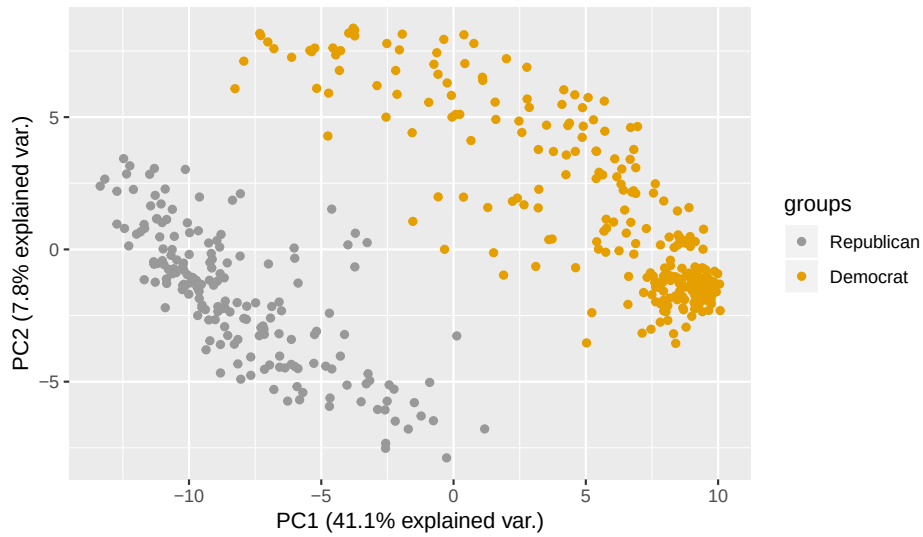
Now display the results, suppressing the biplot since we don't want 232 arrows cluttering our graph.

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE)
```



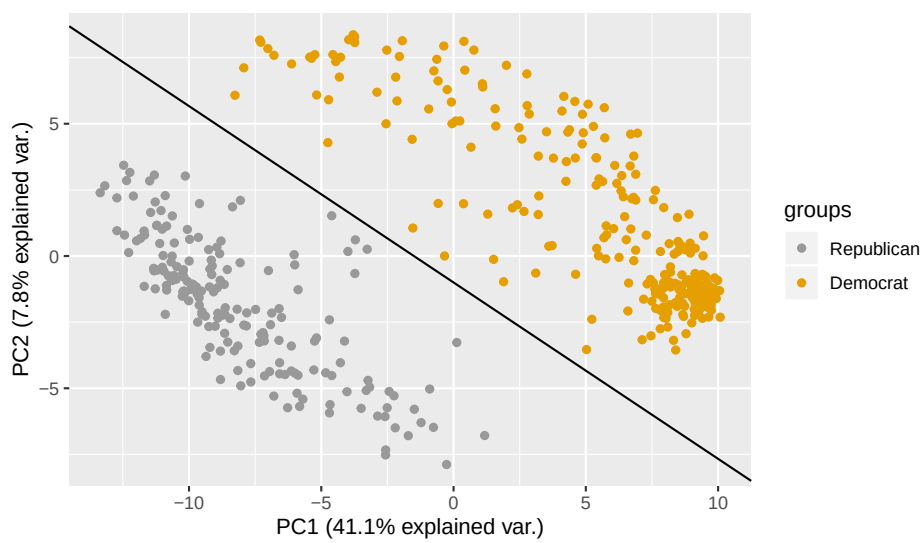
Recode the `party` variable to use as group labels:

```
party = house88$party
party[party == 100] = "Democrat"
party[party == 200] = "Republican"
party = factor(party, levels = c("Republican", "Democrat"))
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE, group = party) + cbp()
```



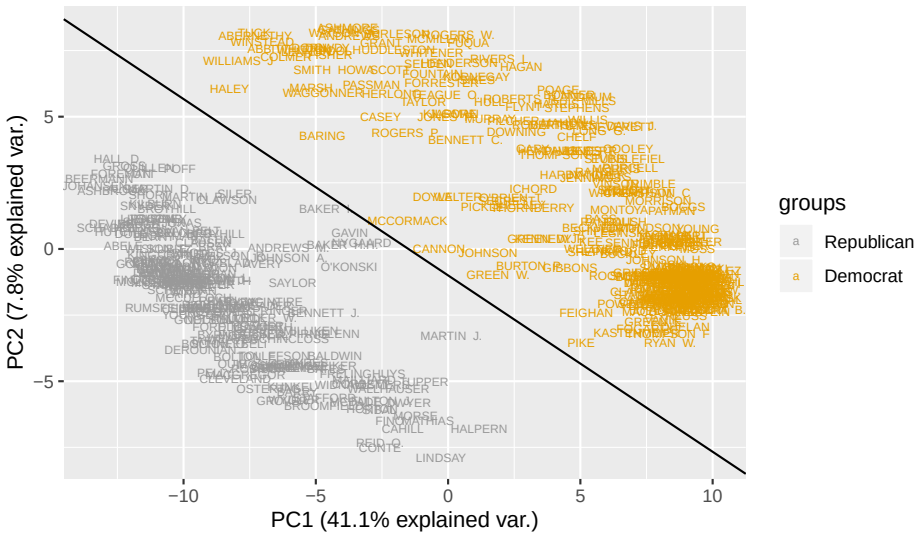
We see pretty clear separation between Republicans and Democrats. Eyeball a line:

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE, group = party) +
  geom_abline(intercept = -1, slope = -2/3) + cbp()
```



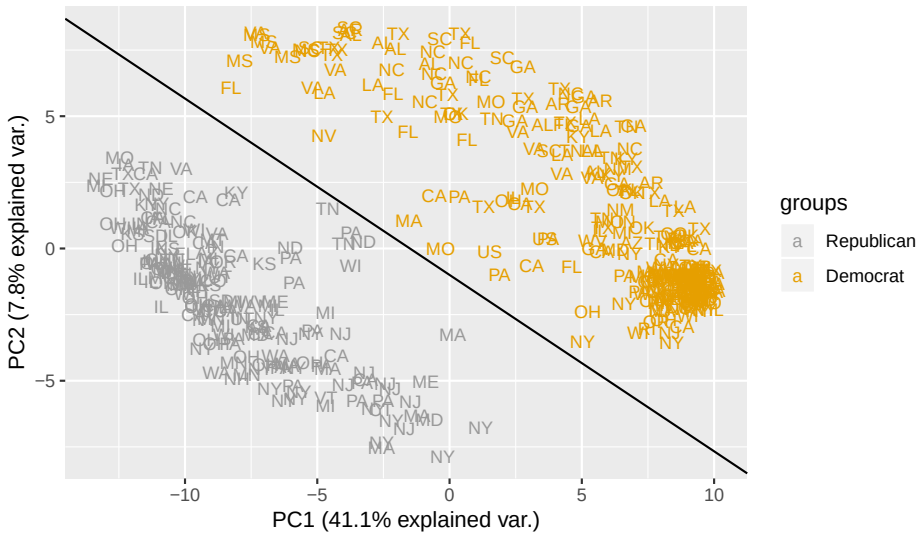
We can add names:

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = party, labels = house88$name, labels.size = 2) +
  geom_abline(intercept = -1, slope = -2/3) + cbp()
```



but these don't mean much unless you're really up to speed with your historical members of Congress. Instead, we find the state codes and use those as labels:

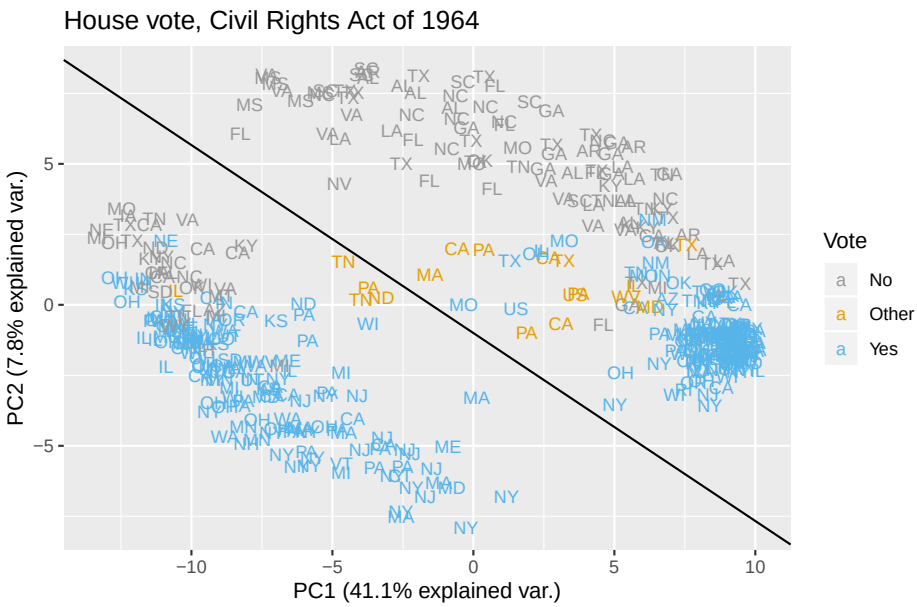
```
statecodes = read.table("statecodes.txt")
state = rep(NA, length(house88$state))
# Too lazy to work out how to do this efficiently
# so use a for loop
for(J in statecodes$V1){
  code = statecodes$V2[statecodes$V1 == J]
  state[house88$state == J] = as.character(code)
}
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = party, labels = state, labels.size = 3) +
  geom_abline(intercept = -1, slope = -2/3) + cbp()
```



In general, northeastern states are toward the bottom-right, while southern and Midwestern states tend to be toward the top left.

We can now look at the results of the key votes we noted at the beginning of the section. Start with the Civil Rights Act. According to ftp://k7moa.com/dtl/88.dtl, this was vote number 128, so we make our group label column 128 of `votes6364`.

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = factor(votes6364[,128]), labels = state, labels.size = 3) +
  geom_abline(intercept = -1, slope = -2/3) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Civil Rights Act of 1964")
```



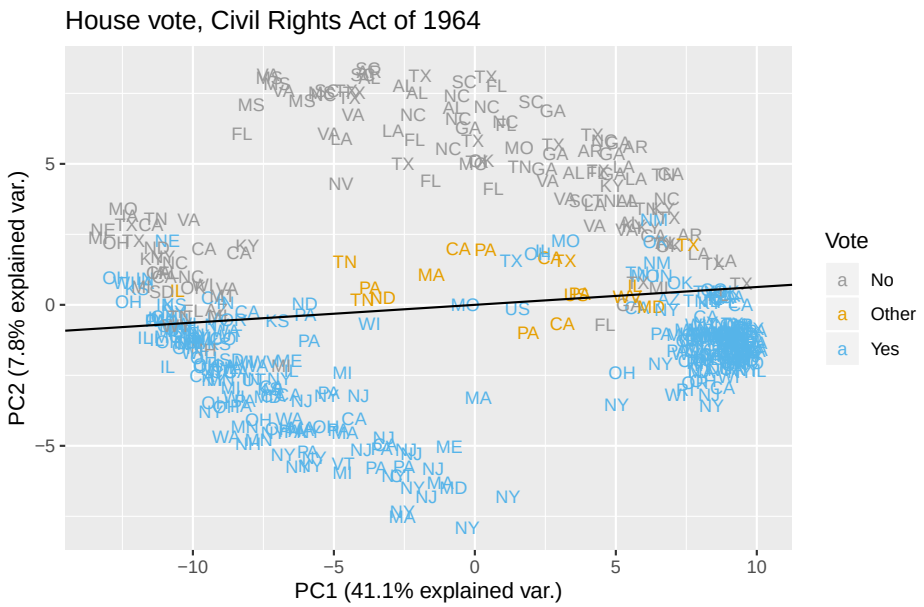
The vote here is almost entirely explained by the second component. The direction of the biplot arrow for this vote is given by the first two numbers of row 128 of the `rotation` matrix produced by the PCA:

```
votes.pca$rotation[128, 1:2]
```

```
##          PC1         PC2
##  0.01627373 -0.21041825
```

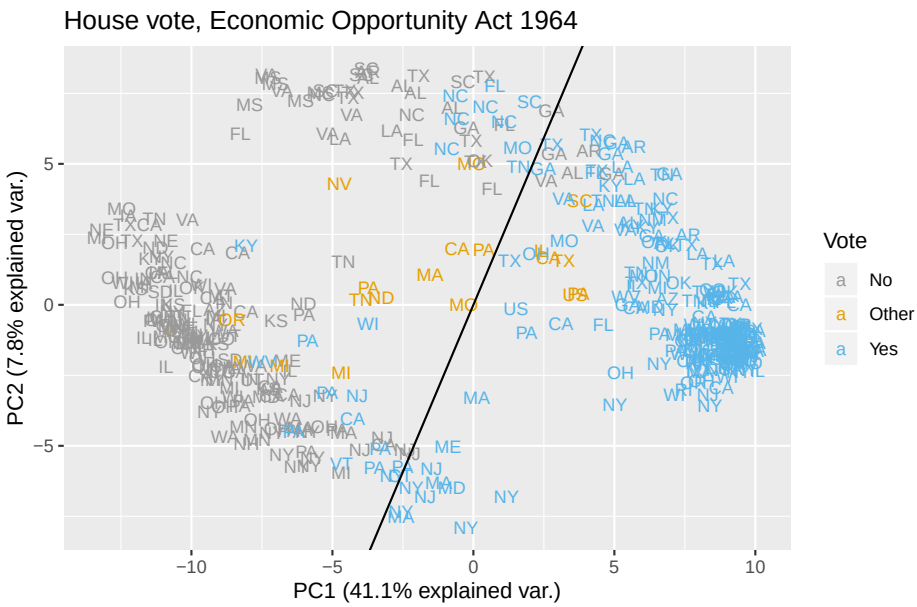
But it might be helpful to add a line *perpendicular* to this to separate ayes from nays. We draw such a line through the origin (though this won't always be the best choice):

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = factor(votes6364[,128]), labels = state, labels.size = 3) +
  geom_abline(slope = - votes.pca$rotation[1,128] / votes.pca$rotation[2,128]) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Civil Rights Act of 1964")
```



The Economic Opportunity Act (vote 201):

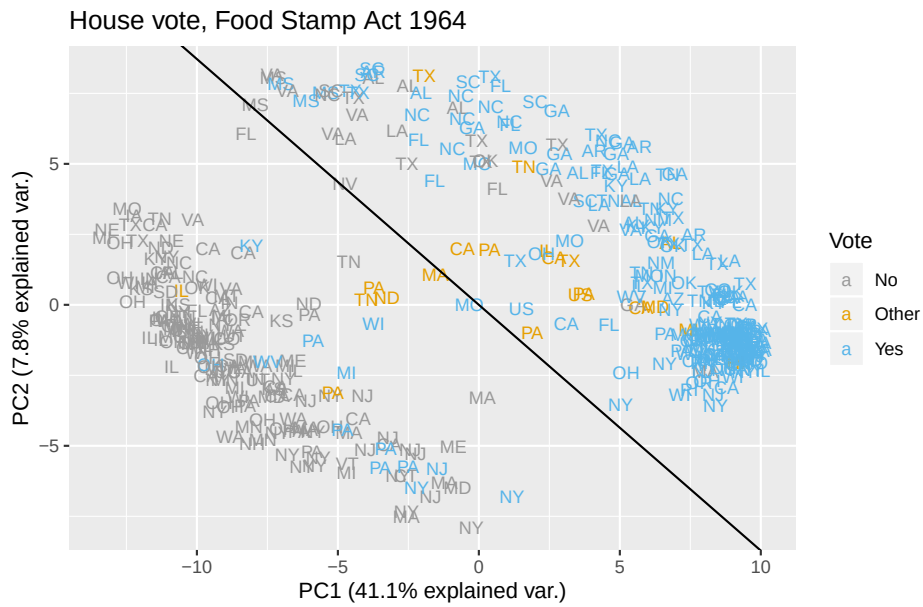
```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = factor(votes6364[,201]), labels = state, labels.size = 3) +
  geom_abline(slope = - votes.pca$rotation[1,201] / votes.pca$rotation[2,201]) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Economic Opportunity Act 1964")
```



This vote is better explained by the first component.

The Food Stamp Act (vote 149):

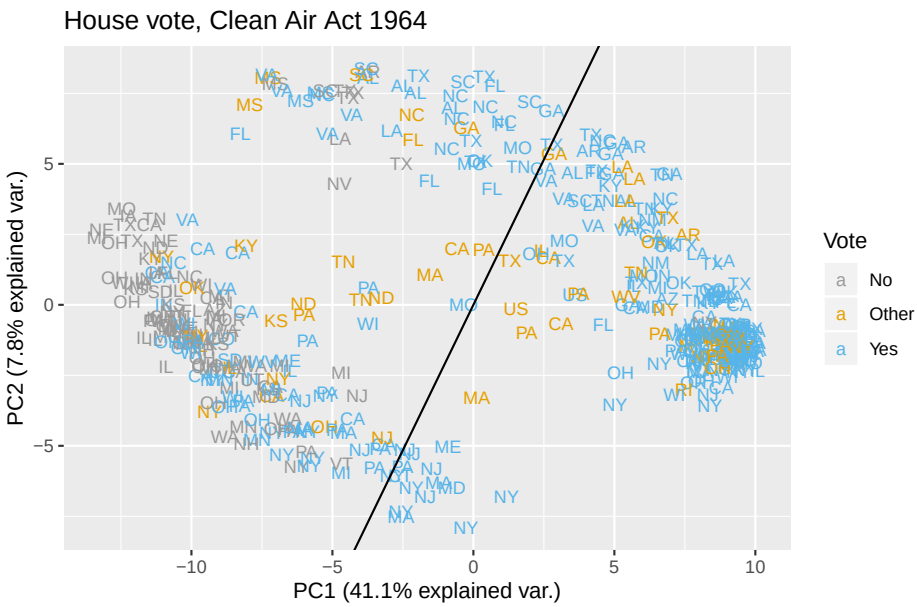
```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = factor(votes6364[,149]), labels = state, labels.size = 3) +
  geom_abline(slope = - votes.pca$rotation[1,149] / votes.pca$rotation[2,149]) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Food Stamp Act 1964")
```



This is a bit closer to being along party lines.

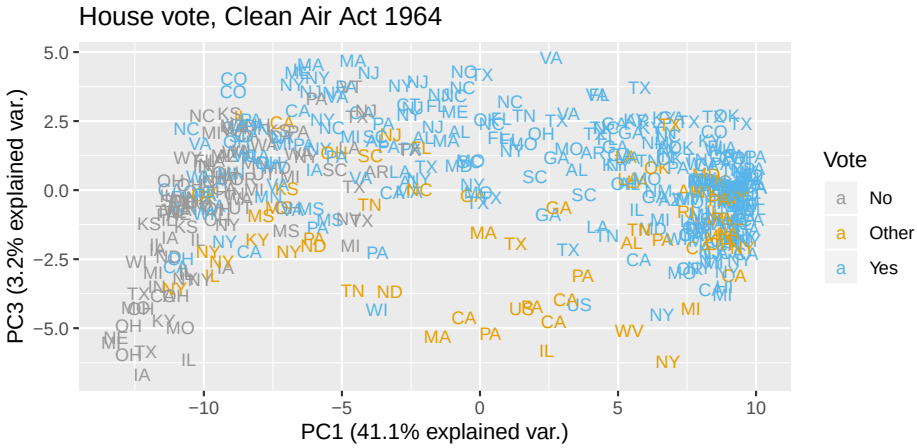
The Clean Air Act (vote 47):

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = factor(votes6364[,47]), labels = state, labels.size = 3) +
  geom_abline(slope = - votes.pca$rotation[1,47] / votes.pca$rotation[2,47]) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Clean Air Act 1964")
```



The line is not doing a good job here. Most Democrats voted yes, but the behavior of Republicans is more complicated, though No votes seem to become more common as we go left. We can try going to the third component:

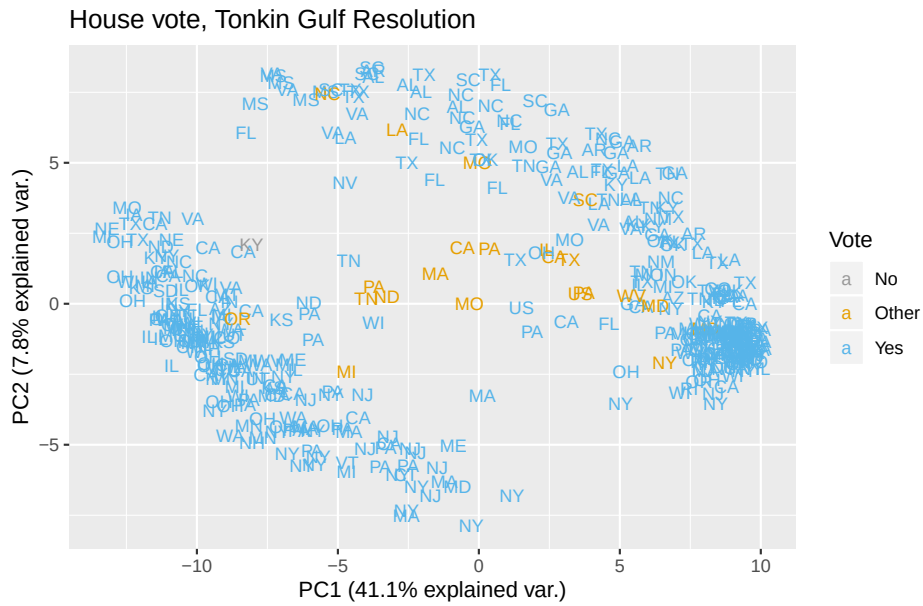
```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE, choice = c(1,3),
    group = factor(votes6364[,47]), labels = state, labels.size = 3) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Clean Air Act 1964")
```



but this doesn't necessarily make things much clearer. (At this point you just phone a political historian.)

The Tonkin Gulf Resolution (vote 197):

```
ggbiplot(votes.pca, obs.scale = 1, var.axes = FALSE,
    group = factor(votes6364[,197]), labels = state, labels.size = 3) +
  scale_color_manual(values = cb_palette, name = "Vote",
    labels = c("No", "Other", "Yes")) +
  ggtitle("House vote, Tonkin Gulf Resolution")
```



Just about everyone voted yes except Eugene Siler (who didn't show up for the vote but was a "paired no.")

It seems that the first component represents something like economic ideology and the second component represents something like social ideology. If you think there really are latent random variables called "economic ideology" and "social ideology", then factor analysis might be a good alternative (but, as usual, you should check your model and be extremely hesitant to make causal claims.)

Chapter 12

Random effects

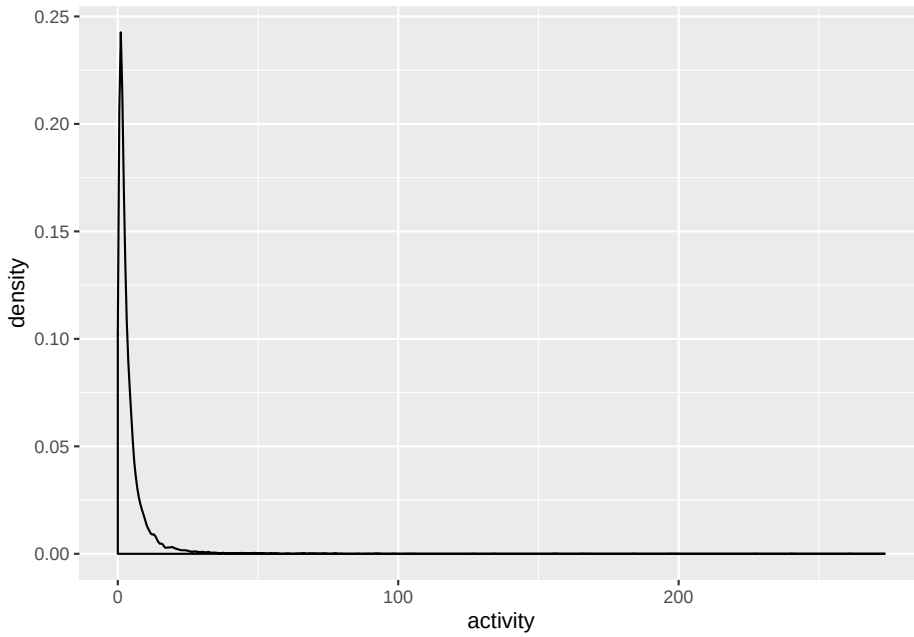
12.1 Radon

This example comes from Gelman & Hill ch. 12. In much of North America, background levels of radon, a radioactive gas formed by decay of elements such as uranium, can be dangerously high (particularly in basements.) The file `srrs2.dat` contains data on the radioactivity measured in a sample of 12,777 U.S. households, along with a collection of covariates. The response variable is `activity`, a measurement of the concentration of radon (in picocuries per liter, I think; the EPA recommends taking action at levels above 4 pci/L.) Read in the data:

```
radon.df = read.table("srrs2.dat", header=T, sep=",")
summary(radon.df$activity)
```

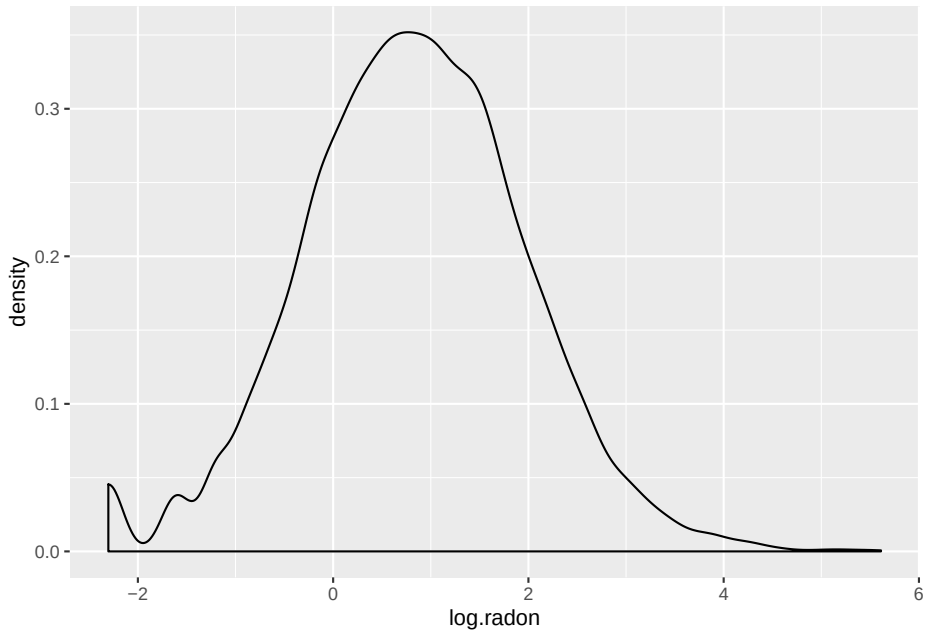
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.10    2.30    4.48    4.80  273.50
```

```
library(ggplot2)
ggplot(radon.df, aes(x = activity)) + geom_density()
```



The distribution of radon is insanely skewed. We'd like to take logs, but the variable contains zeros. Well, we can just replace the zeros with the next lowest measurement (0.1) before taking logs. We'll do natural logs because that's what Gelman & Hill do.

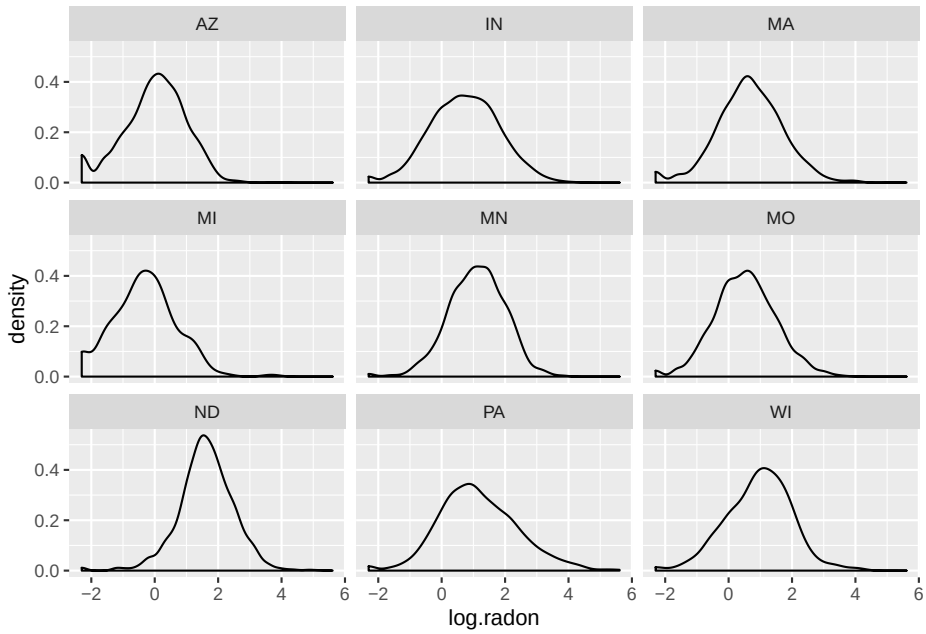
```
radon.df$log.radon = log(ifelse(radon.df$activity==0, 0.1, radon.df$activity))
ggplot(radon.df, aes(x = log.radon)) + geom_density()
```



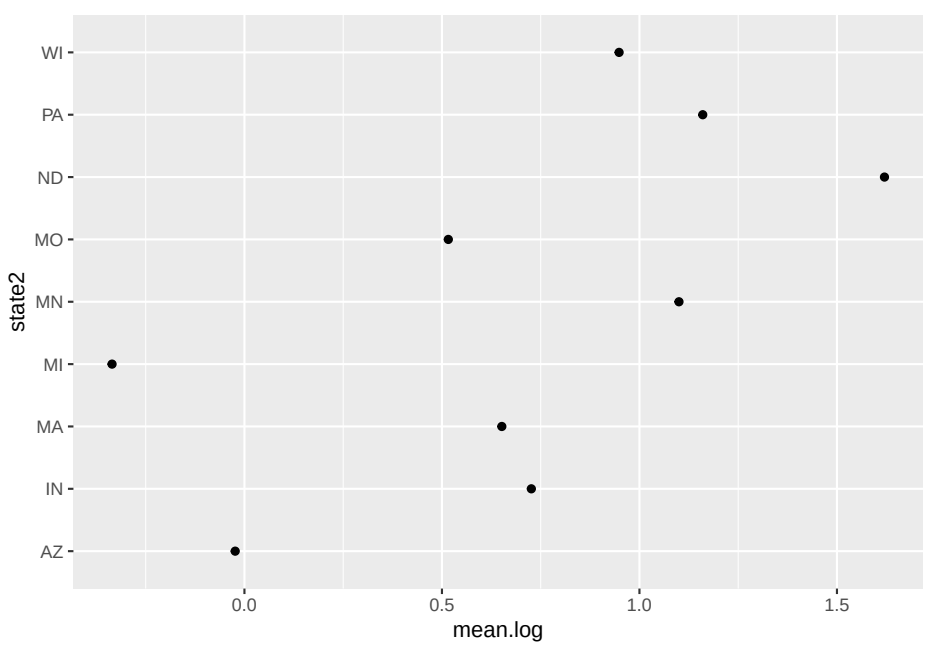
The distribution is now somewhat normalized, save for the unavoidable bump at log(0.1).

The variable `state2` tells us which state each observation was collected in. How much difference is there in radon levels by state?

```
ggplot(radon.df, aes(x = log.radon)) + geom_density() + facet_wrap(~state2, ncol=3)
```




```
library(dplyr)
radon.by.state = summarize(group_by(radon.df, state2), mean.log = mean(log.radon))
ggplot(radon.by.state, aes(x = mean.log, y = state2)) + geom_point()
```



There's substantial difference by state: North Dakota has a much higher typical

radon level than Michigan.

We could also describe the geography by county. The data set has both a numeric (`cntyfips`) and a character version (`county`) of county. Since I don't know much county-level geography I'll use the numeric version. The numbers aren't unique across states; to get a unique version, add a two-digit prefix for the state (in `stfips`.)

```
radon.df$county.no = radon.df$stfips * 1000 + radon.df$cntyfips
```

Let's pick out one state and examine it in more detail. Following Gelman & Hill, we'll do Minnesota, which might be because Minnesota works out nicely but whatever. (G&H use `state` rather than `state2` if you're trying to reporduce their work.)

```
minnesota.df = subset(radon.df, state2 == "MN")
```

One potentially important explanatory variable is `floor`, which tells us on which floor on the builidng the radon measurement was taken. Here "0" means the basement, "1" means the first floor, and so on; "9" indicates a missing value.

```
table(minnesota.df$floor)
```

```
##
##   0   1   9
## 920 267   1
```

Change "9" to missing value:

```
minnesota.df$floor[minnesota.df$floor == 9] = NA
table(minnesota.df$floor, useNA = "always")
```

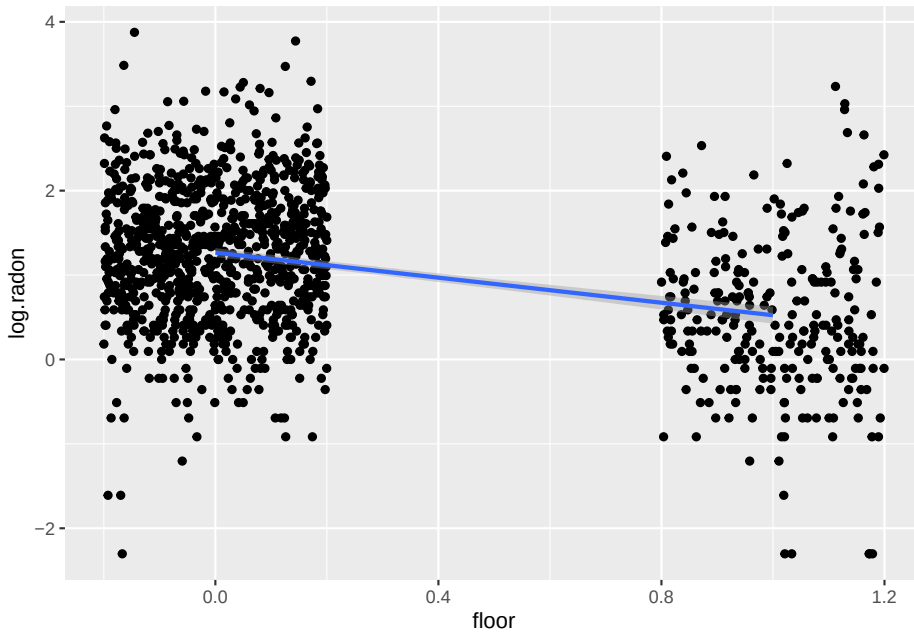
```
##
##    0    1 <NA>
##  920  267    1
```

Now we can look at the relationship between floor and radon:

```
summarize(group_by(minnesota.df, floor), mean.log = mean(log.radon))
```

```
## # A tibble: 3 x 2
##   floor mean.log
##   <int>    <dbl>
## 1     0    1.27
## 2     1    0.524
## 3    NA    2.39
```

```
ggplot(minnesota.df, aes(x = floor, y = log.radon)) + geom_jitter(width = 0.2, height = 0) + geom
```



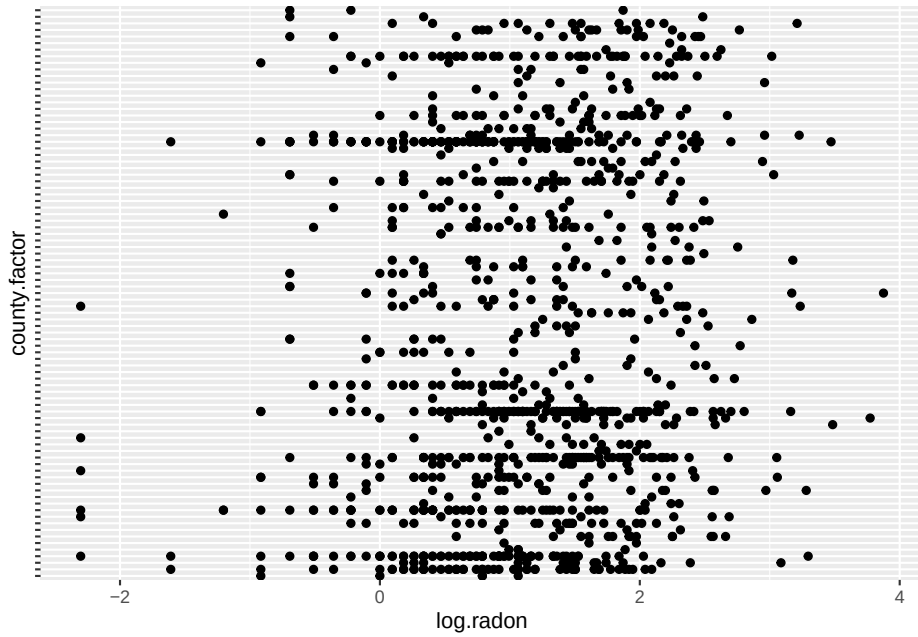

```
minnesota.lm = lm(log.radon ~ floor, data = minnesota.df)
library(arm)
display(minnesota.lm)
```

```
## lm(formula = log.radon ~ floor, data = minnesota.df)
##             coef.est coef.se
## (Intercept)  1.27     0.03
## floor       -0.74     0.06
## ---
## n = 1187, k = 2
## residual sd = 0.83, R-Squared = 0.12
```

The average log radon level is 1.27 in the basement and 1.26 - 0.74 = 0.52 on the first floor (that's 3.5 and 1.7 on the untransformed scale.) Since Minnesota apparently hasn't invented buildings more than one story tall yet, we don't know if the linear pattern can be extrapolated to taller buildings.

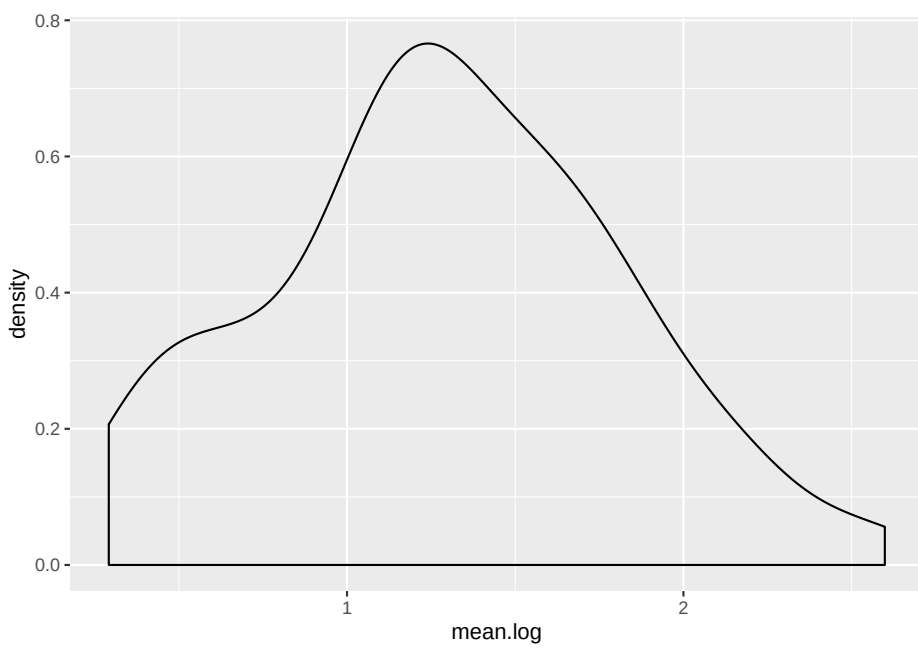
Since we know there's a geographic pattern, the next variable we might try is county. For convenience, we'll create a factor version of the variable.

```
minnesota.df$county.factor = factor(minnesota.df$county.no)
ggplot(minnesota.df, aes(x = log.radon, y = county.factor)) + geom_point() + theme(axis
```

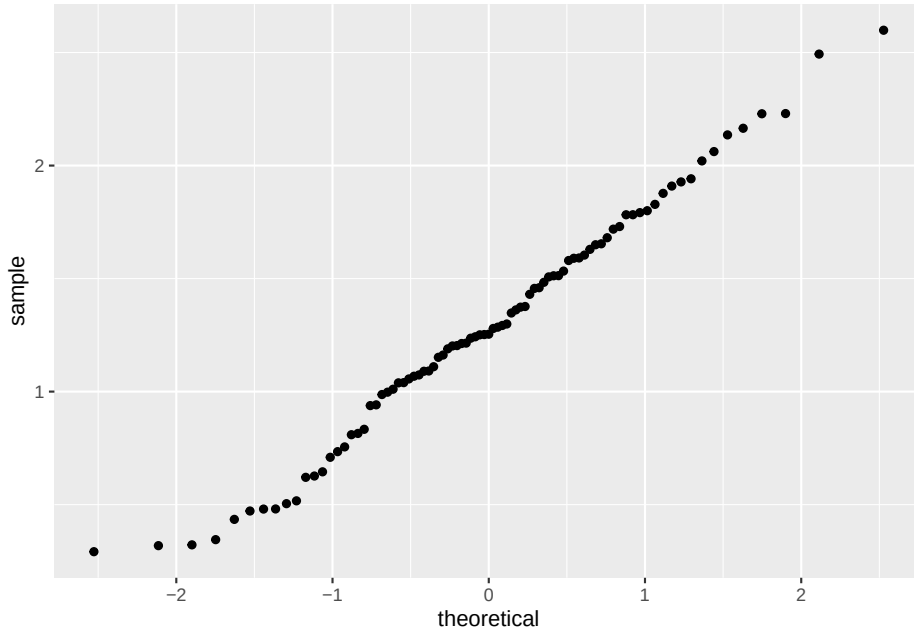


Let's see how the average log radon varies by county:

```
radon.by.county = summarize(group_by(minnesota.df, county.factor), mean.log = mean(log.radon))
ggplot(radon.by.county, aes(x = mean.log)) + geom_density()
```



```
ggplot(radon.by.county, aes(sample = mean.log)) + stat_qq()
```



There's substantial variation by county. Apart from the left tail, the variations seems approximately normal. We could simply put county into our linear model:

```
county.lm = lm(log.radon ~ floor + county.factor, data = minnesota.df)
```

Is there any problem with this? One issue is that the sample sizes vary dramatically by county:

```
table(minnesota.df$county.factor)
```

```
##
## 27001 27003 27005 27007 27009 27011 27013 27015 27017 27019
27021 27023 27025
## 5 52 16 75 4 3 14 4 22 6 60 4 6
## 27027 27029 27031 27033 27035 27037 27039 27041 27043 27045
27047 27049 27053
## 14 9 27 4 12 63 3 9 6 2 9 19 105
## 27055 27057 27059 27061 27063 27065 27067 27069 27071 27073
27075 27077 27079
## 6 5 3 31 5 4 4 3 9 2 9 4 5
## 27081 27083 27085 27087 27089 27091 27093 27095 27097 27099
27101 27103 27105
## 4 8 13 10 9 7 5 8 9 13 1 4 3
## 27107 27109 27111 27113 27115 27117 27119 27121 27123 27127
27129 27131 27133
```

```
## 3 23 8 3 10 4 4 2 32 10 3 11 2
## 27135 27137 27139 27141 27143 27145 27147 27149 27151 27153
27155 27157 27159
## 14 135 17 8 4 25 10 2 4 3 4 7 5
## 27161 27163 27165 27167 27169 27171 27173 27189 27999
## 4 46 3 1 13 13 12 3 3
```

With 135 observations, we should be able to estimate the coefficient for county 27137 (St. Louis County) accurately. On the other hand, some counties only have one or two observations, so we wouldn't expect the coefficients for those counties to be accurately estimated.

12.2 Random effects

A alternative way to model the data is by using a **multilevel model**. This basically means that the county effects are modeled as coming from a probability distribution (by default, normal.) Thus the county is consider to have a **random effect**, rather than the **fixed effect** of `county.lm`. In practice, this regresses the estimates for each county toward the mean for the whole state, with the counties with fewest observations regressed the most heavily.

The `lmer()` function (in package `lme4`) is a quick and dirty way of fitting multilevel models. Firstly, fit a model with just county as a predictor:

```
county.mlm = lmer(log.radon ~ 1 + (1 | county.factor), data = minnesota.df)
display(county.mlm)
```

```
## lmer(formula = log.radon ~ 1 + (1 | county.factor), data =
minnesota.df)
## coef.est coef.se
## 1.24 0.05
##
## Error terms:
## Groups Name Std.Dev.
## county.factor (Intercept) 0.38
## Residual 0.81
## ---
## number of obs: 1188, groups: county.factor, 87
## AIC = 2966.8, DIC = 2952.7
## deviance = 2956.7
```

In this (too) simple model, we see the county effects (after accounting for the overall mean) are modeled as coming from a Normal distribution with mean 0 and standard deviation 0.38. This seems reasonable given our last plot.

Now let's try a model with floor having a fixed effect and county having a random effect. This is what's called a **mixed model**.

```
county.mixed = lmer(log.radon ~ floor + (1 | county.factor), data = minnesota.df)
display(county.mixed)
```

```
## lmer(formula = log.radon ~ floor + (1 | county.factor), data
= minnesota.df)
## coef.est coef.se
## (Intercept) 1.43 0.05
## floor -0.74 0.06
##
## Error terms:
## Groups Name Std.Dev.
## county.factor (Intercept) 0.35
## Residual 0.76
## ---
## number of obs: 1187, groups: county.factor, 87
## AIC = 2812.1, DIC = 2787.9
## deviance = 2796.0
```

How to read this: The predicted log radon for a particular county is 1.42, minus 0.72 if you're on the first floor, plus a county term that's drawn from a normal distribution with mean 0 and standard deviation 0.36. The residual standard error of this model is 0.75.

We can think of this model as fitting a regression line for each county, where each regression line has the same slope but a different intercept. For St. Louis County (number 27137), the regression line is:

```
county.lines = as.data.frame(coef(county.mixed)$county.factor)
names(county.lines) = c("Intercept", "Slope")
stlouis.coef = as.numeric(county.lines[row.names(county.lines) == "27137",])
print(stlouis.coef)
```

```
## [1]  0.8830748 -0.7365275
```

For Lac Qui Parle County (number 27073):

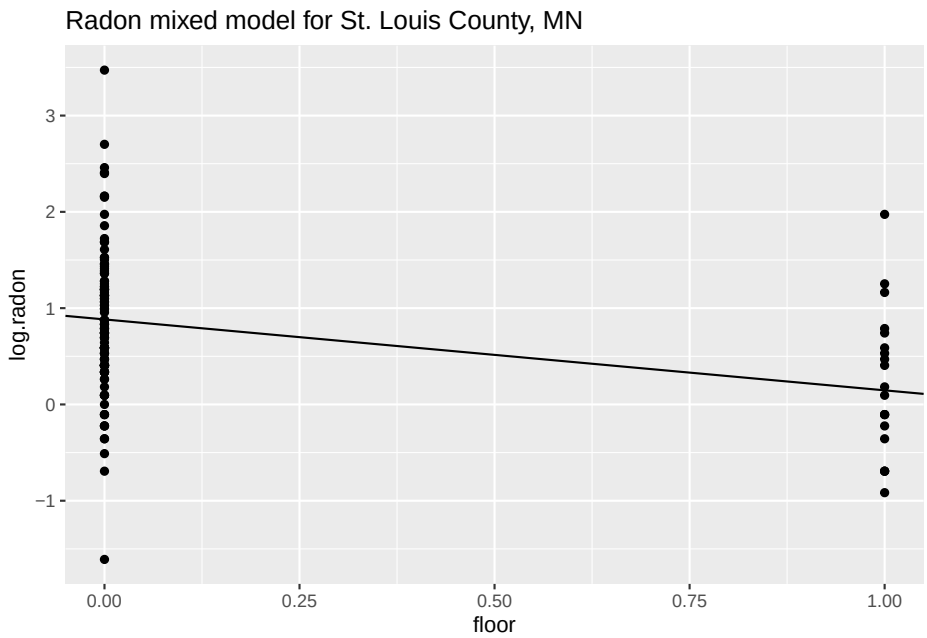
```
lqp.coef = as.numeric(county.lines[row.names(county.lines) == "27073",])
print(lqp.coef)
```

```
## [1]  1.8935175 -0.7365275
```

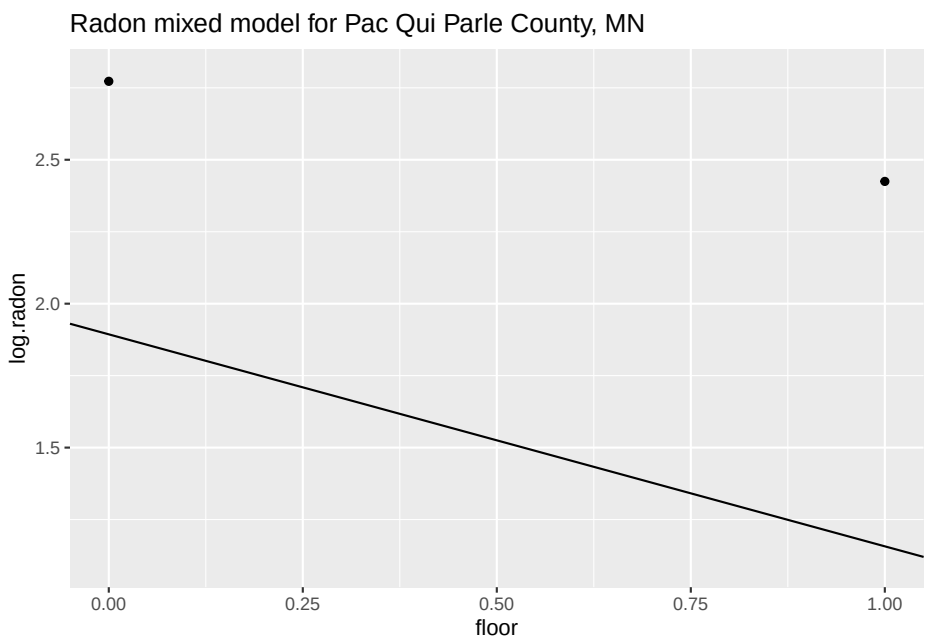
The coefficient for floor is the same but the intercept is different.

Let's see how the mixed model looks for these two counties.

```
ggplot(subset(minnesota.df, county.factor == "27137"), aes(x = floor, y = log.radon)) +
```




```
ggplot(subset(minnesota.df, county.factor == "27073"), aes(x = floor, y = log.radon)) + geom_poir
```

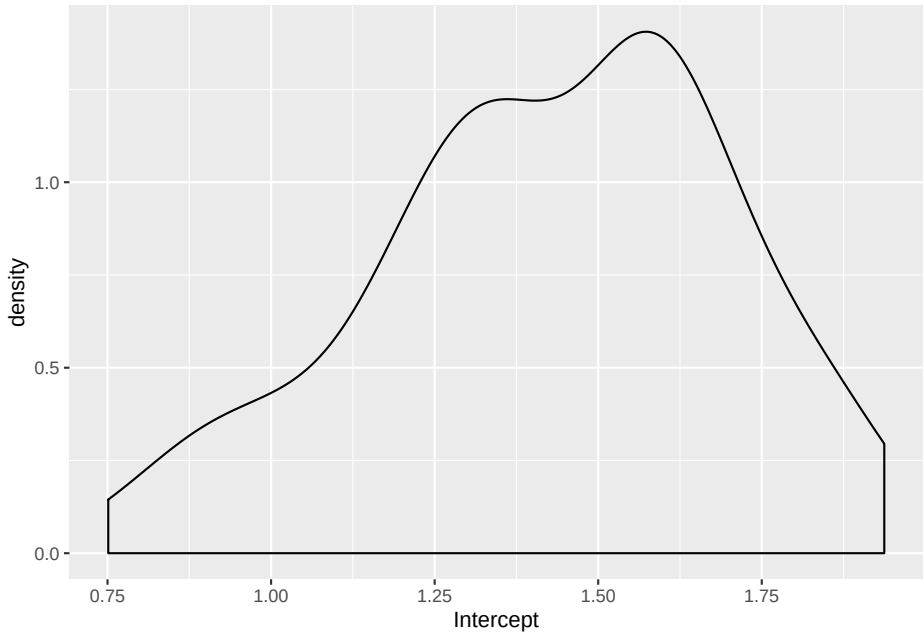


The model is close to the data for St. Louis County but far for Pac Qui Parle. Again, the assumption here is that two data points isn't enough information to make a strong generalization about a county, so we draw a regression line that's

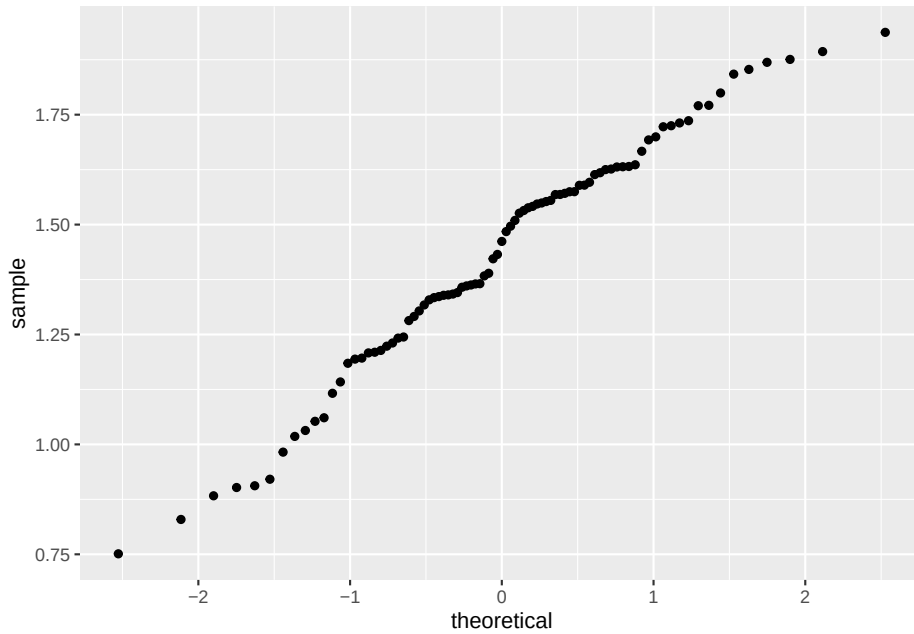
much closer to the data set as a whole.

We can look at the distribution of estimated intercepts:

```
ggplot(county.lines, aes(x = Intercept)) + geom_density()
```




```
ggplot(county.lines, aes(sample = Intercept)) + stat_qq()
```

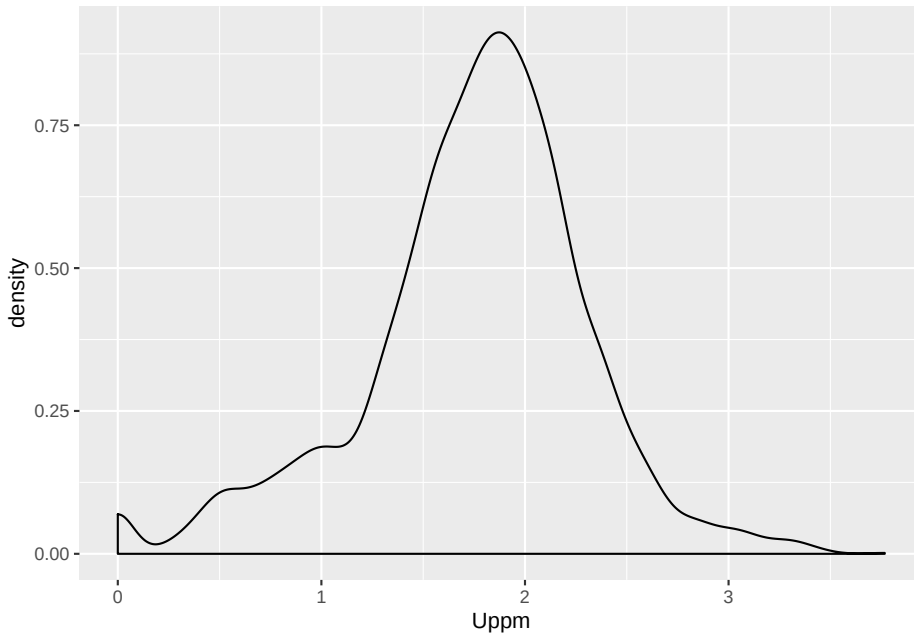


It's not exactly normal, but that isn't a huge deal.

12.3 Uranium, too

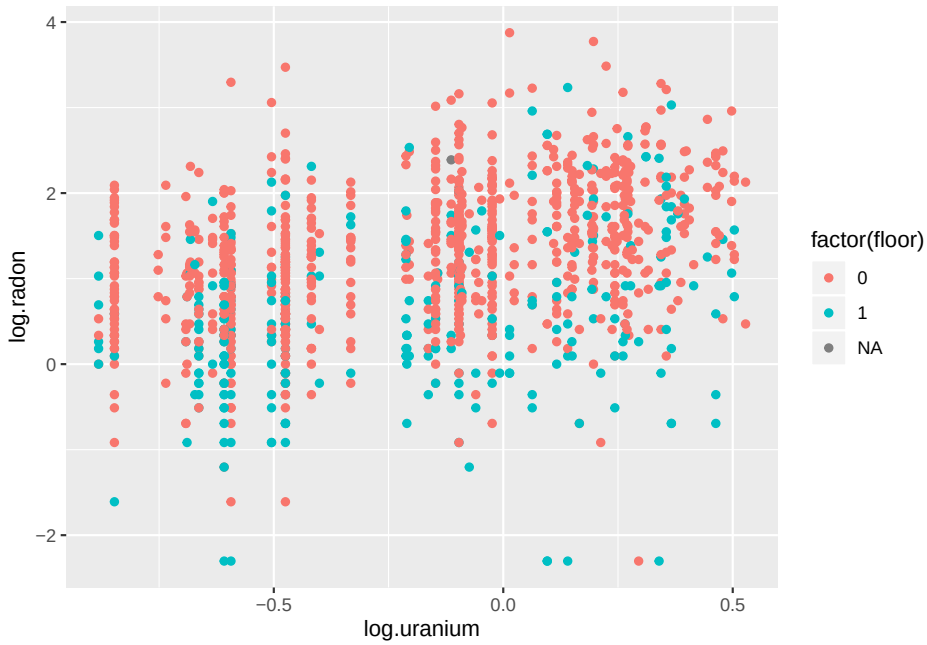
The file `cty.dat` contains data on uranium levels in parts per million in each county (variable `Uppm`), which we can use as an additional covariate.

```
uranium.df = read.table("cty.dat", header=T, sep=",")
ggplot(uranium.df, aes(x = Uppm)) + geom_density()
```



The distribution of uranium looks fine, but since we took the log of radon, it feels like we should log uranium as well. We construct a county number variable and merge the data with our Minnesota data frame:

```
uranium.df$log.uranium = log(uranium.df$Uppm)
uranium.df$county.no = uranium.df$stfips * 1000 + uranium.df$ctfips
minnesota.merged = merge(minnesota.df, uranium.df, by = "county.no")
ggplot(minnesota.merged, aes(x = log.uranium, y = log.radon, color = factor(floor))) +
```



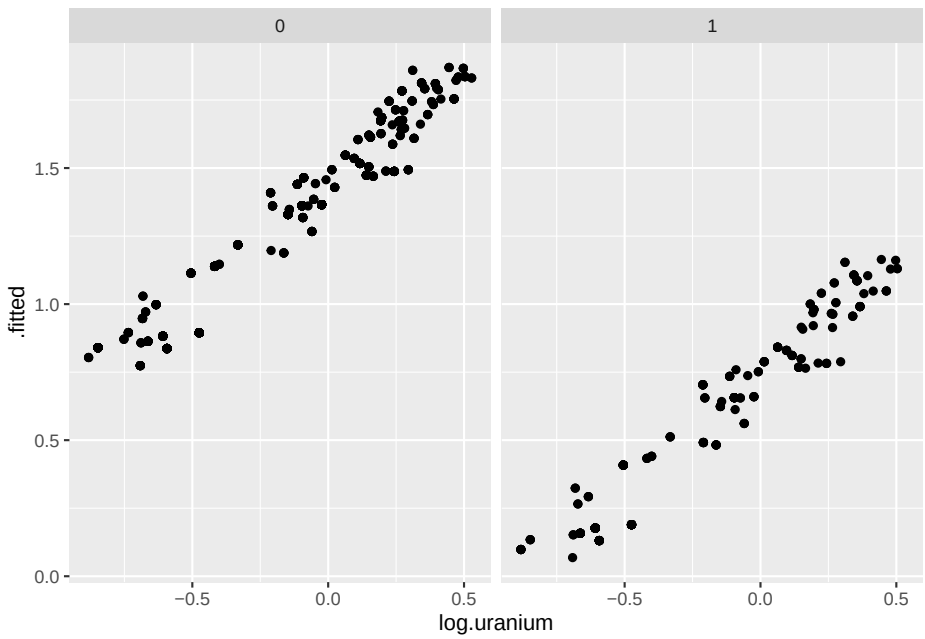
Now add uranium as an explanatory variable:

```
uranium.mixed = lmer(log.radon ~ floor + log.uranium + (1 | county.factor), data = minnesota.merg
display(uranium.mixed)
```

```
## lmer(formula = log.radon ~ floor + log.uranium + (1 |
county.factor),
## data = minnesota.merged)
## coef.est coef.se
## (Intercept) 1.45 0.04
## floor -0.71 0.06
## log.uranium 0.80 0.08
##
## Error terms:
## Groups Name Std.Dev.
## county.factor (Intercept) 0.15
## Residual 0.77
## ---
## number of obs: 1189, groups: county.factor, 85
## AIC = 2789.4, DIC = 2754.9
## deviance = 2767.1
```

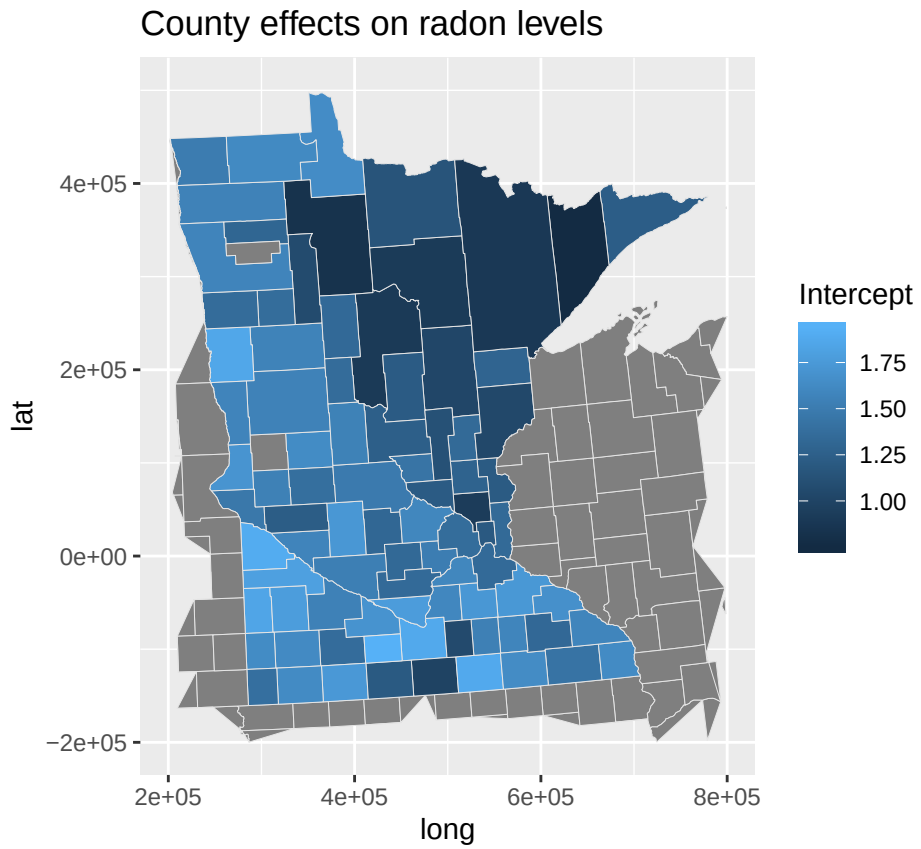
The SD of the county effects is reduced from 0.35 to 0.16 by including uranium as a predictor, which makes sense if uranium is responsible for much but not all of the variation by county. We can also see this from the plot below, which shows that the fitted values are still somewhat scattered about a straight line.

```
library(broom)
uranium.mixed.df = augment(uranium.mixed)
ggplot(uranium.mixed.df, aes(x = log.uranium, y = .fitted)) + geom_point() + facet_gri
```

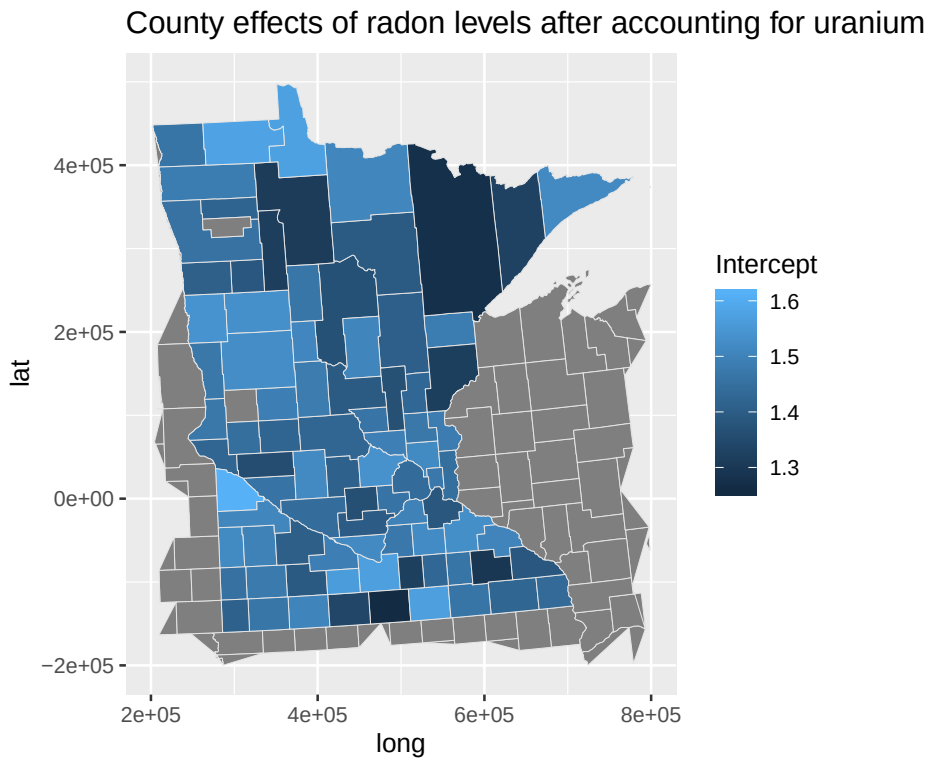


Finally, let's draw maps (using the `socviz` library) to compare the two models.

```
county.lines$id = row.names(county.lines)
# devtools::install_github("kjhealy/socviz")
library(socviz)
county.join = left_join(county_map, county.lines, by = "id")
ggplot(county.join, aes(x = long, y = lat, fill = Intercept, group = group)) + geom_pol
```




```
uranium.lines = as.data.frame(coef(uranium.mixed)$county.factor)
names(uranium.lines) = c("Intercept", "Floor", "Log uranium")
uranium.lines$id = row.names(uranium.lines)
uranium.join = left_join(county_map, uranium.lines, by = "id")
gg = ggplot(uranium.join, aes(x = long, y = lat, fill = Intercept, group = group)) + geom_polygon
gg + ggtitle("County effects of radon levels after accounting for uranium")
```



I could make this look nicer but it's 4:30 am at the time of writing.

Chapter 13

Appendix: Miscellaneous examples and data tricks

Reading: Hadley Wickham, "Tidy Data" (Journal of Statistical Software, 2014):

http://vita.had.co.nz/papers/tidy-data.pdf

Warning: The material here is pretty rough.

13.1 Tuberculosis

Data: https://github.com/hadley/tidy-data/blob/master/data/tb.csv

Let's explore tuberculosis in the U.S. How do male and female rates compare?

```
library(plyr)
tb = read.csv("https://github.com/hadley/tidy-data/raw/master/data/tb.csv", stringsAsFactors = FA
tb.US = subset(tb, iso2 == "US")
# tb.US = as_tibble(tb.US)
```

Gather up the counts into one column:

```
library(dplyr)
library(tidyr)
tb.long = tb.US %>% gather(demographic, count, -iso2, -year, na.rm = TRUE)
```

Separate sex and age using `separate()`:

```
tb.sep = tb.long %>% separate(demographic, c("sex", "age"), sep = 8)
# Get rid of totals
tb.sep = subset(tb.sep, sex != "new_sp")
head(tb.sep)
```

```
##    iso2 year      sex age count
## 56   US 2006 new_sp_m  04     4
## 57   US 2007 new_sp_m  04     4
## 58   US 2008 new_sp_m  04     4
## 85   US 2006 new_sp_m 514     8
## 86   US 2007 new_sp_m 514     8
## 87   US 2008 new_sp_m 514     7
```

```
tail(tb.sep)
```

```
##     iso2 year      sex age count
## 576   US 2004 new_sp_f  65   344
## 577   US 2005 new_sp_f  65   322
## 578   US 2006 new_sp_f  65   303
## 579   US 2007 new_sp_f  65   308
## 580   US 2008 new_sp_f  65   300
## 609   US 2008 new_sp_f   u     0
```

```
# Rename categories
tb.sep$sex[tb.sep$sex == "new_sp_m"] = "male"
tb.sep$sex[tb.sep$sex == "new_sp_f"] = "female"
tb.sep$sex = factor(tb.sep$sex, levels = c("male", "female"))
head(tb.sep)
```

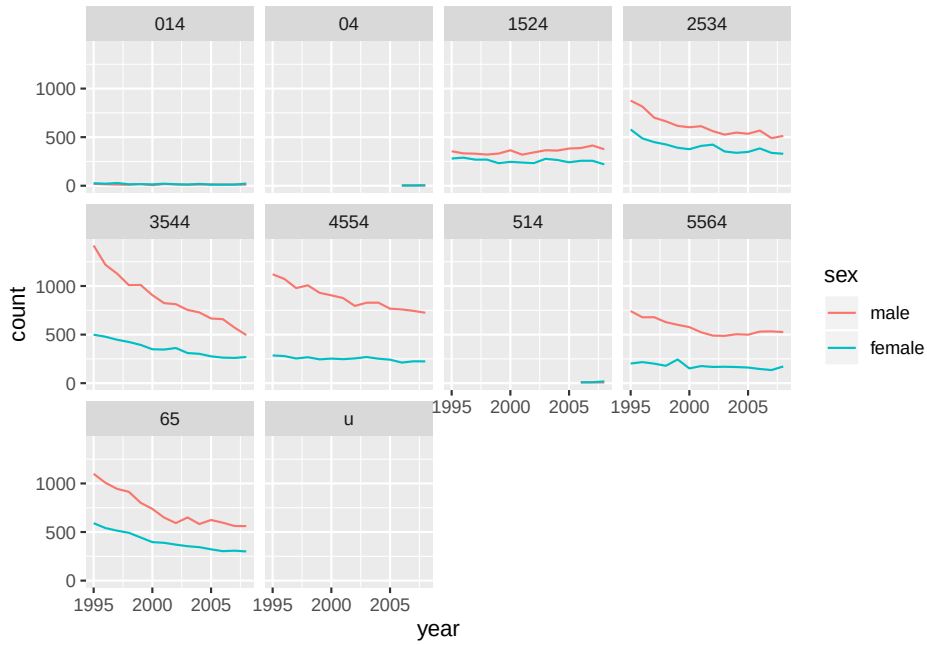
```
##    iso2 year  sex age count
## 56   US 2006 male  04     4
## 57   US 2007 male  04     4
## 58   US 2008 male  04     4
## 85   US 2006 male 514     8
## 86   US 2007 male 514     8
## 87   US 2008 male 514     7
```

```
tail(tb.sep)
```

```
##     iso2 year    sex age count
## 576   US 2004 female  65   344
## 577   US 2005 female  65   322
## 578   US 2006 female  65   303
## 579   US 2007 female  65   308
## 580   US 2008 female  65   300
## 609   US 2008 female   u     0
```

Answer the question:

```
library(ggplot2)
ggplot(tb.sep, aes(x = year, y = count, color = sex)) + geom_line() + facet_wrap(~ age)
```



In all age groups with non-negligible counts, men have higher TB rates than women.

13.2 Baseball

```
# install.packages("Lahman")
library(Lahman)
```

Batting data:

```
# Batting = as_tibble(Batting)
dim(Batting)
```

```
## [1] 105861     22
```

```
names(Batting)
```

```
## [1] "playerID" "yearID" "stint" "teamID" "lgID" "G"
## [7] "AB" "R" "H" "X2B" "X3B" "HR"
## [13] "RBI" "SB" "CS" "BB" "SO" "IBB"
## [19] "HBP" "SH" "SF" "GIDP"
```

Our goal is to predict a player's 2015 statistics based on 2012 to 2014 statistics. Get 2012–2015 data:

```
pred.year = 2015
B = subset(Batting, yearID >= pred.year - 3 & yearID <= pred.year)
```

Add a variable for "plate appearances":

```
B = transform(B, PA = AB + BB + HBP + SF + SH)
```

Aggregate by year. We'll use `ddply()` from the `plyr` library (there are also functions to do this in `dplyr` but I haven't learned them.)

```
stats = c("PA", "AB", "R", "H", "X2B", "X3B", "HR", "RBI", "SB", "CS", "BB", "SO", "IBI
B = ddply(B[, c("playerID", "yearID", stats)], ~playerID + yearID, summarise, PA = sum
dim(B)
```

```
## [1] 5256   19
```

```
summary(B)
```

```
## playerID yearID PA AB
## Length:5256 Min.  :2012 Min.  : 0.0 Min.  : 0
## Class :character 1st Qu.:2013 1st Qu.: 0.0 1st Qu.: 0
## Mode :character Median :2014 Median : 20.0 Median : 18
## Mean :2014 Mean :140.1 Mean :126
## 3rd Qu.:2015 3rd Qu.:225.0 3rd Qu.:204
## Max.  :2015 Max.  :740.0 Max.  :684
## R H X2B X3B
## Min.  : 0.00 Min.  : 0.00 Min.  : 0.000 Min.  : 0.0000
## 1st Qu.: 0.00 1st Qu.: 0.00 1st Qu.: 0.000 1st Qu.: 0.0000
## Median : 1.00 Median : 3.00 Median : 0.000 Median : 0.0000
## Mean : 15.54 Mean : 31.94 Mean : 6.252 Mean : 0.6634
## 3rd Qu.: 22.00 3rd Qu.: 48.00 3rd Qu.: 9.000 3rd Qu.: 0.0000
## Max.  :129.00 Max.  :225.00 Max.  :55.000 Max.  :15.0000
## HR RBI SB CS
## Min.  : 0.000 Min.  : 0.00 Min.  : 0.000 Min.  : 0.0000
## 1st Qu.: 0.000 1st Qu.: 0.00 1st Qu.: 0.000 1st Qu.: 0.0000
## Median : 0.000 Median : 1.00 Median : 0.000 Median : 0.0000
## Mean : 3.556 Mean : 14.78 Mean : 2.129 Mean : 0.8071
## 3rd Qu.: 4.000 3rd Qu.: 20.00 3rd Qu.: 1.000 3rd Qu.: 1.0000
## Max.  :53.000 Max.  :139.00 Max.  :64.000 Max.  :23.0000
## BB SO IBB HBP
## Min.  : 0.00 Min.  : 0.00 Min.  : 0.0000 Min.  : 0.000
## 1st Qu.: 0.00 1st Qu.: 0.00 1st Qu.: 0.0000 1st Qu.: 0.000
## Median : 1.00 Median : 6.00 Median : 0.0000 Median : 0.000
## Mean : 10.93 Mean : 28.16 Mean : 0.7627 Mean : 1.196
## 3rd Qu.: 15.00 3rd Qu.: 45.00 3rd Qu.: 0.0000 3rd Qu.: 1.000
## Max.  :143.00 Max.  :222.00 Max.  :29.0000 Max.  :30.000
## SH SF GIDP
## Min.  : 0.000 Min.  : 0.000 Min.  : 0.000
```

```
## 1st Qu.: 0.000 1st Qu.: 0.000 1st Qu.: 0.000
## Median : 0.000 Median : 0.000 Median : 0.000
## Mean : 1.028 Mean : 0.942 Mean : 2.796
## 3rd Qu.: 1.000 3rd Qu.: 1.000 3rd Qu.: 4.000
## Max.  :17.000 Max.  :12.000 Max.  :31.000
```

Pick out years:

```
B.short = B[,c(1,2,3,4,5,6,9,10,13,14)]
names(B.short)
```

```
## [1] "playerID" "yearID" "PA" "AB" "R" "H"
## [7] "HR" "RBI" "BB" "SO"
```

```
B.2012 = B.short[B.short$yearID==2012,]
B.2013 = B.short[B.short$yearID==2013,]
B.2014 = B.short[B.short$yearID==2014,]
B.2015 = B.short[B.short$yearID==2015,]
```

Add names to things:

```
names(B.2012)=c("playerID","yearID.1","PA.1","AB.1","R.1","H.1","HR.1","RBI.1","BB.1","SO.1")
names(B.2013)=c("playerID","yearID.2","PA.2","AB.2","R.2","H.2","HR.2","RBI.2","BB.2","SO.2")
names(B.2014)=c("playerID","yearID.3","PA.3","AB.3","R.3","H.3","HR.3","RBI.3","BB.3","SO.3")
names(B.2015)=c("playerID","yearID.4","PA.4","AB.4","R.4","H.4","HR.4","RBI.4","BB.4","SO.4")
```

13.2.1 Merging things

For easy problems `cbind()` and `rbind()` may suffice, but this is harder. Instead, we'll `merge()` everything:

```
B2 = merge(B.2012, B.2013, by = "playerID")
B3 = merge(B2, B.2014, by = "playerID")
B4 = merge(B3, B.2015, by = "playerID")
```

Only keep the complete cases:

```
B.complete = B4[complete.cases(B4),]
PAs = (B.complete$PA.1>0) * (B.complete$PA.2>0) * (B.complete$PA.3>0) * (B.complete$PA.4>0)
B.all = B.complete[PAs == 1,]
```

What if we wanted to include birth year? (We don't need it here, but age would be the next variable that I'd try to include in a model.) Let's add that in from the `Master` data frame:

```
birth = Master[,c("playerID", "birthYear")]
B.merged = merge(B.all, birth, by="playerID")
```

13.2.2 Making a model

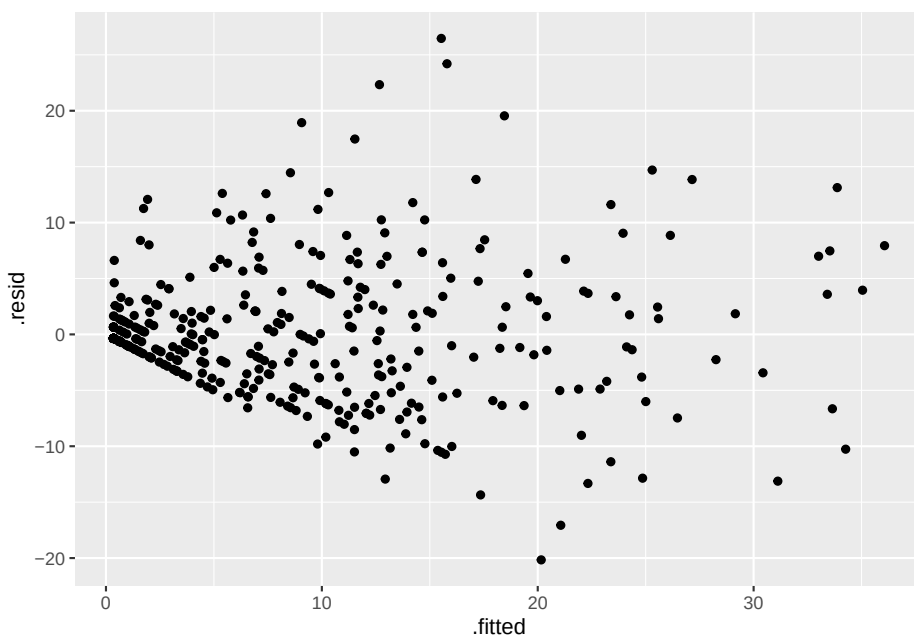
Let's predict home runs based on their past three years of home runs:

```
homerun.lm = lm(HR.4 ~ HR.1 + HR.2 + HR.3, data=B.merged)
library(broom)
tidy(homerun.lm)
```

```
## # A tibble: 4 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   0.349     0.368      0.948 3.43e- 1
## 2 HR.1          0.0358    0.0428     0.838 4.03e- 1
## 3 HR.2          0.273     0.0527     5.18  3.30e- 7
## 4 HR.3          0.687     0.0490    14.0   3.75e-37
```

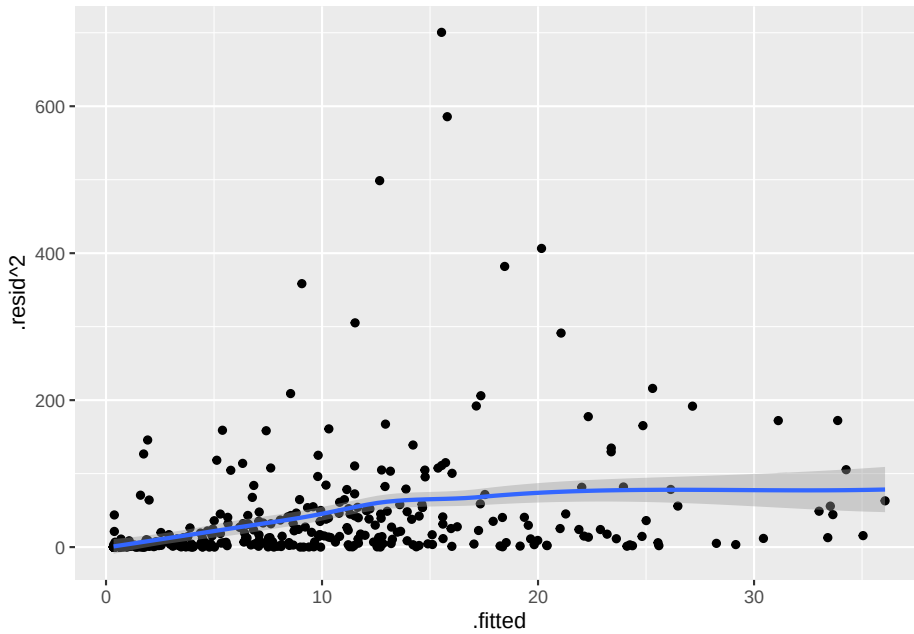
The coefficients get smaller for older years, which makes sense. Look at the residuals:

```
homerun.df = augment(homerun.lm)
ggplot(homerun.df, aes(x = .fitted, y = .resid)) + geom_point()
```



It's heteroskedastic. How do the squared residuals change?

```
homerun.df = augment(homerun.lm)
gg = ggplot(homerun.df, aes(x = .fitted, y = .resid^2)) + geom_point()
gg + geom_smooth(method.args = list(degree = 1))
```

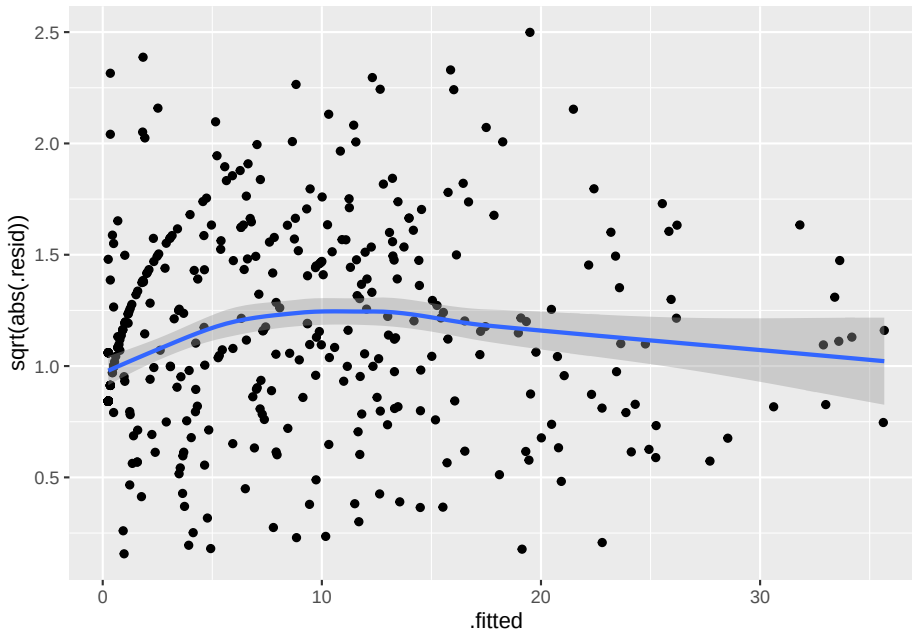


An overdispersed Poisson might work okay. It doesn't make much sense to predict on the log scale, so we fit on the original scale instead.

```
homerun.quasi = glm(HR.4 ~ HR.1 + HR.2 + HR.3, start = coef(homerun.lm),
  family = quasipoisson(link = "identity"), data = B.merged)
display(homerun.quasi)
```

```
## glm(formula = HR.4 ~ HR.1 + HR.2 + HR.3, family =
quasipoisson(link = "identity"),
## data = B.merged, start = coef(homerun.lm))
## coef.est coef.se
## (Intercept) 0.25 0.09
## HR.1 0.10 0.05
## HR.2 0.25 0.06
## HR.3 0.66 0.06
## ---
## n = 444, k = 4
## residual deviance = 1489.7, null deviance = 5056.3
(difference = 3566.6)
## overdispersion parameter = 3.9
```

```
homerun.quasi.df = augment(homerun.quasi)
ggplot(homerun.quasi.df, aes(x = .fitted, y = sqrt(abs(.resid)))) + geom_point() + geom_smooth(me
```



The coefficients are a bit different and the standard errors are a bit bigger. The deviance residuals aren't perfect but they could be worse.

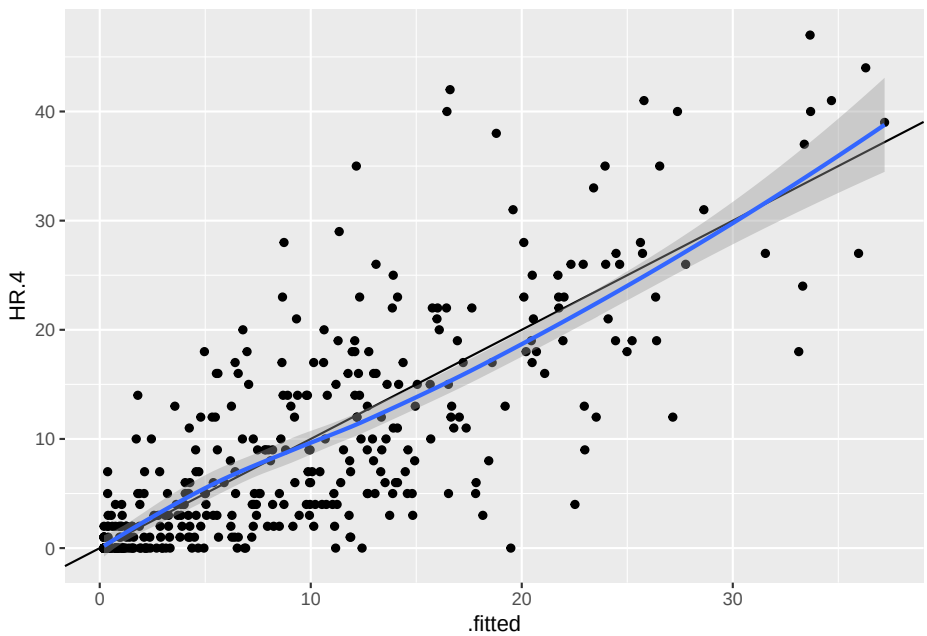
Alternatively, try a negative binomial, which can give probabilistic predictions:

```
library(MASS)
homerun.nb = glm.nb(HR.4 ~ HR.1 + HR.2 + HR.3, data = B.merged, start = coef(homerun.lm
summary(homerun.nb)
```

```
##
## Call:
## glm.nb(formula = HR.4 ~ HR.1 + HR.2 + HR.3, data = B.merged,
## start = coef(homerun.lm), link = identity, init.theta =
2.311204214)
##
## Deviance Residuals:
## Min 1Q Median 3Q Max
## -3.2205 -0.8580 -0.5191 0.2785 4.2226
##
## Coefficients:
## Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.20184 0.04744 4.255 2.1e-05 ***
## HR.1 0.17298 0.04796 3.607 0.000310 ***
## HR.2 0.19714 0.05635 3.499 0.000468 ***
## HR.3 0.66562 0.06987 9.526 < 2e-16 ***
## ---
```

```
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Negative Binomial(2.3112) family
taken to be 1)
##
## Null deviance: 1505.06 on 443 degrees of freedom
## Residual deviance: 487.61 on 440 degrees of freedom
## AIC: 2154.4
##
## Number of Fisher Scoring iterations: 1
##
##
## Theta: 2.311
## Std. Err.: 0.270
##
## 2 x log-likelihood: -2144.445
```

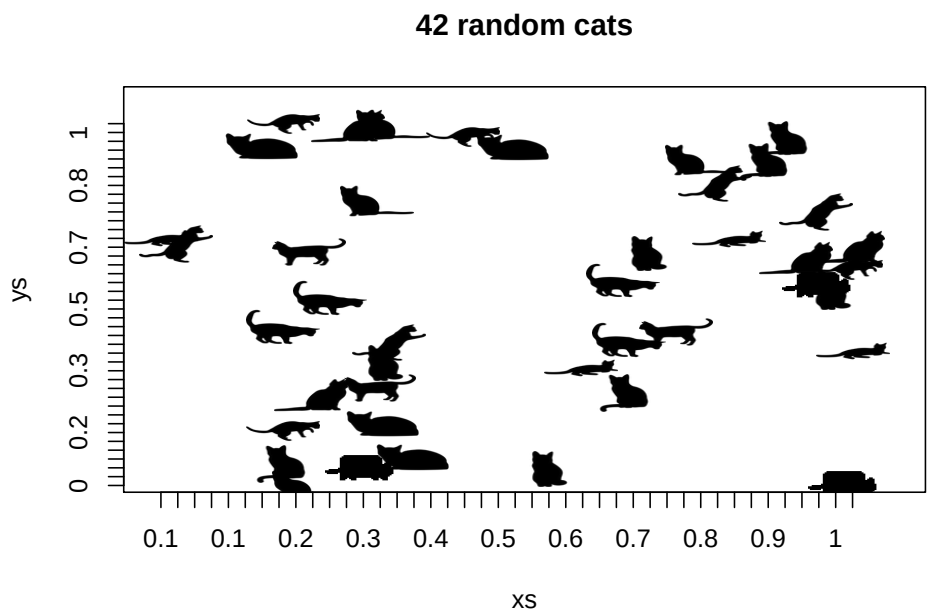
```
homerun.nb.df = augment(homerun.nb, type.residuals = "response")
ggplot(homerun.nb.df, aes(x = .fitted, y = HR.4)) + geom_point() + geom_abline() + geom_smooth()
```



13.3 Catterplots

These are self-explanatory I think.

```r
# library(devtools)
# install_github("Gibbsdavidl/CatterPlots")
library(CatterPlots)
cat.random = multicat(xs=runif(42), ys=runif(42), cat = 1:11, main = "42 random cats")
```



13.4 Things I need to improve in this book

- **Bigger data sets.** Get rid of most of the $n < 100$ examples, since it's really hard to tell if interactions are real in data sets of that size
- A clearer explication of the philosophy: how is Explanatory Data Analysis different from other stuff?
- Interpreting linear/logistic regression coefficients
- Stop relying on lattice.RData examples
- Be consistent about breaking ggplot calls into multiple lines
- Be consistent about colorblind-friendliness (viridisLite?)
- Eradicate the remaining P-values (mostly in the Hi-Di section)
- Random effects section is kind of useless
- Discuss of weights is not even slightly rigorous (this is okay but should point to more rigorous stuff)
- More messy data to clean up (maybe add some black box imputation)