MLE - Lab 12

Andy Ballard

April 7, 2017

Today

- Hierarchical models on hierarchical models on hierarchical models (nonlinear outcomes edition)
- A tutorial in getting information from the web

Mixed effects logit models

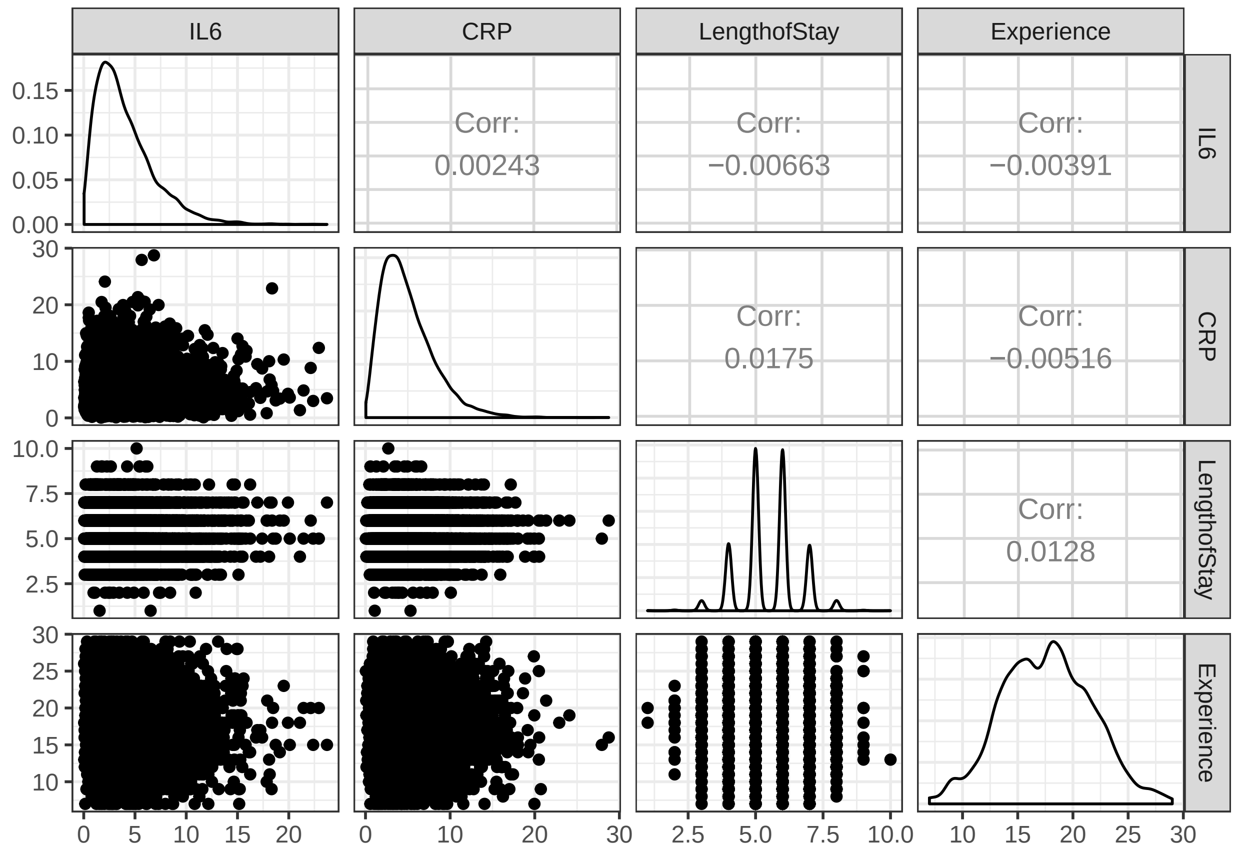
We'll muddle around in the real world for a bit before going off into web data land. We'll do an example of a mixed effects logistic regression model with a sort of data we don't often see. To run these models, you go through nearly the same process as we've been doing, but

A large HMO wants to know what patient and physician factors are most related to whether a patient's lung cancer goes into remission after treatment as part of a larger study of treatment outcomes and quality of life in patients with lunger cancer.

```
hdp <- read.csv(paste0(labPath, "hdp.csv"))
hdp <- within(hdp, {
  Married <- factor(Married, levels = 0:1, labels = c("no", "yes"))
  DID <- factor(DID)
  HID <- factor(HID)
})
```

This dataset has a number of variables about specific patients, who are nested within doctors (doctor ID = `DID`), who are nested within hospitals (hospital ID = `HID`.

```
ggpairs(hdp[, c("IL6", "CRP", "LengthofStay", "Experience")])
```

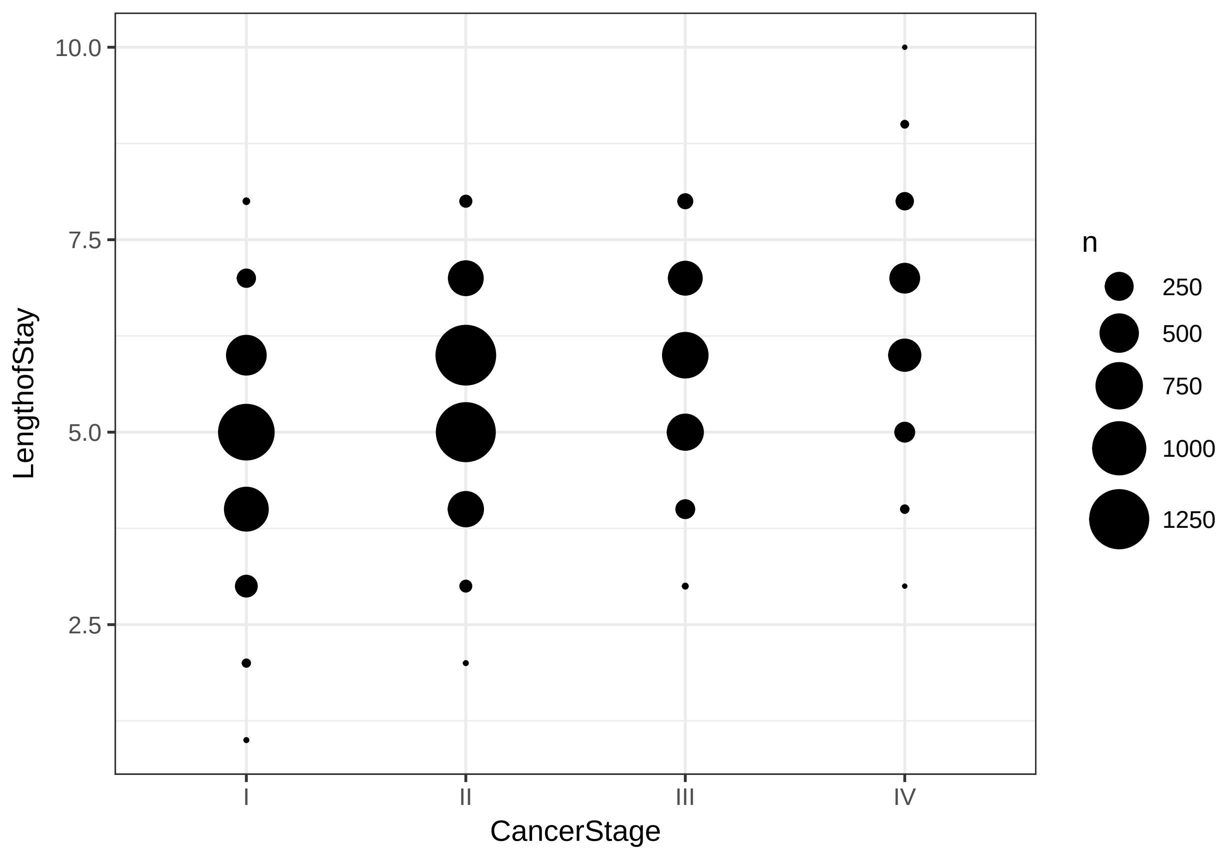


`GGally` is a cool package with some fancy implementations of `ggplot2` plotting techniques. We've plotted the continuous variables we'll use in our prediction, `IL6`, `CRP`, `LengthofStay` and `Experience` against eachother.

The good news is that our predictor variables don't seem to be highly correlated with one another, so we don't have to worry about multicollinearity.

There are some other neat visualizations we can do to look at the relationships between our variables. For instance, what if we want to know how length of stay is related to the stage of cancer?

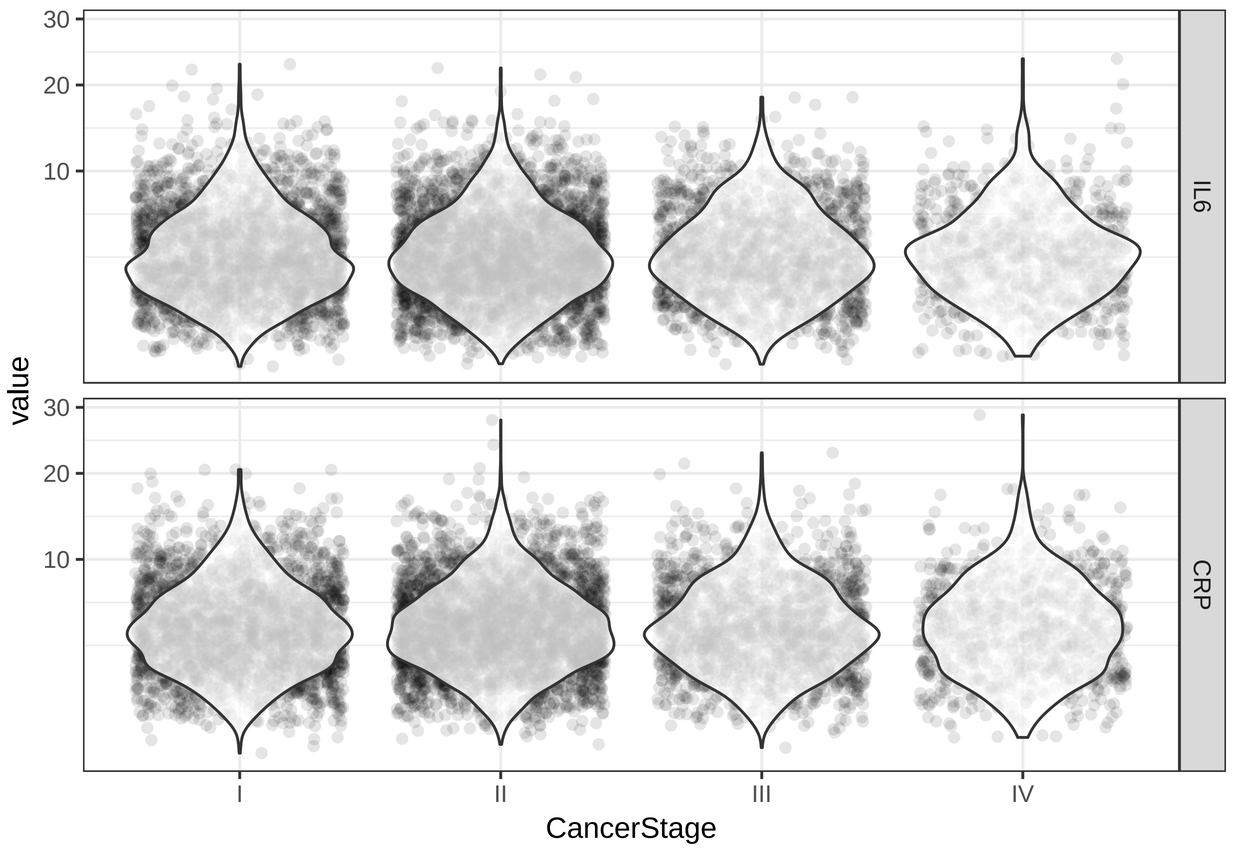
```
ggplot(hdp, aes(x = CancerStage, y = LengthofStay)) +
  stat_sum(aes(size = ..n.., group = 1)) +
  scale_size_area(max_size=10)
```



`stat_sum` creates the dot plot we see, and `scale_size_area` contols the size of the dots, which are proportional to the number of observations at each of the values. Perhaps unsurprisingly, the length of stay seems to increase slightly with the stage of cancer.

Because `IL6` and `CRP` are much more skewed, it may be a good idea to use violin plots to showcase their relationship to cancer stage (and it's also a good opportunity for me to show you violin plots). Violin plots are density plots reflected over the y-axis.

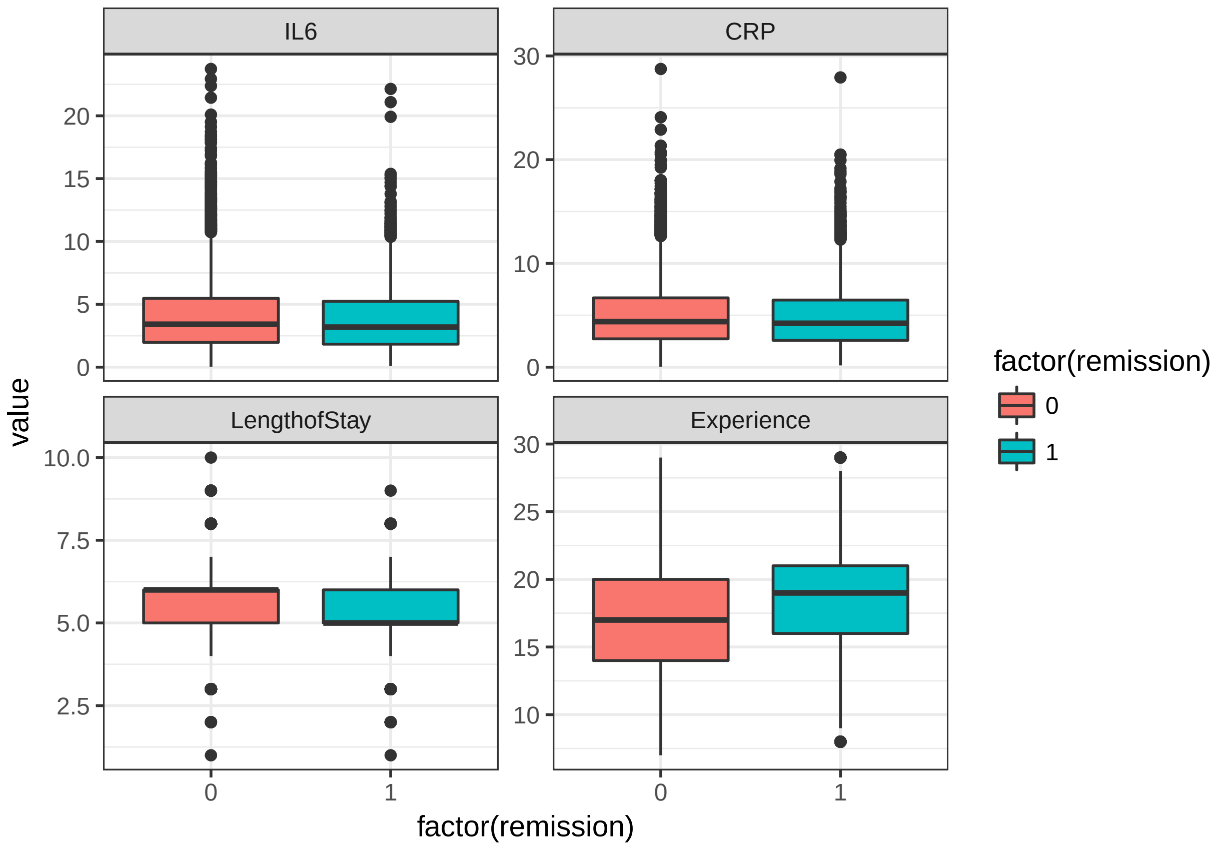
```
tmp <- melt(hdp[, c("CancerStage", "IL6", "CRP")], id.vars="CancerStage")
ggplot(tmp, aes(x = CancerStage, y = value)) +
  geom_jitter(alpha = .1) +
  geom_violin(alpha = .75) +
  facet_grid(variable ~ .) +
  scale_y_sqrt()
```



We also plot the actual data points with `geom_jitter` to show that there isn't much of a relationship with either `IL6` or `CRP` and cancer stage.

We can also flip it around and look to see how each of our continuous predictors differs over a binary variable. In this case, we'll look at box plots of each variable split by whether the patient goes into remission after treatment, which is our DV.

```
tmp <- melt(hdp[, c("remission", "IL6", "CRP", "LengthofStay", "Experience")],
  id.vars="remission")
ggplot(tmp, aes(factor(remission), y = value, fill=factor(remission))) +
  geom_boxplot() +
  facet_wrap(~variable, scales="free_y")
```



Now that we've looked at the data pretty thoroughly, we'll run a model. We will use a mixed effects logistic regression to predict whether a patient goes into remission after treatment based on 4 patient-level predictors (`IL6`, `CRP`, `LengthofStay`, `CancerStage`), 1 doctor-level predictor (`Experience`), and we'll allow the intercepts to vary over doctors via the doctor ID variable `DID`.

```
# Estimate the model
m <- glmer(remission ~ IL6 + CRP + CancerStage + LengthofStay + Experience +
    (1 | DID), data = hdp, family = binomial, control = glmerControl(optimizer = "bobyqa"),
    nAGQ = 10)

# Print the model results
print(m, corr = FALSE)
```

```
## Generalized linear mixed model fit by maximum likelihood (Adaptive
##   Gauss-Hermite Quadrature, nAGQ = 10) [glmerMod]
##  Family: binomial  ( logit )
## Formula:
## remission ~ IL6 + CRP + CancerStage + LengthofStay + Experience +
##     (1 | DID)
##    Data: hdp
##       AIC       BIC    logLik  deviance  df.resid
##  7397.276  7460.733 -3689.638  7379.276      8516
## Random effects:
##  Groups Name        Std.Dev.
##  DID    (Intercept) 2.015
## Number of obs: 8525, groups:  DID, 407
## Fixed Effects:
```

```
##    (Intercept)             IL6             CRP   CancerStageII
##       -2.05272        -0.05677        -0.02148        -0.41394
## CancerStageIII   CancerStageIV    LengthofStay      Experience
##       -1.00347        -2.33704        -0.12118         0.12009
```

Whoa, that took a long time. The reason is the `nAGQ=10` argument. The default is 1, which defines the optimization as using Laplace approximation, which Daniel mentioned briefly. It's fast, but inaccurate. Larger values produce greater accuracy in the evaluation of the log-likelihood at the expanse of speed. the `glmerControl(optimizer="bobyqa")` is actually the default optimizer, I just wanted to draw your attention to the fact that there are multiple types of optimizers that you can use for these types of problems. They may produce slightly different results, but should generally be close.

We can get confidence intervals for the fixed effects pretty easily.

```
se <- sqrt(diag(vcov(m)))
# Table of estimates with 95% CI
(tab <- cbind(Est = fixef(m), Lo = fixef(m) - 1.96 * se, Hi = fixef(m) + 1.96 *
    se))
```

```
##                        Est          Lo           Hi
## (Intercept)    -2.05272275 -3.09424525 -1.011200250
## IL6            -0.05677183 -0.07934783 -0.034195834
## CRP            -0.02148290 -0.04151094 -0.001454863
## CancerStageII  -0.41393507 -0.56243211 -0.265438030
## CancerStageIII -1.00346728 -1.19610141 -0.810833140
## CancerStageIV  -2.33704008 -2.64683307 -2.027247082
## LengthofStay   -0.12118144 -0.18710207 -0.055260814
## Experience      0.12008971  0.06628918  0.173890233
```

And if we want these to be odds ratios rather than on the logit scale, we can exponentiate the estimates and CIs.

```
exp(tab)
```

```
##                       Est         Lo        Hi
## (Intercept)    0.12838487 0.04530920 0.3637821
## IL6            0.94480962 0.92371857 0.9663822
## CRP            0.97874621 0.95933884 0.9985462
## CancerStageII  0.66104387 0.56982151 0.7668700
## CancerStageIII 0.36660611 0.30237074 0.4444876
## CancerStageIV  0.09661318 0.07087532 0.1316976
## LengthofStay   0.88587321 0.82935908 0.9462383
## Experience     1.12759800 1.06853567 1.1899249
```

What sticks out at you so far? Doctor experience is the only variable we've included that has a positive impact on patients' cancer going into remission.

We can visualize our results with some predicted probabilities to drive the point home. We'll look at the probability of going into remission after treatment varying the length of stay.

```
# Subset data
tmpdat <- hdp[, c("IL6", "CRP", "CancerStage", "LengthofStay", "Experience",
    "DID")]
# Look at the variable we'll vary
summary(hdp$LengthofStay)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   5.000   5.000   5.492   6.000  10.000
```

```r
# Make range for predictions
scenRange <- with(hdp, seq(from = min(LengthofStay), to = max(LengthofStay), length.out = 100))

# Calculate predicted probabilities
preds <- lapply(scenRange, function(j) {
    tmpdat$LengthofStay <- j
    predict(m, newdata = tmpdat, type = "response")
})
```

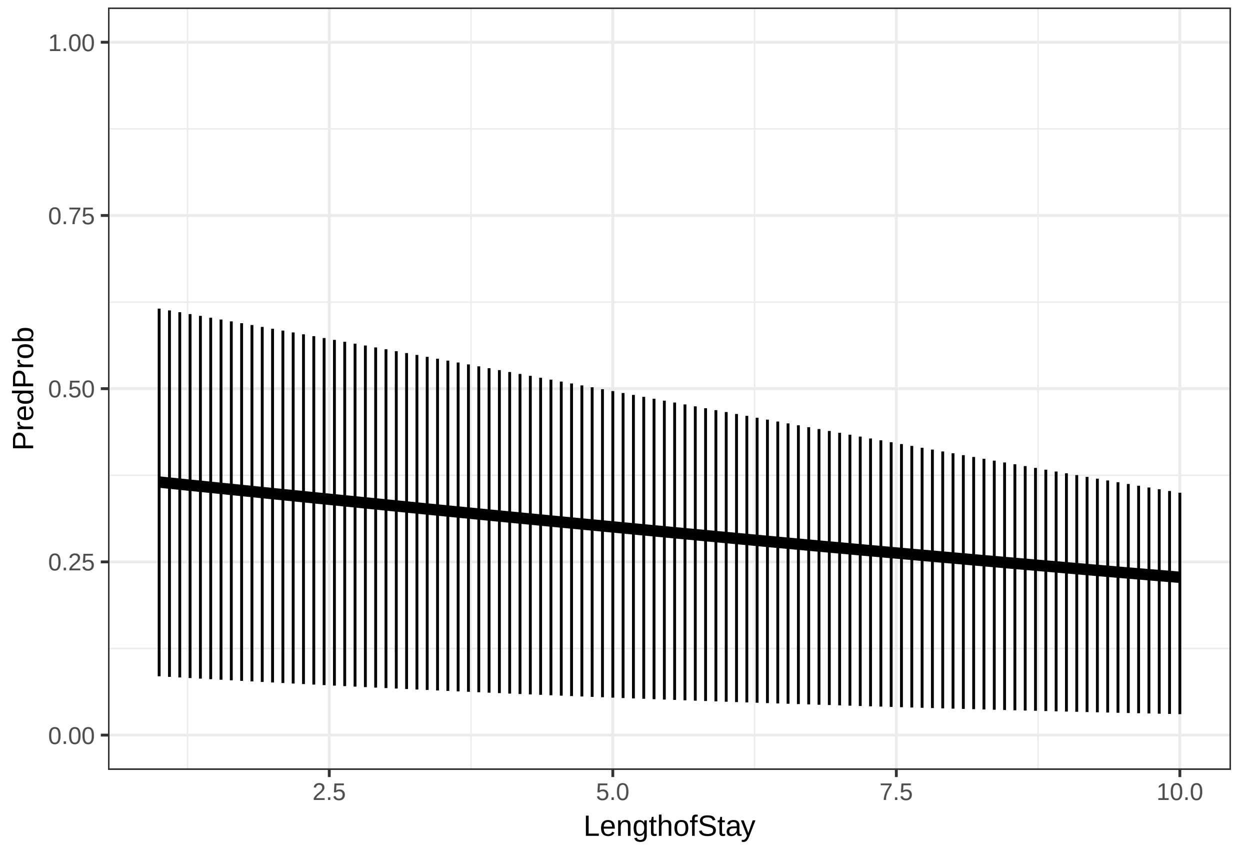
This is doing the predicted probabilities a bit differently than we have been. How does it work?

Now that we have our predicted probabilities, we can graph them. We can also add uncertainty.

```r
# Add uncertainty
plotdat <- t(sapply(preds, function(x) {
    c(M = mean(x), quantile(x, c(0.25, 0.75)))
}))

# Add in LengthofStay values and convert to data frame
plotdat <- as.data.frame(cbind(plotdat, scenRange))

# Better names and show the first few rows
colnames(plotdat) <- c("PredProb", "Lower", "Upper", "LengthofStay")
head(plotdat)
```

```
##    PredProb      Lower     Upper LengthofStay
## 1 0.3652314 0.08489840 0.6155633     1.000000
## 2 0.3637051 0.08404643 0.6129530     1.090909
## 3 0.3621811 0.08320223 0.6103362     1.181818
## 4 0.3606593 0.08236574 0.6077130     1.272727
## 5 0.3591398 0.08153692 0.6050836     1.363636
## 6 0.3576226 0.08071570 0.6024481     1.454545
```

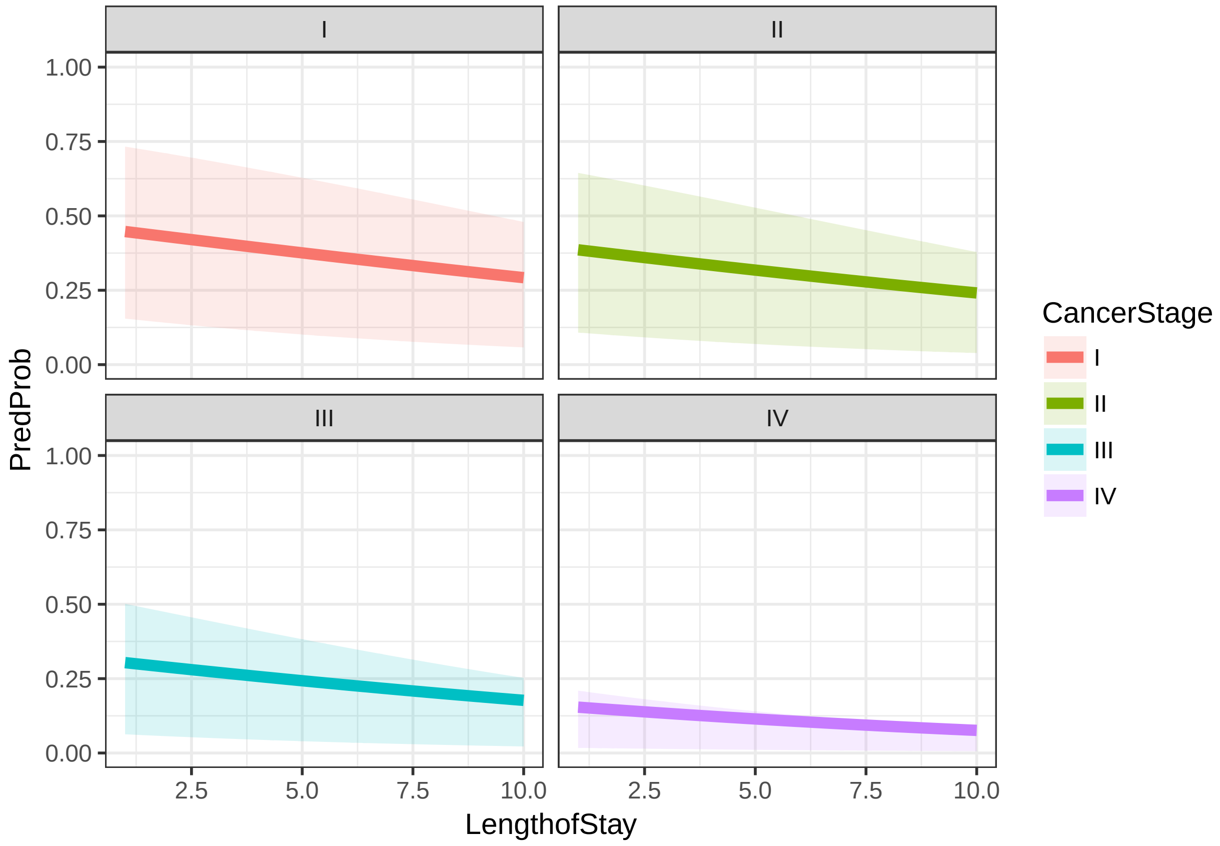
```r
# Plot
ggplot(plotdat, aes(x = LengthofStay, y = PredProb)) + geom_linerange(aes(ymin = Lower, ymax = Upper))
```



The `geom_linerange` command is just a different way of looking at uncertainty. As we can see, the intervals are pretty wide and the slope of the average marginal predicted probabilities is not very steep, so the relationship here is tenuous. However, it may be true that including the cancer stage in our predicted probabilities will change things.

```r
# Calculate predicted probabilities
biprobs <- lapply(levels(hdp$CancerStage), function(stage) {
  tmpdat$CancerStage[] <- stage
  lapply(scenRange, function(j) {
    tmpdat$LengthofStay <- j
    predict(m, newdata = tmpdat, type = "response")
  })
})

# Get means and quartiles for all scenRange values for each level of CancerStage
plotdat2 <- lapply(biprobs, function(X) {
  temp <- t(sapply(X, function(x) {
    c(M=mean(x), quantile(x, c(.25, .75)))
  }))
  temp <- as.data.frame(cbind(temp, scenRange))
  colnames(temp) <- c("PredProb", "Lower", "Upper", "LengthofStay")
  return(temp)
})

# Collapse to one data frame
plotdat2 <- do.call(rbind, plotdat2)
```

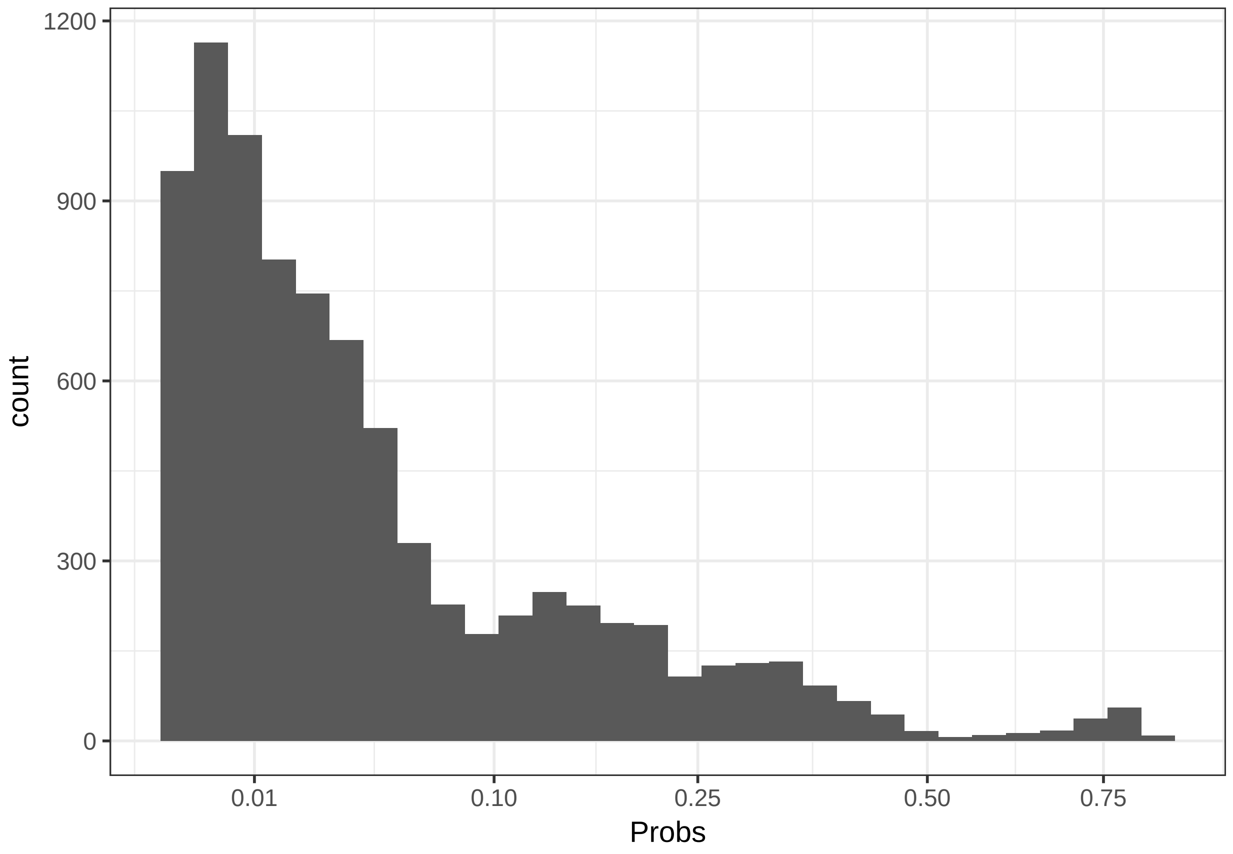
```r
# Add cancer stage
plotdat2$CancerStage <- factor(rep(levels(hdp$CancerStage), each = length(scenRange)))

# Plot
ggplot(plotdat2, aes(x = LengthofStay, y = PredProb)) +
  geom_ribbon(aes(ymin = Lower, ymax = Upper, fill = CancerStage), alpha = .15) +
  geom_line(aes(colour = CancerStage), size = 2) +
  ylim(c(0, 1)) + facet_wrap(~  CancerStage)
```



Things don't look good for those with stage 4 cancer compared to stage 1, 2, or 3, as a function of length of stay. We could even do a more fine-grained analysis and just look at the predicted probabilities of going into remission for a stage 4 cancer patient who has spent at least 10 days in the hospital. A simple histogram will do us wonders here.

```r
ggplot(data.frame(Probs = biprobs[[4]][[100]]), aes(Probs)) + geom_histogram() +
    scale_x_sqrt(breaks = c(0.01, 0.1, 0.25, 0.5, 0.75))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The bottom line is that using nonlinear outcomes in multilevel modeling is quite similar to using continuous outcomes. You've got these tools in your toolbelt.

Getting information from the web

Today, we'll get you a crash course in the basics of getting data off the web. There are two main ways to do this:

1) Interact with the Application Programming Interface (API) for a website and/or company, or
2) Point a programming language at a web address, and use a parser to extract information from it (this is what people usually refer to as scraping)

We will do an example with each. We'll get data on movies from IMDB direct from the website, and we'll use Twitter's API to get some tweets.

Scraping IMDB

What are you actually doing when you scrape data directly from a webpage? Generally, you download all the information about a webpage (usually in HTML), and then extract the relevant information. In `R`, this requires a special package for parsing HTML from websites. Hadley Wickham has a package for this called `rvest` that's inspired by the `beautifulsoup` package in Python. Python is more widely used for web scraping, but the resources in `R` have been catching up over the years, to the point where the biggest limitation is the fact that `R` is often slower than traditional programming languages. For our purposes, though, `R` is just fine.

What we'll do is use the `rvest` (get it, like harvest?) package to get data about movies from IMDB. Let's look at The Lego Movie.

First, we'll use the `read_html()` function to download and parse the information on the IMDB page for The Lego Movie:

```
lego_movie <- read_html("http://www.imdb.com/title/tt1490017/")
```

Now let's say we want to know what the IDMB rating (out of 10) is for the movie. If we actually go to the website, this is trivial. But what if we wanted to get the score for hundreds of movies? We need to be able to find the rating in the `lego_movie` object. To do this, we need to know what to look for. Specific objects displayed on a webpage are called elements, and each specific element can be found in four general ways: 1) element HTML ID, 2) element class name, 3) xpath selectors, and 4) CSS selectors. There is a great chrome extension called SelectorGadget that will let us easily pick out CSS selectors for various elements. For more information, check out this vignette: `vignette("selectorgadget")`.

I've already figured out that the CSS selector we want is 'strong span'. Now we can pull out that element via the `html_node` function, which extracts objects from parsed HTML that match a given selector. However, even then we still need to do more, since the `html_node` function will pull out a list, and we just want a numeric value for the rating. To pull out the text of the object, we use the `html_text` function. Then we make it a numeric variable with `as.numeric`, and voila!

```
lego_movie %>%
  html_node("strong span") %>%
  html_text() %>%
  as.numeric()
```

```
## [1] 7.8
```

Let's say we also want to look at the cast of the movie. We would repeat the same process with a different CSS selector, and this time, because there are multiple members of the cast, we use the plural function `html_nodes`

```
lego_movie %>%
  html_nodes(".itemprop .itemprop") %>%
  html_text()
```

```
##  [1] "Will Arnett"     "Elizabeth Banks" "Craig Berry"
##  [4] "Alison Brie"     "David Burrows"   "Anthony Daniels"
##  [7] "Charlie Day"     "Amanda Farinos"  "Keith Ferguson"
## [10] "Will Ferrell"    "Will Forte"      "Dave Franco"
## [13] "Morgan Freeman"  "Todd Hansen"     "Jonah Hill"
```

We can also pull image locations on the web via `html_node`, except instead of text this time we're interested in getting an HTML attribute called 'src' from the `html_attr` function. We can actually launch a browser instance from `R` with the `browseURL` function.

```
poster <- lego_movie %>%
  html_node(".poster img") %>%
  html_attr("src")
#browseURL(poster)
```

Any information you see on the page, you can generally get. There are some exceptions (e.g. sometimes embedded objects are quite difficult to scrape), but scraping is incredibly powerful.

You could even pull reviews for a movie, for instance if you wanted to do some sentiment analysis on the text to compute ratings from the reviews left for a movie.

```
review <- lego_movie %>%
  html_node("#titleUserReviewsTeaser p") %>%
  html_text()
review
```

```
## [1] "I'd be surprised if anyone saw this coming. The Lego Movie is quite simply unlike anything seen
```

This gets us the latest review for The Lego Movie.

We aren't doing anything computationally intensive or generating much traffic, but if you were to do something intensive, like scraping all the tweets ever produced by members of Congress, then you need to be nice to whomever you're scraping from. One of the main ways to do this is to limit yourself to a certain number of requests per minute/hour/day, and one of the easiest ways to do this is with the `Sys.sleep` function. It takes one argument, which is a scalar for how many seconds to wait. You can put this at the end of a loop or apply function to go easy on the servers that are graciously giving you what you ask for.

Many webmasters who run big sites with lots of data are also aware that many people want to scrape what they have. There are certain ways to detect traffic that seems like scraping, and in many cases sites will rate limit you (throttle your traffic) or kick you out if you make too many requests in a given time.

Twitter API

Scraping is useful, and you can pull down just about anything from a webpage, but it's also messy. You have to look through the page for the part of the source code you're interested in, then download the information from the site and sort through it yourself. With this in mind (and so they have more control over what happens to their data) many sites develop APIs as the preferred way of interacting with their data.

That is true of Twitter. They have an API that is quite easy to use, and for some things it's very useful.

In order to access an API, you usually need to register for an API key. Here's how to do so for the Twitter API:

1) go to dev.twitter.com and sign in
2) click on your username (top-right of screen) and then on "My applications"
3) click on "Create a new application"
4) fill name, description, and website (it can be anything, even google.com)
5) Agree to user conditions and enter captcha.
6) Generate access tokens
7) copy API key, API secret, access token, and access token secret and paste below

For instance, here is my Twitter API key info:

```r
api_key <- "UUUnyVDRZoYQfoAGRTVk8E7C9"
api_secret <- "5cuPOUJTFj1S3H69mGzSQn0GTSCahnJWgFRhA82TcihSSmDYVG"
access_token <- "836246435314032644-aqUYRNAMAgy3hlKsJZAIKOM3C1EFI8I"
access_token_secret <- "ZXHuVdH30jvGZQP9v1IRwOjo1ZVicy3eV5TJyFYbErTtu"
setup_twitter_oauth(api_key, api_secret, access_token, access_token_secret)
```

```
## [1] "Using direct authentication"
```

Let's take a look at Twitter mentions of Bernie Sanders.

```r
# Grab latest tweets
tweets_sanders <- searchTwitter('@BernieSanders', n=1500)

# Loop over tweets and extract text
feed_sanders <- unlist(lapply(tweets_sanders, function(t) t$getText()))
class(feed_sanders)
```

```
## [1] "character"
```

```r
head(feed_sanders)
```

```
## [1] "@TomPerez @BernieSanders What are we doing out there?"
## [2] "RT @AlanDicken1: .@BernieSanders can you help #FreeMaribel!? https://t.co/4wOmkkJQdH"
## [3] "RT @BernieSanders: I want to congratulate @JamesThompsonKS and the progressive community in Kans
## [4] "RT @TaxMarchChicken: Leave it to @BernieSanders to remind us why we're marching this weekend #fo
## [5] "@TRUMPSYOURPREZ @JB_Parrothead @TRiddle_Me_This @LibertyIsALady @Woolecox @CaesarOctavius0 @blal
## [6] "RT @JamesThompsonKS: @BernieSanders Thank you sir!  You inspired me to run and try to make a dif
```

We can also grab information from the man himself.

```r
bernie <- getUser('BernieSanders')
class(bernie)
```

```
## [1] "user"
## attr(,"package")
## [1] "twitteR"
```

What's up with this `bernie` object? It's actually an environment in itself, meaning we can run functions through it, but only those associated with it. What functions are associated with it? There are a couple ways to figure it out.

1) Go to the help file for `getUser` (?getUser) and click on the link for `user` in the 'Value' section.
2) Go look at the documentation for the package online
3) Look in the console by typing the object name (bernie) and then the dollar sign. In the most recent versions of RStudio it will display a list of fields/methods/functions for the object. Unfortunately this won't tell you what they do.

The `toDataFrame()` method will turn all the information inherent in the object into a dataframe, so you can look at it more easily.

```r
bern <- bernie$toDataFrame()
```

Want to know who Bernie Sanders follows? Use the `getFriends()` method. This will return a list of user objects where each element is a user that Bernie Sanders follows.

```r
bernie.follows <- bernie$getFriends(n=100) #Takes a loooong time if you get everything.
bernie.follows <- lapply(bernie.follows, function(x) x$screenName) %>%
                                      unlist(.) %>%
                                      as.character(.) #Why does this work to save it as the object we
head(bernie.follows)
```

```
## [1] "JamesThompsonKS" "tomperriello"    "GeorgeTakei"     "allinwithchris"
## [5] "NekoCase"        "knockeverydoor"
```

The fact that Bernie follows Neko Case is pretty great.

The thing we still haven't gotten is tweets from Bernie himself. We got his mentions earlier, but how do we get the content he produces? The `userTimeline()` function.

```r
bernie.tweets <- userTimeline('BernieSanders', n=20)
bernie.tweets <- unlist(lapply(bernie.tweets, function(x) x$text))
bernie.tweets
```

```
##  [1] ".@JamesThompsonKS The entire country now understands that grassroots America is standing up and
##  [2] "I want to congratulate @JamesThompsonKS and the progressive community in Kansas for running a g
##  [3] "The election results in Kansas show that people are catching on to the fact that Donald Trump i
##  [4] "Our military budget is larger than the next 12 nations combined. We do not need to spend more.
##  [5] "It is easy to go to war with other countries. It is not so easy to comprehend the unintended co
##  [6] "The goal of a health care system should be to keep people well, not to make stockholders rich.
##  [7] "Women are making 81% of what men make and for women of color it<U+0092>s even less. That's noth
##  [8] "I oppose the nomination of Neil Gorsuch. I can't support giving a man with his views a lifetime
##  [9] ".@tomperriello Tom was the first major statewide candidate in VA to run on a $15 minimum wage &
## [10] ".@tomperriello .@tomperriello is standing up to fossil fuel corporations in opposing two fracke
## [11] "Now more than ever we need people in elected office who will fight for working families. That<U
```

What gives? We asked for 20 tweets but only got 11. This is the downside of the Twitter API. It doesn't give you everything you ask for. Not even close in many cases.

We've been using the REST API, meaning it's an API that updates every day, but it's static. We're pinging the archived Tweets, rather than the current stream of tweets. You can do this with the `streamR` package, but I wouldn't recommend it. Because there are more than 500 million tweets per day, Twitter heavily limits what they'll give you. It makes sense, since there is SO MUCH data and people making dumb requests could probably crash their servers, but in many cases you'll only get something like 1% of the tweets on a certain topic with the streaming API. Even with the REST API, it doesn't give you everything. There are ways mostly around this, but it may be beyond the scope of this class.

But! There are lots of things you can do with Twitter data, like measure public opinion, predict trends, sentiment anlaysis, etc.

The podcast Planet Money recently built a bot to buy or sell stocks based on President Trump's tweets (buy if he mentions a company positively, sell if a company is mentioned negatively). They made money! Kind of.

Outside of Twitter, there are lots of APIs that are useful. For instance, ProPublica has a capitolwords API that allows scraping of the Congressional Record (everything that gets said on the floor of Congress). LexisNexis also has an API with which you can access their data, but you need to have access. I'm not sure if Duke does. Pro tip: DO NOT scrape from LexisNexis. They have pursued legal action against some folks who have. Use their API instead.

Next week

- Panel data
- Anything else fun I can think of between now and then
- It's technically after graduate classes end, but I'm planning to hold lab unless that's a problem for y'all