Lecture 4: Tools for data analysis, exploration, and transformation: plyr and reshape2

LSA 2013, LI539
Mixed Effect Models

Dave Kleinschmidt

Brain and Cognitive Sciences
University of Rochester

December 3, 2013

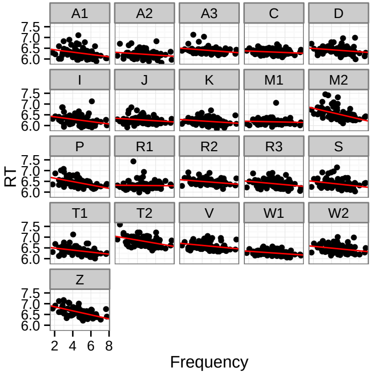
Data manipulation and exploration with plyr and reshape

- Today we'll look at two data manipulation tools which are flexible and powerful, but easy to use once you grasp a few concepts.
- First is `plyr`, which extends functional programming tools in R (like `lapply`) and makes the common data-analysis split-apply-combine procedure easy and elegant.
- Second is `reshape`(2), which makes it easy to change the format of data frames and arrays from "wide" (observations spread across columns) and "long" (observations spread across rows) formats.
- Both are written by Hadley Wickham (like `ggplot2`).
- (you can download the `knitr` source for these slides on my website)

split-apply-combine

```
freqEffects <- ddply(lexdec, .(Subject), function(df) {coef(lm(RT~Frequency, data=df))})
```

```
##   Subject (Intercept) Frequency
## 1      A1       6.533  -0.05379
## 2      A2       6.355  -0.02832
## 3      A3       6.552  -0.03251
## 4       C       6.403  -0.01701
## 5       D       6.551  -0.03048
## 6       I       6.514  -0.05500
```



split-apply-combine

- `plyr` is built around the conceptual structure of **split-apply-combine**
- **split** your data up in some way.
- **apply** some function to each part.
- **combine** the results into the output structure.
- This is a common structure in many data analysis tasks, and R already has some facilities for it.
- `plyr` unifies these in a single interface and provides some nice helper functions, and also makes the split-apply-combine structure explicit.
- Before we get to `plyr` itself, let's have a short review of some basic functional programming concepts.

functions: how do they work

- Formally: take some input, do something, and produce some output.
- You use functions in R all the time.
- Most of the functions you're familiar with have names and are built in (or provided by libraries).

```
mean(runif(100))

## [1] 0.4781
```

- But there's nothing special about functions in R, they're objects, just like any other data type
- This means they can, for instance, be assigned to variables:

```
f <- mean
f(runif(100))

## [1] 0.4859
```

(anonymous) functions: how do they work

- Functions are objects that are created with the `function` keyword

```
function(x) sum(x)/length(x)

## function(x) sum(x)/length(x)
## <environment: 0x1044f3710>
```

- Functions are by their nature "anonymous" in R, and have no name, in the same way that the vector `c(1,2,3)` is just an object, with no intrinsic name (this is unlike other languages, like Java or C).
- New functions can be assigned to variables to be called over and over again

```
mymean <- function(x) sum(x)/length(x)
mymean(runif(100))

## [1] 0.467

mymean(runif(100, 1, 2))

## [1] 1.529
```

- ...or just evaluated once

```
(function(x) sum(x)/length(x)) (runif(100))

## [1] 0.5171
```

(notice the parentheses around the whole function definition)

functions, environments, and closures

- A function object lists an environment when it's printed to the console
- This is because functions are really *closures* in R.
- They include information about the values of variables *when the function was created*.
- You can take advantage of this to make "function factories":

```
make.power <- function(n) {
  return(function(x) x^n)
}
my.square <- make.power(2)
my.square(3)

## [1] 9

(make.power(4)) (2)

## [1] 16
```

- See Hadley Wickham's excellent chapter on functional programming in R for more on this: `https://github.com/hadley/devtools/wiki/Functional-programming`

functions: how do they work

- Function declarations have three parts:
  1. The `function` keyword
  2. Comma-separated list of function arguments
  3. The body of the function, which is an expression (multi-statement expression should be enclosed in braces {}). The value of the expression is used for the returned value of the function if no `return` statement is encountered in the body.
- For instance:

```
mean.and.var <- function(x) {
  m <- mean(x)
  v <- var(x)
  data.frame(mean=m, var=v)
}
```

a few function tips

- The ellipsis ... can be included in the arguments list and “captures” any arguments not specifically named. This is useful to pass on other arguments to other function calls in the body (as we’ll see later).
- You can specify default values for arguments by `argument=default`.
- R has very sophisticated argument resolution when a function is called. It first assigns named arguments by name, and then unnamed arguments are assigned positionally to unfilled arguments. So you can say something like

```
sd(rnorm(sd=5, mean=1, 100))

## [1] 4.912
```

where the last argument is interpreted as `n`, even though the specification of `rnorm` calls for `n` to be first:

```
rnorm

## function (n, mean = 0, sd = 1)
## .Internal(rnorm(n, mean, sd))
## <bytecode: 0x104989510>
## <environment: namespace:stats>
```

Your first foray into functional programming

- R is a functional programming language at its heart.
- One of the most basic operations of functional programming is to apply a function individually to items in a list.
- In base R, this is done via `lapply` (for list-apply) and friends:

```
list.o.nums <- list(runif(100), rnorm(100), rpois(100, lambda=1))
lapply(list.o.nums, mean)

## [[1]]
## [1] 0.4703
##
## [[2]]
## [1] 0.03472
##
## [[3]]
## [1] 0.89
```

- The “big three” apply functions in R are `lapply` (takes and returns a list), `sapply` (like `lapply` but attempts to simplify output into a vector or matrix), and `apply` (which works on arrays).

unleash the power of anonymous functions

- When combined with `lapply` and friends, anonymous functions are extremely powerful.
- You could, for instance, run a simulation with a range of parameter values:

```
sapply(1:10, function(n) rpois(5, lambda=n))

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    2    1    2    4   10   10    7    6     6
## [2,]    1    3    4    6    3    5    9    6   17     9
## [3,]    0    0    2    7    5    4    5   13   12     5
## [4,]    1    1    3    3    5    6    6    4   15     9
## [5,]    0    3    3    5    2    8    8   10    8     6
```

- Or repeat the same simulation multiple times, calculating a summary statistics for each repetition:

```
sapply(1:10, function(n) mean(rnorm(n=5, mean=0, sd=1)))

##  [1]  0.20120  0.03167  0.49330 -0.07796  0.08483  0.01232 -0.68791
##  [8] -0.21145  0.13870 -0.52366
```

split-apply-combine

- You might also use `sapply` calculate the mean RT (for instance) for each subject, by using `split` to create a list of each subject's RTs:

```
data(lexdec, package='languageR')
RT.bysub <- with(lexdec, split(RT, Subject))
RT.means.bysub <- sapply(RT.bysub, mean)
head(data.frame(RT.mean=RT.means.bysub))
```

```
##    RT.mean
## A1   6.278
## A2   6.220
## A3   6.398
## C    6.322
## D    6.406
## I    6.253
```

split-apply-combine

- This is a common data-analysis task: split up the data in some way, analyze each piece, and then put the results back together again.
- The `plyr` package (Wickham, 2011) was designed to facilitate this process.
- For instance, instead of that `split`/`sapply` combo, we could use the `ddply` function:

```
library(plyr)
head(ddply(lexdec, .(Subject), function(df) data.frame(RT.mean=mean(df$RT))))

##   Subject RT.mean
## 1      A1   6.278
## 2      A2   6.220
## 3      A3   6.398
## 4       C   6.322
## 5       D   6.406
## 6       I   6.253
```

split-apply-combine

The `ddply` call has three parts:

```
ddply(lexdec, .(Subject), function(df) data.frame(RT.mean=mean(df$RT)))
```

1. The data, `lexdec`
2. The splitting variables, `.(Subject)`. The `.()` function is a utility function which quotes a list of variables or expressions. We could just as easily have used the variable names as strings `c("Subject")` or (one-sided) formula notation `~Subject`.
3. The function to apply to the individual pieces. In this case, the function takes a `data.frame` as input and returns a `data.frame` which has one variable—`RT.mean`. The splitting variables are automatically added before the results are combined.

plyr functions: input

- Plyr commands are named based on their input and output.
- The first letter refers to the format of the input.
- The input determines how the data is split:
  - `d*ply(.data, .variables, .fun, ...)` takes data frame input and splits it into subsets based on the unique combinations of the `.variables`.
  - `l*ply(.data, .fun, ...)` takes list input, splitting the list and passing each element to `.fun`.
  - `a*ply(.data, .margins, .fun, ...)` takes array input, and splits it into sub arrays by `.margins` (just like base R `apply`). For instance, if `.margins = 1` and `.data` is a three-D array, then `.data[1, , ], .data[2, , ], ...` are each passed to `.fun`.

plyr functions: output

- The second letter of the command name refers to the output format
- The output determines how the data is combined at the end:
  - `*dply` takes the result of its `.fun` and turns it into a data frame, then adds the splitting variables (values of `.variables` for `ddply`, list names for `ldply`, or array dimnames for `adply`) before `rbind`ing the individual data frames together
  - `*lply` just returns a list of the result of applying `.fun` to each individual split, just like `lapply`, but additionally adds names based on the splitting variables.
  - `*aply` tries to assemble the output of `.fun` into a big array, where the combine dimensions are the last ones. For instance, if `.fun` returns a two-dimensional array (always of the same size), and there were three splitting variables or dimensions originally, then the output would be a five-dimensional array, with dimensions 1 to 2 corresponding to the `.fun` output dimensions and 2 to 5 the splitting variables.

try it: output behavior of plyr commands

Task

Let's use the lexdec data set to explore the output behavior of plyr. Start with this ddply call (copy and paste from the slides pdf, or from the accompanying .R file):

```r
library(plyr)
data(lexdec, package='languageR')       # load the dataset if it isn't already
ddply(lexdec, .(PrevType, Class), function(df) with(df, data.frame(meanRT=mean(RT))))
```

1. What does this do? Look at the lexdec data frame, run the command, and interpret the output.
2. Are these numbers "really" different? Change the function to also return the variance (or standard deviation or standard error, or whatever other measure you think might be useful).
3. Change it to return a list instead using dlply. The output might look a little funny. Why? Use str to investigate the output.
4. Now make it return an array using daply. The output will probably look totally wrong. Why? (Hint: use str to look at the output, again). Fix it so that it does what you'd expect/like it to do.

subset

- An added level of convenience comes from the fact that any extra arguments to, e.g., `ddply` are passed to the function which operates on each piece
- This means you can use functions like `subset` or `transform` which take a data frame and return another data frame.
- For instance, to find the trial with the slowest RT for each subject, split by `Subject` and then use `subset`:

```
slowestTrials <- ddply(lexdec, .(Subject), subset, RT==max(RT))
head(slowestTrials[, c('Subject', 'RT', 'Trial', 'Word')])

##   Subject    RT Trial     Word
## 1      A1 7.115    79 tortoise
## 2      A2 6.832    66     lion
## 3      A3 7.132   157   radish
## 4       C 6.680   145     frog
## 5       D 6.984   172  chicken
## 6       I 7.136    48    snake
```

- This is equivalent to both

```
ddply(lexdec, .(Subject), function(df, ...) subset(df, ...), RT==max(RT))
ddply(lexdec, .(Subject), function(df) subset(df, RT==max(RT)))
```

transform

- Another super convenient function is `transform`, which adds variables to a data frame (or replaces them) using expressions evaluated using the data frame as the environment (like the `with` function).
- For instance, we often standardize measures before regression (center and possibly scale).
- If the reaction time distributions of individual subjects are very different, then we might want to standardize them for each subject individually. In "verbose" `ddply`, we could do

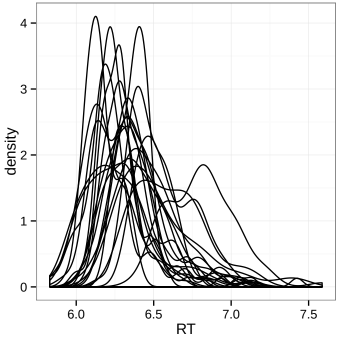
```
ddply(lexdec, .(Subject), function(df) {
  df$RT.s <- scale(df$RT)
  return(df)
})
```

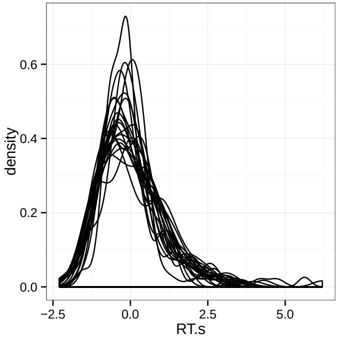
- However, we can be more concise using `transform`:

```
lexdecScaledRT <- ddply(lexdec, .(Subject), transform, RT.s=scale(RT))
```

This expresses very transparently what we're trying to do: transform the data by adding a variable for the scaled (zero mean and unit sd) reaction time.

transform





summarise

- `plyr` also provides the convenience function

  `summarise`

  (with an s!).
- This function, like `transform`, takes the form

```
summarise(.data, summVar1=expr1, summVar2=expr2, ...)
```

  but unlike transform it creates a *new* data frame with only the specified summary variables.

summarise

- For instance, to find the mean and variance of each subject's RT, we could use

```
lexdec.RTsumm <- ddply(lexdec, .(Subject), summarise, mean=mean(RT), var=var(RT))
head(lexdec.RTsumm)

##   Subject  mean     var
## 1      A1 6.278 0.05419
## 2      A2 6.220 0.03204
## 3      A3 6.398 0.02408
## 4       C 6.322 0.01544
## 5       D 6.406 0.03289
## 6       I 6.253 0.04960
```

- This is more concise than the similar example a few slides ago

```
ddply(lexdec, .(Subject), function(df) with(df, data.frame(meanRT=mean(RT))))
```

and, like with `transform`, makes our intentions much clearer.

transform+summarise exercises

Task

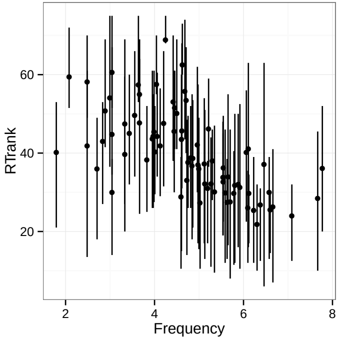
Let's investigate the relative ordering of RTs for different words

1. Add a new variable `RTrank` which is the rank-order of the RT for each trial, by subject. That is, `RTrank=1` for that subject's fastest trial, 2 for the second-fastest, etc. Hint: `rank` finds the rank indices of a vector.
2. Find the average RT rank for each word, using `summarise`.
3. Plot the relationship between the word frequencies and their average rank.
4. If you're feeling fancy, put errorbars on the words showing the 25% and 75% quantiles.
5. Which word has the highest average RT rank? The lowest?

transform+summarise solution

```
lexdec <- ddply(lexdec, .(Subject), transform, RTrank=rank(RT))


word.rt.ranks <- ddply(lexdec, .(Word, Frequency), summarise,
                       RTrank25=quantile(RTrank, 0.25),
                       RTrank75=quantile(RTrank, 0.75),
                       RTrank=mean(RTrank))
```

```
ggplot(word.rt.ranks, aes(x=Frequency, y=RTrank, ymin=RTrank25, ymax=RTrank75)) +
  geom_pointrange()
```



```
subset(word.rt.ranks,
       RTrank %in% c(max(RTrank), min(RTrank)))

##        Word Frequency RTrank RTrank25 RTrank75
## 3     apple     6.304  21.79       10       32
## 75 vulture     4.248  68.88       68       75
```

example use cases

Let's go through some exampes of how you might use `plyr` for

- Data analysis and exploration.
- Exploring models through simulation.

Checking distribution of errors

- Let's check to see whether the errors in `lexdec` responses are evenly distributed across native and non-native speakers.
- There are a couple of ways to do this. We could use `ddply` and `summarise` like above:

```
ddply(lexdec, .(NativeLanguage), summarise, acc=mean(Correct=='correct'))

##   NativeLanguage    acc
## 1        English 0.9705
## 2          Other 0.9480
```

- We could also use `daply` to get an array of raw counts of correct and incorrect responses, by splitting on `NativeLanguage` *and* `Correct` and then extracting the number of rows in each split:

```
daply(lexdec, .(NativeLanguage, Correct), nrow)

##               Correct
## NativeLanguage correct incorrect
##        English     920        28
##        Other       674        37
```

Checking distribution of errors

Why might we want the latter option? It's the format that `chisq.test` expects:

```
correctCounts <- daply(lexdec, .(NativeLanguage, Correct), nrow)
chisq.test(correctCounts)

##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  correctCounts
## X-squared = 4.884, df = 1, p-value = 0.02711
```

Estimate the frequency effect for each subject

- Let's estimate the effect of frequency on RT for each subject separately (perhaps to get a sense of whether to include random slopes in a mixed effects model).
- We can do this using a combination of `ddply` and `coef`:

```
subjectSlopes <- ddply(lexdec, .(Subject), function(df) {coef(lm(RT~Frequency, data=df))})
```

- We can see that these slopes show a fair amount of variability,

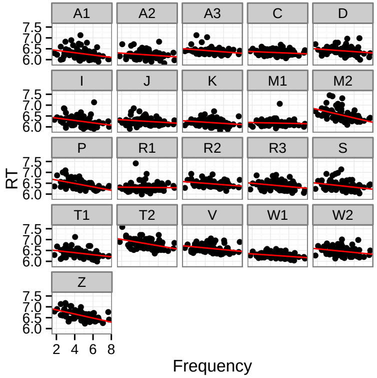
```
summary(subjectSlopes$Frequency)

##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.09810 -0.05380 -0.03830 -0.04290 -0.02830 -0.00403
```

so it might make sense to include random slopes in later regression modeling.

Estimate the frequency effect for each subject

As a sanity check, we can also plot the fitted regression lines against the original data points:

```
ggplot(lexdec, aes(x=Frequency, y=RT)) +
  geom_point() +
  facet_wrap(~Subject) +
  geom_abline(data=subjectSlopes, aes(slope=Frequency, intercept=`(Intercept)`), color='red')
```



data exploration and errorbars

- Investigate interaction between frequency and native language background.
- Let's first construct a factor variable which is a binary high frequency-low frequency variable.

```
lexdec <- transform(lexdec,
                    FreqHiLo=factor(ifelse(Frequency>median(Frequency),
                                           'high', 'low'),
                                    levels=c('low', 'high')))
```

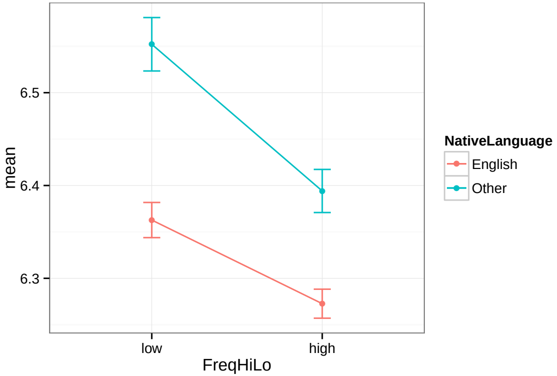
- Then, use ddply and summarise to create a summary table for your conditions of interest

```
se <- function(x) sd(x)/sqrt(length(x))
langfreq.summ <- ddply(lexdec,
                       .(NativeLanguage, FreqHiLo),
                       summarise, mean=mean(RT), se=se(RT))
```

data exploration and errorbars

- We can quickly plot condition means and 95% CIs using `ggplot` (etc.) and the `ddply` output

```
ggplot(langfreq.summ, aes(x=FreqHiLo, color=NativeLanguage,
                          y=mean, ymin=mean-1.96*se, ymax=mean+1.96*se)) +
  geom_point() +
  geom_errorbar(width=0.1) +
  geom_line(aes(group=NativeLanguage))
```



Wow! What a huge effect!

- ...but wait.
- Calculating the standard error in this way assumes that, for the purposes of comparing the condition means, each observed RT is an independent draw from the same normal distribution.
- But in fact, they are *not*: observations are grouped both by subject and by item (word).
- One way of dealing with this: look at the by-subject standard error, by averaging within each subject and then treating each subject as an independent draw from the underlying condition.
- This is the “by-subject” analysis, like the “F1” ANOVA.

by-subject standard errors with ddply

Computing the by-subject standard errors is a two-step process, both of which can be done with a single ddply command:

1. Average within each subject and combination of conditions:

```
langfreq.bysub <- ddply(lexdec, .(NativeLanguage, FreqHiLo, Subject),
                        summarise, RT=mean(RT))
```

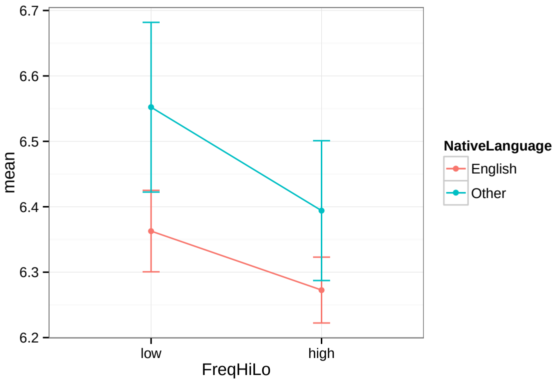
2. Then, calculate the condition means and standard errors as before:

```
langfreq.bysub.summ <- ddply(langfreq.bysub,
                             .(NativeLanguage, FreqHiLo),
                             summarise, mean=mean(RT), se=se(RT))
```

by-subject standard errors

When we plot the resulting error bars, the effects look much smaller compared to the variability across subjects (and small number of subjects):

```
ggplot(langfreq.bysub.summ, aes(x=FreqHiLo, color=NativeLanguage,
                                y=mean, ymin=mean-1.96*se, ymax=mean+1.96*se)) +
  geom_point() +
  geom_errorbar(width=0.1) +
  geom_line(aes(group=NativeLanguage))
```



try it: by-item standard errors

Task

- Use `ddply` to do the “F2” or by-item analysis, finding *by-item* standard errors, treating the words as items.
- Plot the resulting errorbars, and use this (and the actual `ddply` output) to interpret the results.

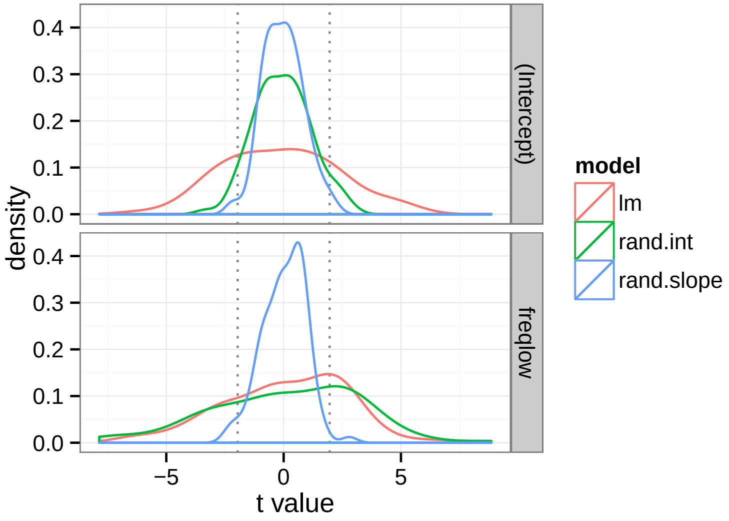
by-item standard errors (solution)

```
langfreq.byitem <- ddply(lexdec, .(NativeLanguage, FreqHiLo, Word), summarise, RT=mean(RT))
langfreq.byitem.summ <- ddply(langfreq.byitem,
                             .(NativeLanguage, FreqHiLo),
                             summarise, mean=mean(RT), se=se(RT))

## ggplot(langfreq.byitem.summ, aes(x=FreqHiLo, color=NativeLanguage,
##                              y=mean, ymin=mean-2*se, ymax=mean+2*se)) +
##   geom_point() +
##   geom_errorbar(width=0.1) +
##   geom_line(aes(group=NativeLanguage))
```

A few other plyr tricks

- The `plyr` functions `m*ply` and `r*ply` are wrappers for other forms which make simulations more convenient. Check out the documentation (`?mdply`) and the `plyr` article in J. Stat. Software.
- Because they combine things in nice ways, `plyr` functions can help R data functions play nicely together.
    - To concatenate a list of data frames into one big data frame, you can use `ldply(list.of.dfs, I)` (the identity function `I` just returns its input).
    - To “shatter” an array into a data frame where the dimension names are stored in columns, you can use `adply(an.array, 1:ndim(an.array), I)`. Any margins left out then index rows. If any dimensions are named, they will be transferred to the data frame in a smart way.
- Use `subset` and `ddply` to remove outliers subject-by-subject
- Check balance (number of trials/subjects in each condition) using `nrow` and `ddply` of `daply` (like the correct/incorrect example).

Simulating data sets and model exploration



Simulating data sets and model exploration

- The best way to understand a model is to simulate fake data and see what the model does with it.
- Frequently the process goes as follows:
  1. Pick some range of parameter values (random effect variance vs. residual variance).
  2. Generate some data using those parameters
  3. Fit model to that data, and record summary statistics.
- This fits well within the split-apply-combine pattern of `plyr`.
- For example, let's look at how **not accounting for random slopes and intercepts inflates Type I error rates**.
- We'll generate fake data for a binary "frequency" variable which has a true effect of 0, then fit `lm` and `lmer` models with random intercept and slope.
- Let's start simple, fitting the models to *one* set of parameters, repeating the simulation 100 times.

step 1: simulate data

```r
library(plyr)
library(mvtnorm)
library(lme4)
make.data.generator <- function(true.effects=c(0,0),
                                resid.var=1,
                                ranef.var=diag(c(1,1)),
                                n.subj=24,
                                n.obs=24
                                )
{
  # create design matrix for our made up experiment
  data.str <- data.frame(freq=factor(c(rep('high', n.obs/2), rep('low', n.obs/2))))
  contrasts(data.str$freq) <- contr.sum(2)
  model.mat <- model.matrix(~ 1 + freq, data.str)

  generate.data <- function() {
    # sample data set under mixed effects model with random slope/intercepts
    simulated.data <- rdply(n.subj, {
      beta <- t(rmvnorm(n=1, sigma=ranef.var)) + true.effects
      expected.RT <- model.mat %*% beta
      epsilon <- rnorm(n=length(expected.RT), mean=0, sd=sqrt(resid.var))
      data.frame(data.str,
                 RT=expected.RT + epsilon)
    })
    names(simulated.data)[1] <- 'subject'
    simulated.data
  }
}
```

step 2: fit model

```
fit.models <- function(simulated.data) {
  # fit models and extract coefs
  lm.coefs <- coefficients(summary(lm(RT ~ 1+freq, simulated.data)))[, 1:3]
  rand.int.coefs <- summary(lmer(RT ~ 1+freq + (1|subject), simulated.data))@coefs
  rand.slope.coefs <- summary(lmer(RT ~ 1+freq + (1+freq|subject), simulated.data))@coefs
  # format output all pretty
  rbind(data.frame(model='lm', predictor=rownames(lm.coefs), lm.coefs),
        data.frame(model='rand.int', predictor=rownames(rand.int.coefs), rand.int.coefs),
        data.frame(model='rand.slope', predictor=rownames(rand.slope.coefs), rand.slope.coefs))
}
```

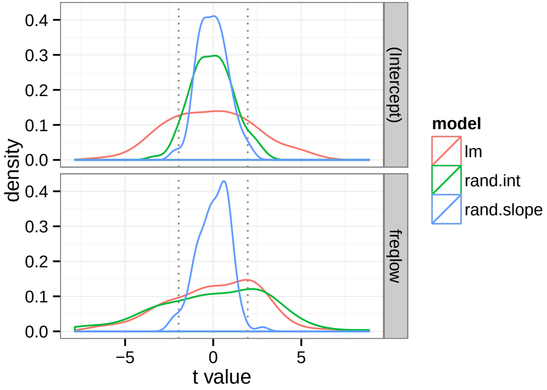
step 3: put it together + repeat

```
gen.dat <- make.data.generator()
simulations <- rdply(.n=100,
                     fit.models(gen.dat()),
                     .progress='text')
```

```
head(simulations)

##   .n      model   predictor Estimate Std..Error t.value
## 1  1         lm (Intercept)  -0.2934    0.09819 -2.9886
## 2  1         lm     freqlow   0.2767    0.13886  1.9925
## 3  1   rand.int (Intercept)  -0.2934    0.19183 -1.5297
## 4  1   rand.int     freqlow   0.2767    0.12062  2.2938
## 5  1 rand.slope (Intercept)  -0.2934    0.27966 -1.0493
## 6  1 rand.slope     freqlow   0.2767    0.43265  0.6395

daply(simulations, .(model, predictor), function(df) type1err=mean(abs(df$t.value)>1.96))

##             predictor
## model         (Intercept) freqlow
##   lm                 0.45    0.52
##   rand.int           0.12    0.62
##   rand.slope         0.03    0.04
```

step 4: visualize

```
# use reshape2::melt to get the data into a more convenient format (see next section)
ggplot(simulations, aes(x=t.value, color=model)) +
  geom_vline(xintercept=c(-1.96, 1.96), color='#888888', linetype=3) +
  scale_x_continuous('t value') +
  geom_density() +
  facet_grid(predictor~.)
```



different parameter values

- What if we want to run the simulation with different sets of parameter values?
- Create a data frame of parameters, using `expand.grid` on arguments which have the same names as the arguments to `make.data.generator`.

```
head(params <- expand.grid(n.obs=c(4, 16, 64), n.subj=c(4, 16, 64)))

##   n.obs n.subj
## 1     4      4
## 2    16      4
## 3    64      4
## 4     4     16
## 5    16     16
## 6    64     16
```

- And then use `mdply` on the result.

```
man.simulations <- mdply(params, function(...) {
                           make.data <- make.data.generator(...)
                           rdply(.n=100, fit.models(make.data()))
                         }, .progress='text')
```

digression: mdply

```
mdply(params, function(...) {
  make.data <- make.data.generator(...)
  rdply(.n=100, fit.models(make.data()))
}, .progress='text')
```

- `mdply` is like `Map`: it passes the variables in a data frame (split row-by-row) as named arguments to the function. The `function(...) {}` syntax means that the function will accept any named arguments, and then recycle them wherever the `...` occurs anywhere inside the body. Thus, this `mdply` will pass the columns of `params` as arguments to the `make.data.generator()` function, no matter which parameters you specify.
- This specific example (where the parameters are `n.obs` and `n.subj`) is equivalent to:

```
ddply(params, .(n.obs, n.subj), function(df) {
  make.data <- make.data.generator(n.obs=df$n.obs, n.subj=df$n.subj)
  rdply(.n=100, .fun=fit.models(make.data()))
}, .progress='text')
```

reshape2

(If we have time): talk about changing format of data using `melt` and `cast` from the `reshape2` package.

What does your data look like?

- Data doesn't always look like tools like ggplot (or ddply) expect it to.
- What if your experiment spits out a data file where each trial is a different column?
- The lexical decision data set might look like this:

```
##   Subject    23_RT 23_Correct    24_RT 24_Correct    25_RT 25_Correct
## 1      A1 6.340359    correct     <NA>       <NA>     <NA>       <NA>
## 2      A2 6.329721    correct     <NA>       <NA> 6.20859    correct
## 3      A3     <NA>       <NA>     <NA>       <NA>     <NA>       <NA>
## 4       C 6.533789    correct     <NA>       <NA>     <NA>       <NA>
## 5       D     <NA>       <NA> 6.232448    correct     <NA>       <NA>
## 6       I     <NA>       <NA> 6.194405    correct     <NA>       <NA>
## 7       J 6.714171    correct     <NA>       <NA>     <NA>       <NA>
```

(there are missing values because non-word trials are excluded)

Wide vs. long data

There are two ways of structuring data:

```
##   Subject Trial variable    value
## 1      A1    23       RT 6.340359
## 2      A1    27       RT 6.308098
## 3      A1    29       RT 6.349139
## 4      A1    30       RT 6.186209
## 5      A1    32       RT 6.025866
```

```
##   Subject    23_RT 23_Correct    24_RT
## 1      A1 6.340359    correct     <NA>
## 2      A2 6.329721    correct     <NA>
## 3      A3     <NA>       <NA>     <NA>
## 4       C 6.533789    correct     <NA>
## 5       D     <NA>       <NA> 6.232448
```

long Each observation gets exactly one row, with values in "id" columns giving identifying information (like subject, trial, whether the observed value is a correct/incorrect response or a RT observation, etc.)

wide Each row contains all the observations for a unique combination of identifying variables (say, one subject). Column names identify the kind of observation in that row (trial number, observation type).

reshaping data

- Converting between wide and long data representations is a common task in data analysis
- (especially data import/cleaning)
- The `reshape2` package streamlines this process in R.
- (Most of the functionality of `reshape2` is a special case of what `plyr` does).

melt and cast

- Two main functions in `reshape2`
- `melt` converts an array or data frame into a long format.
- `dcast` and `acast` convert “molten” data into a range of different shapes from long to wide.

melt with you, (I'll stop the world and)

- Let's start with an example.
- Here's the wide data frame from above:

```
ld.wide[1:5, 1:7]

##   Subject    23_RT 23_Correct    24_RT 24_Correct   25_RT 25_Correct
## 1      A1 6.340359    correct     <NA>       <NA>    <NA>       <NA>
## 2      A2 6.329721    correct     <NA>       <NA> 6.20859    correct
## 3      A3     <NA>       <NA>     <NA>       <NA>    <NA>       <NA>
## 4       C 6.533789    correct     <NA>       <NA>    <NA>       <NA>
## 5       D     <NA>       <NA> 6.232448    correct    <NA>       <NA>
```

- When you `melt` data, you have to specify which variables (columns) are **id variables** and which are **measure variables**.

```
head(ld.m <- melt(ld.wide, id.var='Subject', na.rm=T))

##    Subject variable    value
## 1       A1    23_RT 6.340359
## 2       A2    23_RT 6.329721
## 4        C    23_RT 6.533789
## 7        J    23_RT 6.714171
## 8        K    23_RT 6.011267
## 10      M2    23_RT 6.848005
```

melt

- Now use `str_split` (from the `stringr` package) to separate the trial number and measure information.

```
require(stringr)
trials.and.vars <- ldply(str_split(ld.m$variable, '_'))
names(trials.and.vars) <- c('Trial', 'measure')
```

- `str_split` returns a list of splits but we can use `ldply` to convert to a dataframe, to which we add informative names.
- The extracted trial numbers and RT/correct indicators can then be combined with the melted data with `cbind`.

```
head(ld.m <- cbind(ld.m, trials.and.vars))

##    Subject variable    value Trial measure
## 1       A1    23_RT 6.340359    23      RT
## 2       A2    23_RT 6.329721    23      RT
## 4        C    23_RT 6.533789    23      RT
## 7        J    23_RT 6.714171    23      RT
## 8        K    23_RT 6.011267    23      RT
## 10      M2    23_RT 6.848005    23      RT
```

melt syntax

- To specify the measure and id variables, use `measure.vars=` and `id.vars=` arguments.
- You can specify them as indices (column numbers) or names.
- `melt` will try to guess the id and measure variables if you don't specify them.
- If you specify only measure vars, `melt` will treat the other variables as id variables (and vice-versa)
- If you want some variables ommitted, specify the measure and id variables that you want and the others will be dropped.

cast

- `melt` gets your data into a "raw material" that can be easily converted to other more useful formats.
- Molten data can be converted to different shapes using the `*cast` commands.
- `dcast` creates a data frame, and `acast` creates an array.
- Both commands take molten data and a formula which defines the new shape.

cast

- `dcast` takes a two-dimensional formula. The left hand side tells which variables determine the rows, and the right side the columns
- Let's put RT and correct in their own columns. The `ld.m$measure` variable indicates whether the `ld.m$value` is an RT or a correct measure, so we put that variable on the right-hand side of the formula.

```
head(dcast(ld.m, Subject+Trial ~ measure))

##   Subject Trial Correct       RT
## 1      A1   100 correct 6.126869
## 2      A1   102 correct 6.284134
## 3      A1   106 correct 6.089045
## 4      A1   108 correct 6.383507
## 5      A1   109 correct  6.22059
## 6      A1   111 correct 6.381816
```

cast

- We can also use the shorthand `...` to indicate all other (non-value) variables:

```
head(dcast(ld.m, ... ~ measure))

##   Subject    variable Trial Correct       RT
## 1      A1       23_RT    23    <NA> 6.340359
## 2      A1 23_Correct    23 correct     <NA>
## 3      A1       27_RT    27    <NA> 6.308098
## 4      A1 27_Correct    27 correct     <NA>
## 5      A1       29_RT    29    <NA> 6.349139
## 6      A1 29_Correct    29 correct     <NA>
```

- But this is no good here because `ld.m$variable` also encodes information about `measure`, so we have to remove it first to be able to use `...`

```
ld.m$variable <- NULL
head(dcast(ld.m, ... ~ measure))

##   Subject Trial Correct       RT
## 1      A1   100 correct 6.126869
## 2      A1   102 correct 6.284134
## 3      A1   106 correct 6.089045
## 4      A1   108 correct 6.383507
## 5      A1   109 correct  6.22059
## 6      A1   111 correct 6.381816
```

cast syntax

- Specify the shape of the “cast” data using a formula. For each combination of the values of variables on the left-hand side, there will be one row, and likewise for columns with the right-hand side.
- For `dcast`, the data frame will also have left-hand variables in columns in the resulting data frame. Right-hand variables will have their values pasteed together as column names for the other columns.
- If you want higher-dimensional output, you can use `acast` which creates an array (specify dimensions like `dim1var1 ~ dim2var1 + dim2var2 ~ dim3var1`).
- If there is no variable called `value`, then `cast` will try to guess. You can override the defaults by specifying the `value.var=` argument.

cast aggregation

- If the formula results in more than one value in each cell, you need to specify an aggregating function (like in `ddply`) via the `fun.aggregate=` argument (you can abbreviate to `fun.agg=`).
- The default is `length` which tells you how many observations are in that cell.

```
head(dcast(ld.m, Subject ~ measure))

##   Subject Correct RT
## 1      A1      79 79
## 2      A2      79 79
## 3      A3      79 79
## 4       C      79 79
## 5       D      79 79
## 6       I      79 79
```

- The function you specify must return a single value (more constrained than `plyr`).

"but can't I just do this in Excel?"

- You can!
- But it will be tedious and you *will* make mistakes.
- Using tools designed for these data-manipulation tasks makes you be explicit about the things you are doing to your data.
- And when you are done, you have a script which is a complete record of what you did (and, if you're using knitr, a nicely formatted report, too).

plyr and reshape (melt)

- The `reshape` library operations are conceptually related to the split-apply-combine logic of `plyr`.
- Question: what's the `plyr` equivalent of the `melt` command we saw before?

```
melt(ld.wide, id.var='Subject', na.rm=T)
```

- Remember what `melt` does: split the data by the id variables, and rearrange the measure variable columns so that they're in one `value` column, moving the column names into a new `variable` column.

```
ddply(ld.wide, .(Subject), function(df) {
  vars <- names(df)
  vals <- t(df)
  dimnames(vals) <- NULL
  return(subset(data.frame(variable=vars, value=vals),
                variable != 'Subject' & !is.na(value)))
})
```

plyr and reshape (cast)

Question: how would you write the dcast command from before?

```
head(dcast(ld.m, Subject+Trial ~ measure))
```

```
head(ddply(ld.m, .(Subject, Trial), function(df) {
  res <- data.frame(t(df$value))
  names(res) <- df$measure
  return(res)
}))

##   Subject Trial       RT Correct
## 1      A1   100 6.126869 correct
## 2      A1   102 6.284134 correct
## 3      A1   106 6.089045 correct
## 4      A1   108 6.383507 correct
## 5      A1   109  6.22059 correct
## 6      A1   111 6.381816 correct
```