Exploring data 2

Lists

Lists

Lists are the “kitchen sink” of R objects. They can be used to keep together a variety of different R objects of different classes, dimensions, and structures in a single R object.

Because there are often cases where an R operation results in output that doesn’t have a simple structure, lists can be a very useful way to output complex output from an R function.

Most lists are not “tidy” data. However, we’ll cover some ways that you can easily “tidy” some common list objects you might use a lot in your R code, including the output of fitting linear and generalized linear models.

Lists

```r
example_list <- list(a = sample(1:10, 5),
                     b = data_frame(letters = letters[1:3],
                                    number = 1:3))
example_list
```

```
## $a
## [1]  5  1  9  8 10
##
## $b
## # A tibble: 3 x 2
##   letters number
##     <chr>  <int>
## 1       a      1
## 2       b      2
## 3       c      3
```

Indexing lists

To pull an element out of a list, you can either use $ or [[]] indexing:

```
example_list$a
```

```
## [1]  5  1  9  8 10
```

```
example_list[[2]]
```

```
## # A tibble: 3 x 2
##   letters number
##     <chr>  <int>
## 1       a      1
## 2       b      2
## 3       c      3
```

Exploring lists

If an R object is a list, running `class` on the object will return “list”:

```r
class(example_list)
```

```
## [1] "list"
```

Often, lists will have names for each element (similar to column names for a dataframe). You can get the names of all elements of a list using the `names` function:

```r
names(example_list)
```

```
## [1] "a" "b"
```

Exploring lists

The `str` function is also useful for exploring the structure of a list object:

```
str(example_list)
```

```
## List of 2
##  $ a: int [1:5] 5 1 9 8 10
##  $ b:Classes 'tbl_df', 'tbl' and 'data.frame':   3 obs. of  2
##   ..$ letters: chr [1:3] "a" "b" "c"
##   ..$ number : int [1:3] 1 2 3
```

Lists versus dataframes

As a note, a dataframe is actually just a very special type of list. It is a list where every element (column in the dataframe) is a vector of the same length, and the object has a special attribute specifying that it is a dataframe.

```
example_df <- data_frame(letters = letters[1:3],
                         number = 1:3)
class(example_df)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
is.list(example_df)
```

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

Regression models

`nepali` example data

For the `nepali` dataset, each observation is a single measurement for a child; there can be multiple observations per child.

I'll limit it to the columns with the child's id, sex, weight, height, and age, and I'll limit to each child's first measurement.

```r
nepali <- nepali %>%
  # Limit to certain columns
  select(id, sex, wt, ht, age) %>%
  # Convert id and sex to factors
  mutate(id = factor(id),
         sex = factor(sex, levels = c(1, 2),
                      labels = c("Male", "Female"))) %>%
  # Limit to first obs. per child
  distinct(id, .keep_all = TRUE)
```

nepali example data

The data now looks like:

```r
head(nepali)
```

```
##       id    sex   wt    ht age
## 1 120011   Male 12.8  91.2  41
## 2 120012 Female 14.9 103.9  57
## 3 120021 Female  7.7  70.1   8
## 4 120022 Female 12.1  86.4  35
## 5 120023   Male 14.2  99.4  49
## 6 120031   Male 13.9  96.4  46
```

Formula structure

Regression models can be used to estimate how the expected value of a *dependent variable* changes as *independent variables* change.

In R, regression formulas take this structure:

```
## Generic code
[response variable] ~ [indep. var. 1] +  [indep. var. 2] + ...
```

Notice that ~ used to separate the independent and dependent variables and the + used to join independent variables. This format mimics the statistical notation:

$$Y_i \sim X_1 + X_2 + X_3$$

You will use this type of structure in R for a lot of different function calls, including those for linear models (`lm`) and generalized linear models (`glm`).

Linear models

To fit a linear model, you can use the function `lm()`. Use the `data` option to specify the dataframe from which to get the vectors. You can save the model as an object.

```r
mod_a <- lm(wt ~ ht, data = nepali)
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where:

- Y_i : weight of child i
- $X_{1,i}$: height of child i

Model objects

The output from fitting a model using `lm` is a list object:

```
class(mod_a)
```

```
## [1] "lm"
```

This list object has a lot of different information from the model, including overall model summaries, estimated coefficients, fitted values, residuals, etc.

```
names(mod_a)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.resi
##  [9] "na.action"     "xlevels"       "call"          "terms"
## [13] "model"
```

Model objects and broom

This list object is not in a “tidy” format. However, there is a package named broom that you can use to pull “tidy” dataframes from this model object.

For example, you can use the glance function to pull out a tidy dataframe with model summaries.

```
library(broom)
glance(mod_a)
```

```
##   r.squared adj.r.squared     sigma statistic       p.value d
## 1 0.9161429     0.9156846 0.9016978  1999.283 1.962647e-100
##        AIC      BIC deviance df.residual
## 1 490.7103 500.3714 148.7898         183
```

Model objects and `broom`

If you want to get the estimated model coefficients (and some related summaries) instead, you can use the `tidy` function to do that:

```r
tidy(mod_a)
```

```
##          term  estimate   std.error statistic       p.value
## 1 (Intercept) -8.694768 0.427397843 -20.34350  7.424640e-49
## 2          ht  0.235050 0.005256822  44.71334 1.962647e-100
```

This output includes, for each model term, the **estimated coefficient** (`estimate`), its **standard error** (`std.error`), the **test statistic** (for `lm` output, the statistic for a test with the null hypothesis that the model coefficient is zero), and the associated **p-value** for that test (`p.value`).

Model objects and broom

Some of the model output have a value for each original observation (e.g., fitted values, residuals). You can use the augment function to add those elements to the original data used to fit the model (note: at the moment, you might get a warning message if you run this– it looks fairly benign and I imagine will be fixed in later versions of the package):

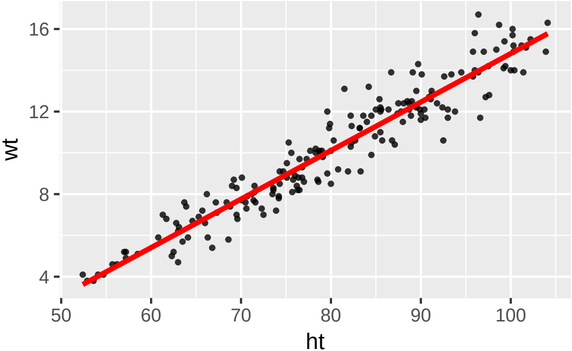
```
augment(mod_a) %>% slice(1:2) %>% select(1:6)
```

```
## # A tibble: 2 x 6
##   .rownames    wt    ht  .fitted    .se.fit      .resid
##       <chr> <dbl> <dbl>    <dbl>      <dbl>       <dbl>
## 1         1  12.8  91.2 12.74180 0.08755808  0.05820415
## 2         2  14.9 103.9 15.72693 0.14057265 -0.82693141
```

Model objects and broom

One important use of this augment output is to create a plot with both the original data and a line showing the fit model (via the predictions):

```
augment(mod_a) %>%
  ggplot(aes(x = ht, y = wt)) +
  geom_point(size = 0.8, alpha = 0.8) +
  geom_line(aes(y = .fitted), color = "red", size = 1.2)
```



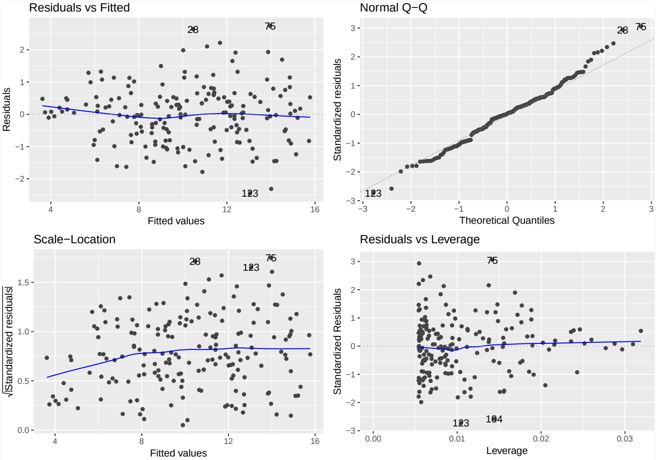
Model objects and `autoplot`

There is a function called `autoplot` in the `ggplot2` package that will check the class of an object and then create a certain default plot for that class. Although the generic `autoplot` function is in the `ggplot2` package, for `lm` and `glm` objects, you must have the `ggfortify` package installed and loaded to be able to access the methods of `autoplot` specifically for these object types.

If you have the package that includes an `autoplot` method for a specific object type, you can just run `autoplot` on the objects name and get a plot that is considered a useful default for that object type. For `lm` objects, `autoplot` gives small graphics with model diagnostic plots.

Model objects and `autoplot`

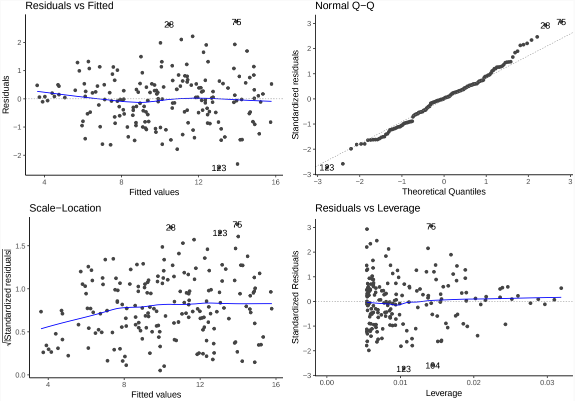
```
library(ggfortify)
autoplot(mod_a)
```



Model objects and autoplot

The output from autoplot is a ggplot object, so you can add elements to it as you would with other ggplot objects:

```
autoplot(mod_a) +
  theme_classic()
```



Model objects and base R “method” functions

Here, I’ve focused on a “tidy” approach to working with the output from fitting an `lm` model. However, you are certain to come across base R functions to work with this output in a similar way. Some base R functions you can use on model objects:

Function	Description
`summary`	Get a variety of information on the model, including coefficients and p-values for the coefficients
`coef`	Pull out just the coefficients for a model
`fitted`	Get the fitted values from the model (for the data used to fit the model)
`plot`	Create plots to help assess model assumptions
`residuals`	Get the model residuals

Examples of using a model object

One base R function you should definitely know is the summary function. The summary function gives you a lot of information about the model:

```
summary(mod_a)
```

(see next slide)

```
##
## Call:
## lm(formula = wt ~ ht, data = nepali)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.44736 -0.55708  0.01925  0.49941  2.73594
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -8.694768   0.427398  -20.34   <2e-16 ***
## ht           0.235050   0.005257   44.71   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
##
## Residual standard error: 0.9017 on 183 degrees of freedom
##   (15 observations deleted due to missingness)
## Multiple R-squared:  0.9161, Adjusted R-squared:  0.9157
## F-statistic:  1999 on 1 and 183 DF,  p-value: < 2.2e-16
```

In-course exercise

We’ll take a break now to do part of the In-Course Exercise (Section 7.6.1).

Fitting a model with a factor

You can also use binary variables or factors as independent variables in regression models:

```
mod_b <- lm(wt ~ sex, data = nepali)
tidy(mod_b)
```

```
##          term  estimate std.error statistic      p.value
## 1 (Intercept) 10.497980 0.3110957 33.745177 1.704550e-80
## 2   sexFemale -0.674724 0.4562792 -1.478752 1.409257e-01
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where $X_{1,i}$: sex of child i, where $0 =$ male; $1 =$ female

Linear models versus GLMs

You can fit a variety of models, including linear models, logistic models, and Poisson models, using generalized linear models (GLMs).

For linear models, the only difference between `lm` and `glm` is how they're fitting the model (least squares versus maximum likelihood). You should get the same results regardless of which you pick.

Linear models versus GLMs

For example:

```
mod_c <- glm(wt ~ ht, data = nepali)
tidy(mod_c)
```

```
##          term  estimate   std.error statistic       p.value
## 1 (Intercept) -8.694768 0.427397843 -20.34350  7.424640e-49
## 2          ht  0.235050 0.005256822  44.71334 1.962647e-100
```

```
tidy(mod_a)
```

```
##          term  estimate   std.error statistic       p.value
## 1 (Intercept) -8.694768 0.427397843 -20.34350  7.424640e-49
## 2          ht  0.235050 0.005256822  44.71334 1.962647e-100
```

GLMs

You can fit other model types with `glm()` using the `family` option:

Model type	`family` option
Linear	`family = gaussian(link = 'identity')`
Logistic	`family = binomial(link = 'logit')`
Poisson	`family = poisson(link = 'log')`

Logistic example

For example, say we wanted to fit a logistic regression for the nepali data of whether the probability that a child weighs more than 13 kg is associated with the child's height.

First, create a binary variable for wt_over_13:

```
nepali <- nepali %>%
  mutate(wt_over_13 = wt > 13)
head(nepali)
```

```
##       id    sex   wt    ht age wt_over_13
## 1 120011   Male 12.8  91.2  41      FALSE
## 2 120012 Female 14.9 103.9  57       TRUE
## 3 120021 Female  7.7  70.1   8      FALSE
## 4 120022 Female 12.1  86.4  35      FALSE
## 5 120023   Male 14.2  99.4  49       TRUE
## 6 120031   Male 13.9  96.4  46       TRUE
```

Logistic example

Now you can fit a logistic regression:

```
mod_d <- glm(wt_over_13 ~ ht, data = nepali,
             family = binomial(link = "logit"))
tidy(mod_d)
```

```
##          term    estimate  std.error statistic      p.value
## 1 (Intercept) -32.7016520 5.85196755 -5.588147 2.295060e-08
## 2          ht   0.3495227 0.06331892  5.520036 3.389307e-08
```

Here, the model coefficient gives the **log odds** of having a weight higher than 13 kg associated with a unit increase in height.

Formula structure

There are some conventions that can be used in R formulas. Common ones include:

Convention	Meaning
`I()`	calculate the value inside before fitting (e.g., `I(x1 + x2)`)
`:`	fit the interaction between two variables (e.g., `x1:x2`)
`*`	fit the main effects and interaction for both variables (e.g., `x1*x2` equals `x1 + x2 + x1:x2`)
`.`	fit all variables other than the response (e.g., `y ~ .`)
`-`	do not include a variable (e.g., `y ~ . - x1`)
`1`	intercept (e.g., `y ~ 1`)

To find out more

Great resources to find out more about using R for basic statistics:

- Statistical Analysis with R for Dummies, Joseph Schmuller (free online through our library; Chapter 14 covers regression modeling)
- The R Book, Michael J. Crawley (free online through our library; Chapter 14 covers regression modeling, Chapters 10 and 13 cover linear and generalized linear regression modeling)
- R for Data Science (Section 4)

If you want all the details about fitting linear models and GLMs in R, Faraway's books are fantastic (more at level of Master's in Applied Statistics):

- Linear Models with R, Julian Faraway (also freely available online through our library)
- Extending the Linear Model with R, Julian Faraway (available in hardcopy through our library)

In-course exercise

We'll take a break now to do part of the In-Course Exercise (Section 7.6.2).

Functions

Functions

As you move to larger projects, you will find yourself using the same code a lot.

Examples include:

- Reading in data from a specific type of equipment (air pollution monitor, accelerometer)
- Running a specific type of analysis (e.g., fitting the same model format to many datasets)
- Creating a specific type of plot or map

If you find yourself cutting and pasting a lot, convert the code to a function.

Functions

Advantages of writing functions include:

- Coding is more efficient
- Easier to change your code (if you've cut and paste code and you want to change something, you have to change it everywhere)
- Easier to share code with others

Functions

You can name a function anything you want, as long as you follow the naming rules for all R objects (although try to avoid names of preexisting-existing functions). You then specify any inputs (arguments; separate multiple arguments with commas) and put the code to run in braces. You **define** a function as an R object just like you do with other R objects (<-).

Here is the basic structure of “where things go” in an R function definition.

```
## Note: this code will not run
[function name] <- function([any arguments]){
        [code to run]
}
```

Functions

Here is an example of a very basic function. This function takes a number as input and adds 1 to that number. An R function will only return one R object. By default, that object will be the last line of code in the function body.

```
add_one <- function(number){
        number + 1 # Value returned by the function
}

add_one(number = 1:3)
```

```
## [1] 2 3 4
```

```
add_one(number = -1)
```

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

Functions

```
add_one <- function(number){
        number + 1 # Value returned by the function
}
```

- I picked the name of the function (`add_one`) (just like you pick what name you want to use with any R object)
- The only input is a numeric vector. I pick the name I want to use for the vector that is input to the function. I picked `number`.
- Within the code inside the function, the `number` refers to the numeric vector object that the user passed into the function.

Functions

As another example, you could write a small function to fit a specific model to a dataframe you input and return the model object:

```
fit_ht_wt_mod <- function(df){
  lm(wt ~ ht + sex, data = df) # Returns result from this call
}
```

- I picked the name of the function (`fit_ht_wt_mod`) (just like you pick what name you want to use with any R object)
- The only input is a dataframe. I pick the name I want to use for the dataframe that is input to the function. I picked `df` (I often use this as a default parameter name for a dataframe).
- Within the code inside the function, the `df` refers to the dataframe object that the user passed into the function.

Functions

Now you can apply that function within a tidy pipeline, for example to fit the model to a specific subset of the data (all children with an age over 12 months):

```
nepali %>%
  filter(age > 12) %>%
  fit_ht_wt_mod() %>%
  tidy()
```

```
##          term   estimate   std.error  statistic      p.value
## 1 (Intercept) -9.4434471 0.719494354 -13.125116 3.249112e-26
## 2          ht  0.2451880 0.008358455  29.334128 6.813073e-62
## 3   sexFemale -0.3473839 0.157770633  -2.201829 2.930270e-02
```

Functions

- Functions can input any type of R object (for example, vectors, data frames, even other functions and ggplot objects)
- Similarly, functions can output any type of R object
- However, functions can only output one R object. If you have complex things you want to output, a list might be a good choice for the output object type.
- Functions can have "side effects". Examples include printing something or drawing a plot. Any action that a function takes *besides returning an R object* is a "side effect".

Functions– parameter defaults

When defining a function, you can set default values for some of the parameters. For example, in the `add_one` function, you can set the default value of the `number` input to `0`.

```r
add_one <- function(number = 0){
        number + 1 # Value returned by the function
}
```

Now, if someone runs the function without providing a value for `number`, the function will use `0`. If they do provide a value for `number`, the function will use that instead.

```r
add_one()    # Uses 0 for `number`
```

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

```r
add_one(number = 3:5)   # Uses 5 for `number`
```

```
## [1] 4 5 6
```

Functions– parameters

You could write a function with no parameters:

```r
hello_world <- function(){
  print("Hello world!")
}

hello_world()
```

```
## [1] "Hello world!"
```

However, this will be pretty uncommon as you're first learning to write functions.

Functions– parameters

You can include multiple parameters, some with defaults and some without. For example, you could write a function that inputs two numbers and adds them. If you don't include a second value, 0 will be added as the second number:

```r
add_two_numbers <- function(first_number, second_number = 0){
  first_number + second_number
}
```

```r
add_two_numbers(first_number = 5:7, second_number = 0:2)
```

```
## [1] 5 7 9
```

```r
add_two_numbers(first_number = 5:7)
```

```
## [1] 5 6 7
```

Functions– the `return` function

You can explicitly specify the value to return from the function (use `return` function).

```r
add_one <- function(number = 0){
        new_number <- number + 1
        return(new_number)
}
```

If using `return` helps you think about what's happening with the code in your function, you can use it. However, outside of a few exceptions, you usually won't need to do it.

Functions– Error checking

There are ways to check for errors in the arguments a user inputs to the function. One useful check is to see if user inputs are in the required class.

The `assertive` package has some functions that you can use for common checks of the inputs to a function. If someone inputs something of the wrong class, it will give a useful error message. For example, the `assert_is_numeric` function will check if an object is in a numeric class. If so, it will do nothing. If not, it will return an error message:

```r
library(assertive)
assert_is_numeric(1:3)
assert_is_numeric(c("a", "b"))
```

```
## Error in eval(expr, envir, enclos): is_numeric : c("a", "b")
```

Functions– Error checking

You could add this in the code for the add_one function, so a useful error message will be returned if a user tries to input something besides a numeric vector for number:

```
add_one <- function(number){
  assert_is_numeric(number)
  number + 1
}
```

```
add_one(number = 1:3)
```

```
## [1] 2 3 4
```

```
add_one(number = c("a", "b"))
```

```
## Error in add_one(number = c("a", "b")): is_numeric : number i
```

Functions– Error checking

I would recommend that you not worry about this too much when you're learning to write functions for your own use.

However, once you have mastered the basics of writing functions and start writing them for others to use, you'll want to start incorporating this.

if / else

In R, the if statement evaluates everything in the parentheses and, if that evaluates to TRUE, runs everything in the braces. This means that you can trigger code in an if statement with a single-value logical vector:

```r
tell_date <- function(){
  cat("Today's date is: ")
  cat(format(Sys.time(), "%b %d, %Y"))

  todays_wday <- lubridate::wday(Sys.time(),
                                 label = TRUE)
  if(todays_wday %in% c("Sat", "Sun")){
    cat("\n")
    cat("It's the weekend!")
  }
}
```

if / else

```r
tell_date()
```

```
## Today's date is: Oct 04, 2017
```

`if` / `else`

You can add `else if` and `else` statements to tell R what to do if the condition in the `if` statement isn't met.

For example, in the `tell_date` function, we might want to add some code so it will print `"It's almost the weekend!"` on Fridays and how many days until Saturday on other weekdays.

if / else

```r
tell_date <- function(){
  # Print out today's date
  cat("Today's date is: ")
  cat(format(Sys.time(), "%b %d, %Y."), "\n")

  # Add something based on the weekday of today's date
  todays_wday <- lubridate::wday(Sys.time())

  if(todays_wday %in% c(1, 7)){      # What to do on Sat / Sun
    cat("It's the weekend!")
  } else if (todays_wday == c(6)) {  # What to do on Friday
    cat("It's almost the weekend!")
  } else {                           # What to do other days
    cat("It's ", 7 - todays_wday, "days until the weekend.")
  }
}
```

if / else

```
tell_date()
```

```
## Today's date is: Oct 04, 2017.
## It's  3 days until the weekend.
```

Functional programming

Applying a function repeatedly

One way that functions are really useful is that you can use the map family of functions from the purrr package to apply that function to all elements in a vector of a list (remember, a list could hold lots of similar dataframes).

For example, you could use map to apply the add_one function separately to 1, 2, and 3 by using map on a vector with those values:

```r
library(purrr)
my_list <- list(a = 1:2, b = 3:5)
map(my_list, add_one)
```

```
## $a
## [1] 2 3
##
## $b
## [1] 4 5 6
```

Applying a function repeatedly

This can also be very useful if you have a dataframe for which you would like to apply the same function to subsets of the data. For example, for the `nepali` data you may want to apply the model of weight regressed on height and sex separately for children 12 months and younger versus older children.

First, we can add a factor variable that specifies whether the child is younger or older than 12 months:

```r
nepali <- nepali %>%
  mutate(young = age < 12,
         young = factor(young, levels = c(TRUE, FALSE),
                        labels = c("younger", "older")))
```

Applying a function repeatedly

```r
nepali %>%
  slice(1:3)
```

```
## # A tibble: 3 x 7
##       id    sex    wt    ht   age wt_over_13   young
##   <fctr> <fctr> <dbl> <dbl> <int>      <lgl>  <fctr>
## 1 120011   Male  12.8  91.2    41      FALSE   older
## 2 120012 Female  14.9 103.9    57       TRUE   older
## 3 120021 Female   7.7  70.1     8      FALSE younger
```

Applying a function repeatedly

Then, you can use the `nest` function to “nest” a dataframe by a factor variable. This function will create a column that actually stores its own dataframe:

```r
nested_nepali <- nepali %>%
  group_by(young) %>%
  nest()
nested_nepali
```

```
## # A tibble: 2 x 2
##     young               data
##    <fctr>             <list>
## 1   older <tibble [162 x 6]>
## 2 younger  <tibble [38 x 6]>
```

Applying a function repeatedly

Each element of the data column in the nested dataset is actually a full dataframe:

```
nested_nepali$data[[1]] %>% slice(1:3)
```

```
## # A tibble: 3 x 6
##       id    sex    wt    ht   age wt_over_13
##   <fctr> <fctr> <dbl> <dbl> <int>      <lgl>
## 1 120011   Male  12.8  91.2    41      FALSE
## 2 120012 Female  14.9 103.9    57       TRUE
## 3 120022 Female  12.1  86.4    35      FALSE
```

Applying a function repeatedly

Now, you can use map to apply the modeling function to each of these subsets of the dataframe. Use mutate to add a column with the results. We can also add a column with the results of applying augment to each of the model results:

```
modeled_nepali <- nepali %>%
  group_by(young) %>%
  nest() %>%
  mutate(mod_results = map(data, fit_ht_wt_mod),
         augmented_data = map(mod_results, augment))
modeled_nepali
```

```
## # A tibble: 2 x 4
##     young                 data mod_results          augmented_data
##    <fctr>               <list>      <list>                  <list>
## 1   older <tibble [162 x 6]>    <S3: lm> <data.frame [151 x 11]>
## 2 younger  <tibble [38 x 6]>    <S3: lm>  <data.frame [34 x 11]>
```

Applying a function repeatedly

Each element of the `mod_results` column is the output from an `lm` model:

```
modeled_nepali$mod_results[[1]]
```

```
##
## Call:
## lm(formula = wt ~ ht + sex, data = df)
##
## Coefficients:
## (Intercept)           ht    sexFemale
##     -9.4638       0.2452      -0.3095
```

Applying a function repeatedly

We can apply `tidy` and `glance` to this output, just like we did with the output from fitting a single model:

```
tidy(modeled_nepali$mod_results[[1]])
```

```
##          term   estimate   std.error  statistic      p.value
## 1 (Intercept) -9.4637940 0.662450802 -14.286033 1.200446e-29
## 2          ht  0.2452427 0.007707689  31.817924 4.615549e-68
## 3   sexFemale -0.3095451 0.152599469  -2.028481 4.430458e-02
```

Applying a function repeatedly

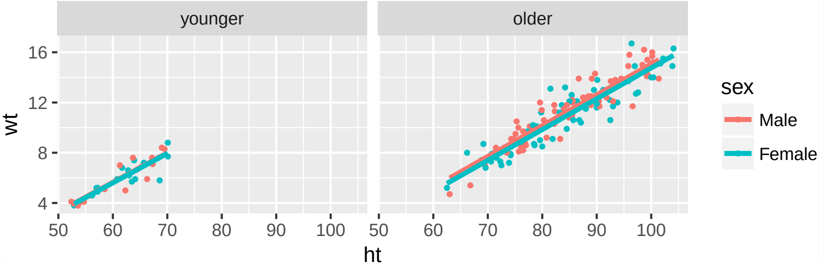
To get back to a regular dataframe, you can “unnest”. Before you do this, you should limit the data to only columns that will “unnest” to the same number of rows (if you have multiple columns with dataframes or lists in them). For example, we could unnest the augmented_data column, so we can plot observed data versus the model fit:

```
modeled_nepali %>%
  select(young, augmented_data) %>%
  unnest() %>%
  slice(1:3) %>% select(1:7)
```

```
## # A tibble: 3 x 7
##    young .rownames    wt    ht    sex  .fitted   .se.fit
##   <fctr>     <chr> <dbl> <dbl> <fctr>    <dbl>     <dbl>
## 1  older         1  12.8  91.2   Male 12.90234 0.1131341
## 2  older         2  14.9 103.9 Female 15.70737 0.1916600
## 3  older         3  12.1  86.4 Female 11.41563 0.1147084
```

Applying a function repeatedly

```
modeled_nepali %>%
  select(young, augmented_data) %>%
  unnest() %>%
  ggplot(aes(x = ht, y = wt, group = sex, color = sex)) +
  geom_point(size = 0.8) +
  geom_line(aes(y = .fitted), size = 1.2) +
  facet_wrap(~ young, ncol = 2)
```



Applying a function repeatedly

The full pipe for creating this figure is:

```r
nepali %>%
  group_by(young) %>%    # Group by young and nest
  nest() %>%             # to model each group of "young"
  mutate(mod_results = map(data, fit_ht_wt_mod),
         augmented_data = map(mod_results, augment)) %>%
  select(young, augmented_data) %>%
  unnest %>%             # Unnest `augmented_data` (nested)
  ggplot(aes(x = ht, y = wt, group = sex, color = sex)) +
  geom_point(size = 0.8) +
  geom_line(aes(y = .fitted), size = 1.2) +
  facet_wrap(~ young, ncol = 2)
```

Regular expressions

Regular expressions

For these examples, we'll use some data on passengers of the Titanic. You can load this data using:

```r
# install.packages("titanic")
library(titanic)
data("titanic_train")
```

We will be using the stringr package:

```r
library(stringr)
```

Regular expressions

This data includes a column called “Name” with passenger names. This column is somewhat messy and includes several elements that we might want to separate (last name, first name, title). Here are the first few values of “Name”:

```
titanic_train %>% select(Name) %>% slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                   Name
##                                                  <chr>
## 1                              Braund, Mr. Owen Harris
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 3                               Heikkinen, Miss. Laina
```

Regular expressions

We’ve already done some things to manipulate strings. For example, if we wanted to separate “Name” into last name and first name (including title), we could actually do that with the separate function:

```
titanic_train %>%
  select(Name) %>%
  slice(1:3) %>%
  separate(Name, c("last_name", "first_name"), sep = ", ")
```

```
## # A tibble: 3 x 2
##   last_name                                 first_name
## *     <chr>                                      <chr>
## 1    Braund                            Mr. Owen Harris
## 2   Cumings Mrs. John Bradley (Florence Briggs Thayer)
## 3 Heikkinen                                Miss. Laina
```

Regular expressions

Notice that `separate` is looking for a regular pattern (“,”) and then doing something based on the location of that pattern in each string (splitting the string).

There are a variety of functions in R that can perform manipulations based on finding regular patterns in character strings.

Regular expressions

The `str_detect` function will look through each element of a character vector for a designated pattern. If the pattern is there, it will return `TRUE`, and otherwise `FALSE`. The convention is:

```
## Generic code
str_detect(string = [vector you want to check],
           pattern = [pattern you want to check for])
```

For example, to create a logical vector specifying which of the Titanic passenger names include “Mrs.”, you can call:

```r
mrs <- str_detect(titanic_train$Name, "Mrs.")
head(mrs)
```

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

Regular expressions

The result is a logical vector, so `str_detect` can be used in `filter` to subset data to only rows where the passenger's name includes “Mrs.”:

```
titanic_train %>%
  filter(str_detect(Name, "Mrs.")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                   Name
##                                                  <chr>
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2        Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

There is an older, base R function called `grepl` that does something very similar (although note that the order of the arguments is reversed).

```
titanic_train %>%
  filter(grepl("Mrs.", Name)) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                  Name
##                                                 <chr>
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2        Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

The `str_extract` function can be used to extract a string (if it exists) from each value in a character vector. It follows similar conventions to `str_detect`:

```
## Generic code
str_extract(string = [vector you want to check],
            pattern = [pattern you want to check for])
```

Regular expressions

For example, you might want to extract “Mrs.” if it exists in a passenger’s name:

```
titanic_train %>%
  mutate(mrs = str_extract(Name, "Mrs.")) %>%
  select(Name, mrs) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                    Name   mrs
##                                                   <chr> <chr>
## 1                               Braund, Mr. Owen Harris  <NA>
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                                Heikkinen, Miss. Laina  <NA>
```

Notice that now we’re creating a new column (`mrs`) that either has “Mrs.” (if there’s a match) or is missing (`NA`) if there’s not a match.

Regular expressions

For this first example, we were looking for an exact string (“Mrs”). However, you can use patterns that match a particular pattern, but not an exact string. For example, we could expand the regular expression to find “Mr.” or “Mrs.”:

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr\\.|Mrs\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                   Name title
##                                                  <chr> <chr>
## 1                              Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                               Heikkinen, Miss. Laina  <NA>
```

Note that this pattern uses a special operator (|) to find one pattern **or**

Regular expressions

As a note, in regular expressions, all of the following characters are special characters that need to be escaped with backslashes if you want to use them literally:

```
. * + ^ ? $ \ | ( ) [ ] { }
```

Regular expressions

Notice that “Mr.” and “Mrs.” both start with “Mr”, end with “.”, and may or may not have an “s” in between.

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr(s)*\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                   Name title
##                                                  <chr> <chr>
## 1                              Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                               Heikkinen, Miss. Laina  <NA>
```

This pattern uses (s)* to match zero or more “s”s at this spot in the pattern.

Regular expressions

In the previous code, we found “Mr.” and “Mrs.”, but missed “Miss.”. We could tweak the pattern again to try to capture that, as well. For all three, we have the pattern that it starts with “M”, has some lowercase letters, and then ends with “.”.

```
titanic_train %>%
  mutate(title = str_extract(Name, "M[a-z]+\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                   Name title
##                                                  <chr> <chr>
## 1                            Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                             Heikkinen, Miss. Laina Miss.
```

Regular expressions

The last pattern used `[a-z]+` to match one or more lowercase letters. The `[a-z]`is a **character class**.

You can also match digits (`[0-9]`), uppercase letters (`[A-Z]`), just some letters (`[aeiou]`), etc.

You can negate a character class by starting it with `^`. For example, `[^0-9]` will match anything that **isn't** a digit.

Regular expressions

Sometimes, you want to match a pattern, but then only subset a part of it. For example, each passenger seems to have a title ("Mr.", "Mrs.", etc.) that comes after "," and before ".". We can use this pattern to find the title, but then we get some extra stuff with the match:

```
titanic_train %>%
  mutate(title = str_extract(Name, ",\\s[A-Za-z]*\\.\\s")) %>%
  select(title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##       title
##       <chr>
## 1   , Mr.
## 2  , Mrs.
## 3 , Miss.
```

As a note, in this pattern, `\\s` is used to match a space.

Regular expressions

We are getting things like ", Mr.", when we really want "Mr". We can use the `str_match` function to do this. We group what we want to extract from the pattern in parentheses, and then the function returns a matrix. The first column is the full pattern match, and each following column gives just what matches within the groups.

```r
head(str_match(titanic_train$Name,
         pattern = ",\\s([A-Za-z]*)\\.\\s"))
```

```
##      [,1]       [,2]  
## [1,] ", Mr. "   "Mr"  
## [2,] ", Mrs. "  "Mrs" 
## [3,] ", Miss. " "Miss"
## [4,] ", Mrs. "  "Mrs" 
## [5,] ", Mr. "   "Mr"  
## [6,] ", Mr. "   "Mr"  
```

Regular expressions

To get just the title, then, we can run:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[ , 2]) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                   Name title
##                                                  <chr> <chr>
## 1                              Braund, Mr. Owen Harris    Mr
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs
## 3                               Heikkinen, Miss. Laina  Miss
```

The `[ , 2]` pulls out just the second column from the matrix returned by `str_match`.

Regular expressions

Here are some of the most common titles:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[ , 2]) %>%
  group_by(title) %>% summarize(n = n()) %>%
  arrange(desc(n)) %>% slice(1:5)
```

```
## # A tibble: 5 x 2
##    title     n
##    <chr> <int>
## 1     Mr   517
## 2   Miss   182
## 3    Mrs   125
## 4 Master    40
## 5     Dr     7
```

Regular expressions

The following slides have a few other examples of regular expressions in action with this dataset.

Get just names that start with ("^") the letter "A":

```
titanic_train %>%
  filter(str_detect(Name, "^A")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                         Name
##                                                        <chr>
## 1                                   Allen, Mr. William Henry
## 2                                Andersson, Mr. Anders Johan
## 3 Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)
```

Regular expressions

Get names with “II” or “III” ({2,} says to match at least two times):

```r
titanic_train %>%
  filter(str_detect(Name, "I{2,}")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 2 x 1
##                                    Name
##                                   <chr>
## 1  Carter, Master. William Thornton II
## 2 Roebling, Mr. Washington Augustus II
```

Regular expressions

Get names with “Andersen” or “Anderson” (alternatives in square brackets):

```
titanic_train %>%
  filter(str_detect(Name, "Anders[eo]n")) %>%
  select(Name)
```

```
##                                               Name
## 1 Andersen-Jensen, Miss. Carla Christine Nielsine
## 2                             Anderson, Mr. Harry
## 3                    Walker, Mr. William Anderson
## 4                     Olsvigen, Mr. Thor Anderson
## 5      Soholt, Mr. Peter Andreas Lauritz Andersen
```

Regular expressions

Get names that start with (“^” outside of brackets) the letters “A” and “B”:

```
titanic_train %>%
  filter(str_detect(Name, "^[AB]")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                        Name
##                       <chr>
## 1  Braund, Mr. Owen Harris
## 2 Allen, Mr. William Henry
## 3 Bonnell, Miss. Elizabeth
```

Regular expressions

Get names that end with (“$”) the letter “b” (either lowercase or uppercase):

```r
titanic_train %>%
  filter(str_detect(Name, "[bB]$")) %>%
  select(Name)
```

```
##                         Name
## 1   Emir, Mr. Farred Chehab
## 2 Goldschmidt, Mr. George B
## 3            Cook, Mr. Jacob
## 4           Pasic, Mr. Jakob
```

Regular expression

Some useful regular expression operators include:

Operator	Meaning
.	Any character
	Match 0 or more times (greedy)
?	Match 0 or more times (non-greedy)
+	Match 1 or more times (greedy)
+?	Match 1 or more times (non-greedy)
ˆ	Starts with (in brackets, negates)
$	Ends with
[. . .]	Character classes

Regular expressions

For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Introduction to stringr by Hadley Wickham
- Handling and Processing Strings in R by Gaston Sanchez (seven chapter ebook)
- `http://gskinner.com/RegExr` and `http://www.txt2re.com`: Interactive tools for helping you build regular expression pattern strings