Exploring data 2

Simple statistical tests in R

Example: Probability of fatal crashes in Las Vegas

Let's pull the fatal accident data just for the county that includes Las Vegas, NV.

Each US county has a unique identifier (FIPS code), composed of a two-digit state FIPS and a three-digit county FIPS code. The state FIPS for Nevada is 32; the county FIPS for Clark County is 003.

Example: Probability of fatal crashes in Las Vegas

Therefore, we can filter down to Clark County data in the FARS data we collected with the following code:

```r
library(readr)
library(dplyr)
clark_co_accidents <- read_csv("../data/accident.csv") %>%
  filter(STATE == 32 & COUNTY == 3)
```

We can also check the number of accidents:

```r
clark_co_accidents %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   201
```

Example: Probability of fatal crashes in Las Vegas

We want to test if the probability, on a Friday or Saturday, of a fatal accident occurring is higher than on other days of the week. Let's clean the data up a bit as a start:

```r
library(tidyr)
library(lubridate)
clark_co_accidents <- clark_co_accidents %>%
  select(DAY, MONTH, YEAR) %>%
  unite(date, DAY, MONTH, YEAR, sep = "-") %>%
  mutate(date = dmy(date))
```

Example: Probability of fatal crashes in Las Vegas

Here's what the data looks like now:

```r
clark_co_accidents %>%
  slice(1:5)
```

```
## # A tibble: 5 x 1
##   date
##   <date>
## 1 2016-01-10
## 2 2016-02-21
## 3 2016-01-06
## 4 2016-01-13
## 5 2016-02-18
```

Example: Probability of fatal crashes in Las Vegas

Next, let's get the count of accidents by date:

```
clark_co_accidents <- clark_co_accidents %>%
  group_by(date) %>%
  count() %>%
  ungroup()
clark_co_accidents %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##   date           n
##   <date>     <int>
## 1 2016-01-03     1
## 2 2016-01-06     1
## 3 2016-01-09     3
```

Example: Probability of fatal crashes in Las Vegas

We're missing the dates without a fatal crash, so let's add those. First, create a dataframe with all dates in 2016:

```r
all_dates <- data_frame(date = seq(ymd("2016-01-01"),
                                    ymd("2016-12-31"), by = 1))
all_dates %>%
  slice(1:5)
```

```
## # A tibble: 5 x 1
##   date
##   <date>
## 1 2016-01-01
## 2 2016-01-02
## 3 2016-01-03
## 4 2016-01-04
## 5 2016-01-05
```

Example: Probability of fatal crashes in Las Vegas

Then merge this with the original dataset on Las Vegas fatal crashes and make any day missing from the fatal crashes dataset have a "0" for number of fatal accidents (`n`):

```r
clark_co_accidents <- clark_co_accidents %>%
  right_join(all_dates, by = "date") %>%
  # If `n` is missing, set to 0. Otherwise keep value.
  mutate(n = ifelse(is.na(n), 0, n))
clark_co_accidents %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##   date           n
##   <date>     <dbl>
## 1 2016-01-01    0.
## 2 2016-01-02    0.
## 3 2016-01-03    1.
```

Example: Probability of fatal crashes in Las Vegas

Next, let's add some information about day of week and weekend:

```
clark_co_accidents <- clark_co_accidents %>%
  mutate(weekday = wday(date, label = TRUE),
         weekend = weekday %in% c("Fri", "Sat"))
clark_co_accidents %>%
  slice(1:3)
```

```
## # A tibble: 3 x 4
##   date           n weekday weekend
##   <date>     <dbl> <ord>   <lgl>
## 1 2016-01-01    0. Fri     TRUE
## 2 2016-01-02    0. Sat     TRUE
## 3 2016-01-03    1. Sun     FALSE
```

Example: Probability of fatal crashes in Las Vegas

Now let's calculate the probability that a day has at least one fatal crash, separately for weekends and weekdays:

```
clark_co_accidents <- clark_co_accidents %>%
  mutate(any_crash = n > 0)
crash_prob <- clark_co_accidents %>%
  group_by(weekend) %>%
  summarize(n_days = n(),
            crash_days = sum(any_crash)) %>%
  mutate(prob_crash_day = crash_days / n_days)
crash_prob
```

```
## # A tibble: 2 x 4
##   weekend n_days crash_days prob_crash_day
##   <lgl>    <int>      <int>          <dbl>
## 1 FALSE      260        107          0.412
## 2 TRUE       106         43          0.406
```

Example: Probability of fatal crashes in Las Vegas

In R, you can use `prop.test` to test if two proportions are equal. Inputs include the total number of trials in each group (`n =`) and the number of "successes"" (`x =`):

```r
prop.test(x = crash_prob$crash_days,
          n = crash_prob$n_days)
```

```
##
##  2-sample test for equality of proportions with continuity
##  correction
##
## data:  crash_prob$crash_days out of crash_prob$n_days
## X-squared = 1.5978e-30, df = 1, p-value = 1
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##  -0.1109757  0.1227318
## sample estimates:
##    prop 1    prop 2
## 0.4115385 0.4056604
```

Find out more about statistical tests in R

I won't be teaching in this course how to find the correct statistical test. That's something you'll hopefully learn in a statistics course.

There are also a variety of books that can help you with this, including some that you can access free online through CSU's library. One servicable introduction is "Statistical Analysis with R for Dummies".

In-course exercise

We'll take a break now to do the first part of the in-course exercise.

Output of statistical tests: List objects

You can create an object from the output of any statistical test in R. Typically, this will be (at least at some level) in an object class called a "list":

```r
vegas_test <- prop.test(x = crash_prob$crash_days,
                        n = crash_prob$n_days)
is.list(vegas_test)
```

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

Output of statistical tests: List objects

So far, we've mostly worked with two object types in R, **dataframes** and **vectors**.

In the next subsection we'll look more at two object classes we haven't looked at much, **matrices** and **lists**. Both have important roles once you start applying more advanced methods to analyze your data.

Matrices

Matrices

A matrix is like a data frame, but all the values in all columns must be of the same class (e.g., numeric, character). (Another way you can think of it is as a "wrapped" vector.)

Matrices can be faster and more memory-efficient than data frames. Also, a lot of statistical methods within R code is implemented using linear algebra and other mathematical techniques based on matrices.

Matrices

We can use the `matrix()` function to construct a matrix:

```
foo <- matrix(1:10, ncol = 5)
foo
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Matrices

The `as.matrix()` function is used to convert an object to a matrix:

```
foo <- data.frame(col_1 = 1:2, col_2 = 3:4,
                  col_3 = 5:6, col_4 = 7:8,
                  col_5 = 9:10)
foo <- as.matrix(foo)
foo
```

```
##      col_1 col_2 col_3 col_4 col_5
## [1,]     1     3     5     7     9
## [2,]     2     4     6     8    10
```

Matrices

You can index matrices with square brackets, just like data frames:

```
foo[1, 1:2]
```

```
## col_1 col_2
##     1     3
```

You cannot, however, use `dplyr` functions with matrices:

```
foo %>% filter(col_1 == 1)
```

```
Error in UseMethod("filter_") :
  no applicable method for 'filter_' applied to an object of
  class "c('matrix', 'integer', 'numeric')"
```

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] 2 1 9 8 7
##
## $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] 2 1 9 8 7
```

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

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

Indexing lists

To access a specific value within a list element we can index the element e.g.:

```r
example_list[[1]][[2]]
```

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

```r
example_list[["b"]][[2]]
```

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

Exploring lists

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

```
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:

```
names(example_list)
```

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

Exploring lists

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

```r
str(example_list)
```

```
## List of 2
##  $ a: int [1:5] 2 1 9 8 7
##  $ b:Classes 'tbl_df', 'tbl' and 'data.frame':   3 obs. of  2 variables:
##   ..$ letters: chr [1:3] "a" "b" "c"
##   ..$ number : int [1:3] 1 2 3
```

Exploring lists

A list can even contain other lists. We can use the `str` function to see the structure of a list:

```r
a_list <- list(list("a", "b"), list(1, 2))

str(a_list)
```

```
## List of 2
##  $ :List of 2
##   ..$ : chr "a"
##   ..$ : chr "b"
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
```

Exploring lists

Sometimes you'll see unnecessary lists-of-lists, perhaps when importing data into R created. Or a list with multiple elements that you would like to combine. You can remove a level of hierarchy from a list using the `flatten` function from the `purrr` package:

```
library(purrr)
a_list
```

```
## [[1]]
## [[1]][[1]]
## [1] "a"
##
## [[1]][[2]]
## [1] "b"
##
##
## [[2]]
## [[2]][[1]]
```

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
```

List object from statistical test

Let's look at the list object from the statistical test we ran for Las Vegas:

```r
str(vegas_test)
```

```
## List of 9
##  $ statistic  : Named num 1.6e-30
##   ..- attr(*, "names")= chr "X-squared"
##  $ parameter  : Named num 1
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 1
##  $ estimate   : Named num [1:2] 0.412 0.406
##   ..- attr(*, "names")= chr [1:2] "prop 1" "prop 2"
##  $ null.value : NULL
##  $ conf.int   : atomic [1:2] -0.111 0.123
##   ..- attr(*, "conf.level")= num 0.95
##  $ alternative: chr "two.sided"
##  $ method     : chr "2-sample test for equality of proportions with continui
##  $ data.name  : chr "crash_prob$crash_days out of crash_prob$n_days"
##  - attr(*, "class")= chr "htest"
```

List object from statistical test

We can pull out an element using the $ notation:

```
vegas_test$p.value
```

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

Or using the [[notation:

```
vegas_test[[4]]
```

```
##    prop 1    prop 2
## 0.4115385 0.4056604
```

Broom package

You may have noticed, though, that this output is not a tidy dataframe.

Ack! That means we can't use all the tidyverse tricks we've learned so far in the course!

Fortunately, David Robinson noticed this problem and came up with a package called `broom` that can "tidy up" a lot of these kinds of objects.

Broom package

The `broom` package has three main functions:

- `glance`: Return a one-row, tidy dataframe from a model or other R object
- `tidy`: Return a tidy dataframe from a model or other R object
- `augment`: "Augment" the dataframe you input to the statistical function

Broom package

Here is the output for `tidy` for the `vegas_test` object (`augment` won't work for this type of object, and `glance` gives the same thing as `tidy`):

```r
library(broom)
tidy(vegas_test)
```

```
##   estimate1 estimate2    statistic p.value parameter   conf.low conf.high
## 1 0.4115385 0.4056604 1.597806e-30       1         1 -0.1109757 0.1227318
##                                                                    method
## 1 2-sample test for equality of proportions with continuity correction
##   alternative
## 1   two.sided
```

Regression models

World Cup example

In the World Cup data, we may wonder if the number of tackles is associated with the time the player played. Let's start by grabbing just the variables we care about (we'll be using `Position` later, so we'll include that):

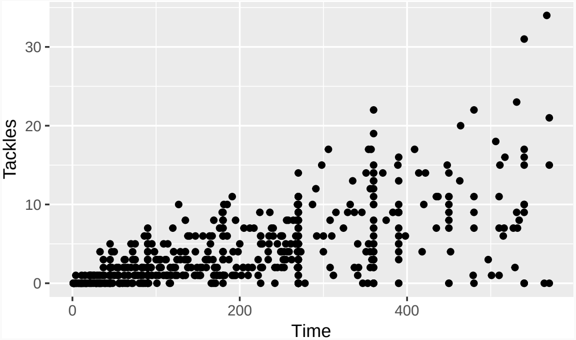
```
library(faraway)
data(worldcup)
worldcup <- worldcup %>%
  select(Time, Tackles, Position)
worldcup %>% slice(1:3)
```

```
## # A tibble: 3 x 3
##    Time Tackles Position
##   <int>   <int> <fct>
## 1    16       0 Midfielder
## 2   351      14 Midfielder
## 3   180       6 Defender
```

World Cup example

We can start by plotting the relationship between the time a player played and the number of tackles they had:

```
library(ggplot2)
ggplot(worldcup, aes(Time, Tackles)) +
  geom_point()
```



World Cup example

There does indeed seem to be an association. Next, we might want to test this using some kind of statistical model or test.

Let's start by fitting a linear regression model, to see if there's evidence that tackles tend to change (increase or decrease) as the player's time played increases.

(In a bit, we'll figure out that a linear model might not be the best way to model this, since the number of tackles is a count, rather than a variable with a normal distribution, but bear with me. . .)

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:

$$Y_i \sim X_1 + X_2 + \cdots + \epsilon_i$$

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
tackle_model <- lm(Tackles ~ Time, data = worldcup)
```

This call fits the model:

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

where:

- Y_i : Number of tackles for player i (dependent variable)
- $X_{1,i}$: Minutes played by player i (independent variable)

Linear models

A few things to point out:

- By default, an intercept is fit to the model.
- If you specify a dataframe using `data` in the `lm` call, you can write the model formula using just the column names for the independent variable(s) and dependent variable you want, without quotation marks around those names.
- You can save the output of fitting the model to an R object (if you don't, a summary of the fit model will be print out at the console).

Model objects

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

```
class(tackle_model)
```

```
## [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(tackle_model)
```

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

Model objects and broom

This list object is not in a "tidy" format. However, you can use functions from broom to pull "tidy" dataframes from this model object.

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

```
glance(tackle_model)
```

```
##   r.squared adj.r.squared    sigma statistic      p.value df    logLik
## 1 0.3729221     0.3718646 3.688568   352.656 4.305563e-62  2 -1619.884
##        AIC      BIC deviance df.residual
## 1 3245.767 3258.933 8068.084         593
```

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:

```
tidy(tackle_model)
```

```
##          term  estimate   std.error  statistic      p.value
## 1 (Intercept) 0.1099166 0.264779664  0.4151249 6.782006e-01
## 2        Time 0.0195423 0.001040639 18.7791373 4.305563e-62
```

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:

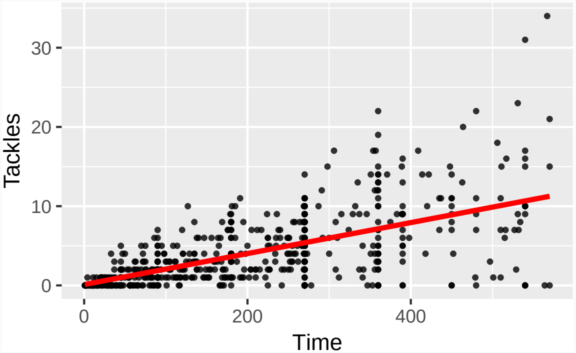
```
augment(tackle_model) %>%
  slice(1:2)
```

```
## # A tibble: 2 x 10
##   .rownames Tackles  Time .fitted .se.fit .resid    .hat .sigma   .cooksd
##   <chr>       <int> <int>   <dbl>   <dbl>  <dbl>   <dbl>  <dbl>     <dbl>
## 1 Abdoun          0    16   0.423   0.251 -0.423 0.00464   3.69 0.0000307
## 2 Abe            14   351   6.97    0.212  7.03  0.00329   3.68 0.00601
## # ... with 1 more variable: .std.resid <dbl>
```

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):

```r
augment(tackle_model) %>%
  ggplot(aes(x = Time, y = Tackles)) +
  geom_point(size = 0.8, alpha = 0.8) +
  geom_line(aes(y = .fitted), color = "red", size = 1.2)
```



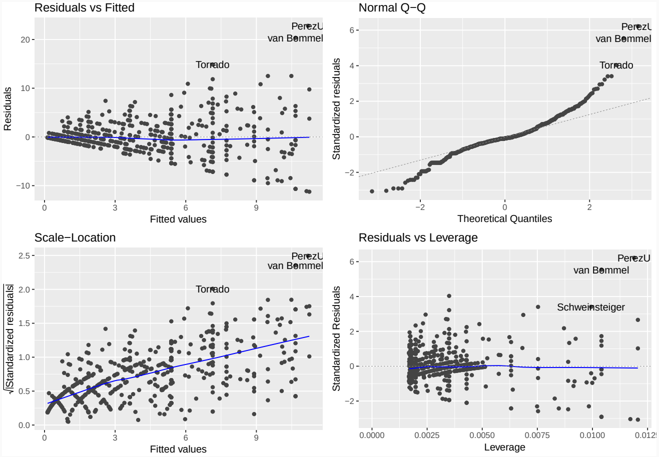
Model objects and `autoplot`

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`

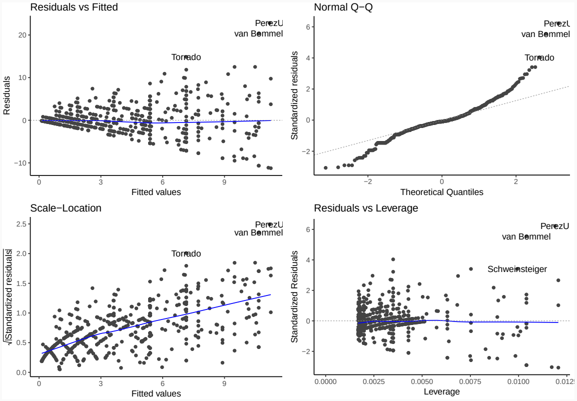
```r
library(ggfortify)
autoplot(tackle_model)
```



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(tackle_model) +
  theme_classic()
```

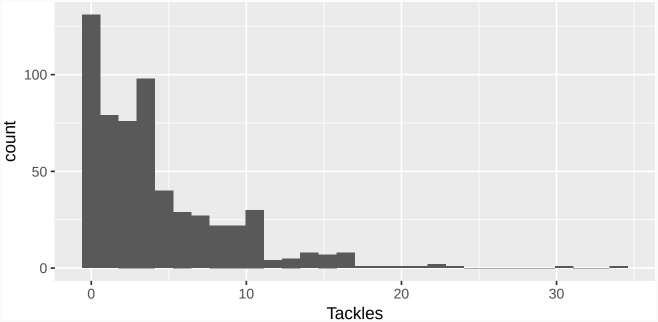


Regression models

In this case, these diagnostics clearly show that there are some problems with using a linear regression model to fit this data.

Many of these issues arise because the outcome (dependent) variable doesn't follow a normal distribution.

```r
ggplot(worldcup, aes(x = Tackles)) +
  geom_histogram()
```



Regression models

A better model, therefore, might be one where we assume that `Tackles` follows a Poisson distribution, rather than a normal distribution. (For variables that represent counts, this will often be the case.)

In the a little bit, we'll look at **generalized linear models**, which let us extend the idea of a linear model to situations where the dependent variable follows a distribution other than the normal distribution.

In-course exercise

We'll take a break now to do the second part of the In-Course Exercise.

Fitting a model with a factor

You can also use binary variables or factors (i.e., categorical variables) as independent variables in regression models:

```r
tackles_model_2 <- lm(Tackles ~ Position, data = worldcup)
```

This call fits the model:

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

where $X_{1,i}$: Position of player i

Fitting a model with a factor

If there are more than one levels to the factor, then the model will fit a separate value for each level of the factor above the first level (which will serve as a baseline):

```
levels(worldcup$Position)
```

```
## [1] "Defender"   "Forward"    "Goalkeeper" "Midfielder"
```

```
tidy(tackles_model_2)
```

```
##                  term   estimate std.error  statistic      p.value
## 1         (Intercept)  5.4627660 0.3153563 17.3225216 1.137808e-54
## 2    PositionForward -3.4417869 0.4797859 -7.1735895 2.195901e-12
## 3 PositionGoalkeeper -5.4349882 0.7866368 -6.9091459 1.265115e-11
## 4 PositionMidfielder -0.3004853 0.4259717 -0.7054113 4.808322e-01
```

Fitting a model with a factor

The intercept is the expected (average) value of the outcome (`Tackles`) for the first level of the factor. Each other estimate gives the expected difference between the value of the outcome for this first level of `Position` and one of the other levels of the factor.

```r
levels(worldcup$Position)
```

```
## [1] "Defender"   "Forward"    "Goalkeeper" "Midfielder"
```

```r
tidy(tackles_model_2)
```

```
##                 term   estimate std.error  statistic      p.value
## 1        (Intercept)  5.4627660 0.3153563 17.3225216 1.137808e-54
## 2    PositionForward -3.4417869 0.4797859 -7.1735895 2.195901e-12
## 3 PositionGoalkeeper -5.4349882 0.7866368 -6.9091459 1.265115e-11
## 4 PositionMidfielder -0.3004853 0.4259717 -0.7054113 4.808322e-01
```

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:

```r
glm(Tackles ~ Time, data = worldcup) %>%
  tidy()
```

```
##          term  estimate   std.error  statistic      p.value
## 1 (Intercept) 0.1099166 0.264779664  0.4151249 6.782006e-01
## 2        Time 0.0195423 0.001040639 18.7791373 4.305563e-62
```

```r
lm(Tackles ~ Time, data = worldcup) %>%
  tidy()
```

```
##          term  estimate   std.error  statistic      p.value
## 1 (Intercept) 0.1099166 0.264779664  0.4151249 6.782006e-01
## 2        Time 0.0195423 0.001040639 18.7791373 4.305563e-62
```

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')`

GLM example

For example, say we wanted to fit a GLM, but specifying a Poisson distribution for the outcome (and a log link) since we think that `Tackles` might be distributed with a Poisson distribution:

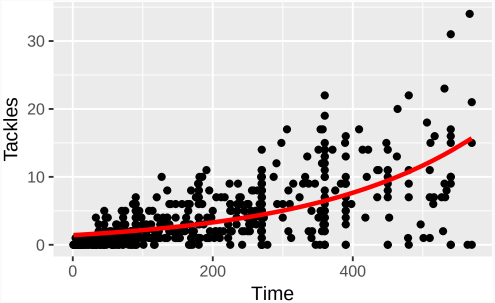
```
tackle_model_3 <- glm(Tackles ~ Time, data = worldcup,
                      family = poisson(link = "log"))
tackle_model_3 %>%
  tidy()
```

```
##          term    estimate    std.error statistic       p.value
## 1 (Intercept) 0.349969480 0.0443155157  7.897222  2.851874e-15
## 2        Time 0.004215125 0.0001286443 32.765727 1.812941e-235
```

GLM example

Here are the predicted values from this model (red line):

```
tackle_model_3 %>%
  augment() %>%
  mutate(.fitted = exp(.fitted)) %>%
  ggplot(aes(x = Time, y = Tackles)) +
  geom_point() +
  geom_line(aes(y = .fitted), color = "red", size = 1.2)
```



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 the third part of the In-Course Exercise.