Data Visualization

Getting Fancy: Working with Models

Ciara Zogheib

Data Sciences Institute, University of Toronto

Today, we will...

- Work through practice code from Chapter 6 (*Working with Models*) of Healy, K. (2018). Data Visualization: A Practical Introduction. Princeton University Press. This code will let us
  - Show several fitted models on the same plot
  - Understand the structure of model objects
  - Generate predictions as a data visualization
  - Tidy model objects with Broom
  - Perform grouped analysis using list columns

Models?????

- This class works with statistical models from a data visualization and ggplot standpoint
- If you need a refresher on fitting statistical models in R, try exploring: James, Witten, Hastie, Tibshirani. (2021). An Introduction to Statistical Learning: With Applications in R, 2nd Edition, Springer

Recall stat_ functions

- We learned that ggplot does not only plot raw data - it uses `stat_` functions within geoms to summarize or transform parts of the data and then plot the results
- For example, the `geom_smooth()` function can fit LOESS, OLS, and robust regression lines using the `method` argument

(Healy, 2018)

Show several fits at once, with a legend

Layering geoms

- If we want to view several *different* fits on the same plot , we can layer new smoothers using `geom_smooth()`
- We can set the `color` and `fill` aesthetics for each fit, so that we can tell apart the different lines
- BUT ggplot will not know that the different fits (each its own layer) are connected to each other, and will not automatically draw a legend telling us which is which

(Healy, 2018)

Layering geoms - Showing a legend

- We get around this limitation by mapping color and fill to a string describing each model we fit, and using `scale_color_manual()` and `scale_color_fill()` to create a legend
- The first step to doing this is to use `brewer.pal()` from `RColorBrewer` to create our colour palette

```
model_colors <- RColorBrewer::brewer.pal(3, "Set1")

model_colors
```

(Healy, 2018)

Layering geoms - Showing a legend

- Next, we make a ggplot of the three `geom_smooth()` options we want to view, mapping the colour and fill within the `aes()` function as the name of the smoother

```
p0 <- ggplot(data = gapminder, mapping = aes(x = log(gdpPercap), y =
lifeExp))

p1 <- p0 + geom_point(alpha = 0.2) +

    geom_smooth(method = "lm", aes(color = "OLS", fill = "OLS")) +

     geom_smooth(method = "lm", formula = y ~ splines::bs(x, df = 3),
aes(color = "Cubic Spline", fill = "Cubic Spline")) +

    geom_smooth(method = "loess", aes(color = "LOESS", fill =
"LOESS"))
```

(Healy, 2018)

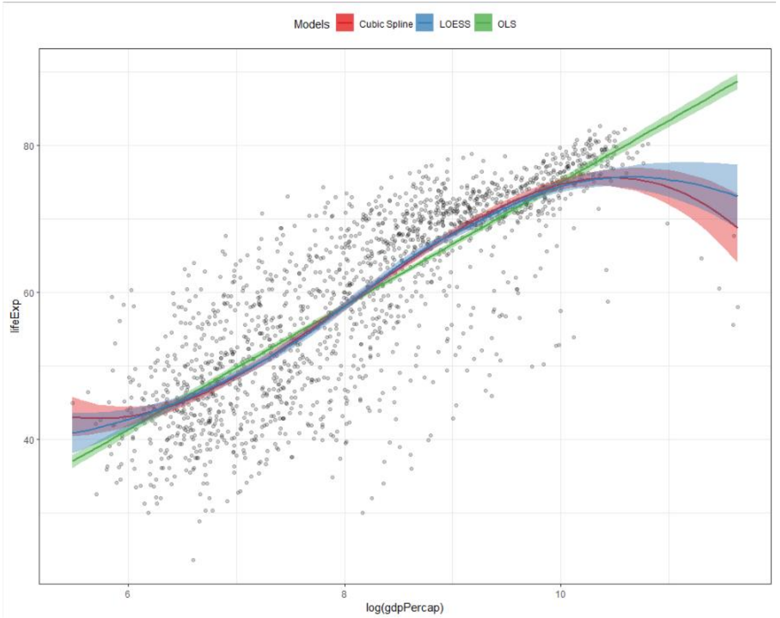
Layering geoms - Showing a legend

- Finally, we call `scale_color_manual()` and `scale_fill_manual()` to assign a colour from our `model_colors` palette to each of our model variables

```
p1 + scale_color_manual(name = "Models", values =
model_colors) + scale_fill_manual(name = "Models",
values = model_colors) + theme(legend.position = "top")
```

(Healy, 2018)

Layering geoms - Showing a legend




```
p1 + scale_color_manual(name = "Models", values = model_colors) +
scale_fill_manual(name = "Models", values = model_colors) +
theme(legend.position = "top")
```

(Healy, 2018)

- We can see that ggplot is useful for exploratory data analysis and comparing model-based trends

- How can we look deeper and get the most out of ggplot as a tool to understand our modeling work in R?

Look inside model objects

Models as objects

- Objects in R range from numbers to vectors to formulas and more
- Until now, we have been working with tibbles and data frame objects that store data in table format, with named columns and different classes of variable (eg. integers, characters, dates)
- Models in R are also objects, but with a different structure
- **If we know the structure of our model objects, we can extract information and visualize it using ggplot**, just like how we have been visualizing data from data frame objects

Example - gapminder dataframe object

- The gapminder sample dataset we have been using is a dataframe - if we enter `gapminder`, our output is a table with named columns
- **We can use `str()` to generate information about the internal structure of any object**
- If we run `str(gapminder)`, our output gives us information on:
  - The class(es) of the `gapminder` object
  - The size of the `gapminder` object
  - Components of the `gapminder` object (eg. our variables)

(Healy, 2018)

Example - gapminder model object

- Models have a more complex internal structure than dataframes, so the output of our `str()` function will give us more pieces of information if we run it on a model
- For example, we create a linear model using our `gapminder` data and store it in an object called `out`, with life expectancy as our dependent variable:

```
out <- lm(formula = lifeExp ~ gdpPercap + pop +
continent, data = gapminder)
```

(Healy, 2018)

Example - gapminder model object

If we run:

```
str(out)
```

our output is a complex set of information

```
List of 13
 $ coefficients : Named num [1:7] 4.78e+01 4.50e-04 6.57e-09 1.35e+01 8.19 ...
  ..- attr(*, "names")= chr [1:7] "(Intercept)" "gdpPercap" "pop" "continentAmericas" ...
 $ residuals    : Named num [1:1704] -27.6 -26.1 -24.5 -22.4 -20.3 ...
  ..- attr(*, "names")= chr [1:1704] "1" "2" "3" "4" ...
 $ effects      : Named num [1:1704] -2455.1 311.1 42.6 101.1 -17.2 ...
  ..- attr(*, "names")= chr [1:1704] "(Intercept)" "gdpPercap" "pop" "continentAmericas" ...
 $ rank         : int 7
 $ fitted.values: Named num [1:1704] 56.4 56.4 56.5 56.5 56.4 ...
  ..- attr(*, "names")= chr [1:1704] "1" "2" "3" "4" ...
 $ assign       : int [1:7] 0 1 2 3 3 3 3
 $ qr           :List of 5
  ..$ qr   : num [1:1704, 1:7] -41.2795 0.0242 0.0242 0.0242 0.0242 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:1704] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:7] "(Intercept)" "gdpPercap" "pop" "continentAmericas" ...
  .. ..- attr(*, "assign")= int [1:7] 0 1 2 3 3 3 3
  .. ..- attr(*, "contrasts")=List of 1
  .. .. ..$ continent: chr "contr.treatment"
  ..$ qraux: num [1:7] 1.02 1.02 1 1.01 1.04 ...
  ..$ pivot: int [1:7] 1 2 3 4 5 6 7
  ..$ tol  : num 1e-07
  ..$ rank : int 7
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 1697
 $ contrasts    :List of 1
  ..$ continent: chr "contr.treatment"
 $ xlevels      :List of 1
  ..$ continent: chr [1:5] "Africa" "Americas" "Asia" "Europe" ...
 $ call         : language lm(formula = lifeExp ~ gdpPercap + pop + continent, data = gapminder)
 $ terms        :Classes 'terms', 'formula'  language lifeExp ~ gdpPercap + pop + continent
  .. ..- attr(*, "variables")= language list(lifeExp, gdpPercap, pop, continent)
  .. ..- attr(*, "factors")= int [1:4, 1:3] 0 1 0 0 0 0 1 0 0 0 ...
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:4] "lifeExp" "gdpPercap" "pop" "continent"
  .. .. .. ..$ : chr [1:3] "gdpPercap" "pop" "continent"
  .. ..- attr(*, "term.labels")= chr [1:3] "gdpPercap" "pop" "continent"
  .. ..- attr(*, "order")= int [1:3] 1 1 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(lifeExp, gdpPercap, pop, continent)
  .. ..- attr(*, "dataClasses")= Named chr [1:4] "numeric" "numeric" "numeric" "factor"
  .. .. ..- attr(*, "names")= chr [1:4] "lifeExp" "gdpPercap" "pop" "continent"
 $ model        :'data.frame':  1704 obs. of  4 variables:
  ..$ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
  ..$ gdpPercap: num [1:1704] 779 821 853 836 740 ...
  ..$ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 22227415 ...
  ..$ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
  ..- attr(*, "terms")=Classes 'terms', 'formula'  language lifeExp ~ gdpPercap + pop + continent
  .. .. ..- attr(*, "variables")= language list(lifeExp, gdpPercap, pop, continent)
  .. .. ..- attr(*, "factors")= int [1:4, 1:3] 0 1 0 0 0 0 1 0 0 0 ...
  .. .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. .. ..$ : chr [1:4] "lifeExp" "gdpPercap" "pop" "continent"
  .. .. .. .. ..$ : chr [1:3] "gdpPercap" "pop" "continent"
  .. .. ..- attr(*, "term.labels")= chr [1:3] "gdpPercap" "pop" "continent"
  .. .. ..- attr(*, "order")= int [1:3] 1 1 1
```

(Healy, 2018)

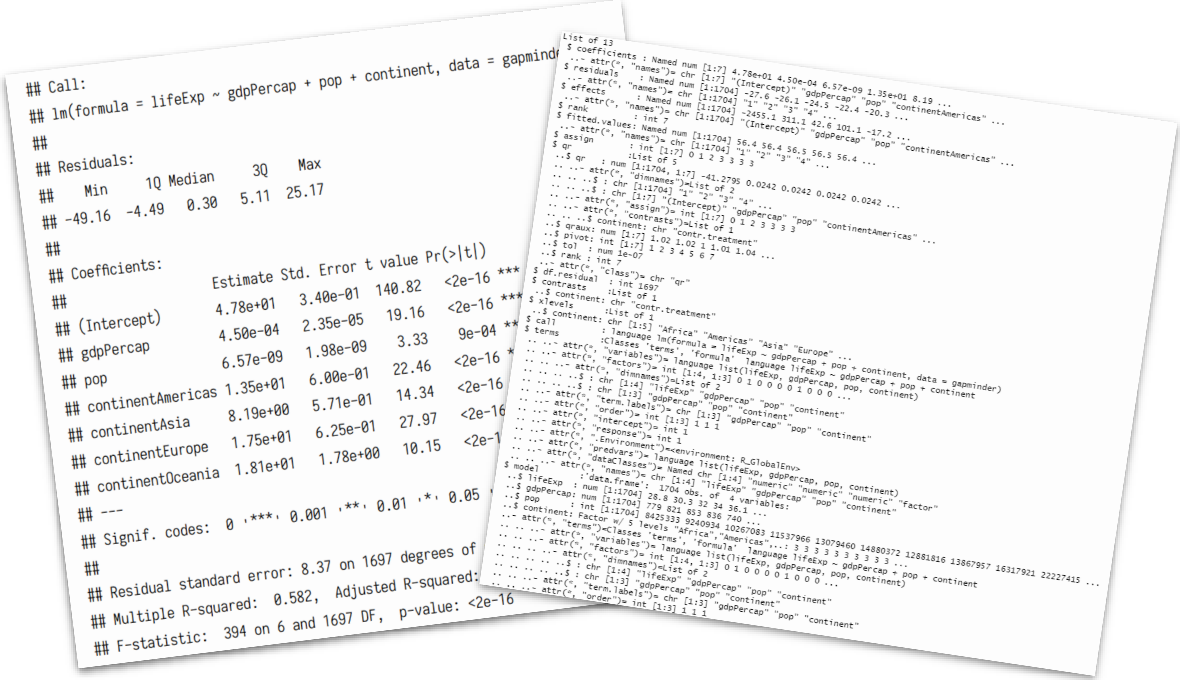
Example - gapminder model object

To view the results of our linear model, we run:

```
summary(out)
```

```
## Call:
## lm(formula = lifeExp ~ gdpPercap + pop + continent, data = gapminder)
##
## Residuals:
##     Min      1Q Median      3Q     Max
## -49.16  -4.49   0.30   5.11  25.17
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       4.78e+01   3.40e-01  140.82   <2e-16 ***
## gdpPercap         4.50e-04   2.35e-05   19.16   <2e-16 ***
## pop               6.57e-09   1.98e-09    3.33    9e-04 ***
## continentAmericas 1.35e+01   6.00e-01   22.46   <2e-16 ***
## continentAsia     8.19e+00   5.71e-01   14.34   <2e-16 ***
## continentEurope   1.75e+01   6.25e-01   27.97   <2e-16 ***
## continentOceania  1.81e+01   1.78e+00   10.15   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.37 on 1697 degrees of freedom
## Multiple R-squared:  0.582,  Adjusted R-squared:  0.581
## F-statistic:  394 on 6 and 1697 DF,  p-value: <2e-16
```

(Healy, 2018)



There is clearly a **lot** of information in our `out` model object, but how do we use it? How can we access specific pieces of information from the model and plot them with ggplot?

(Healy, 2018)

Model elements

- Our model object contains several different named elements
- We can access specific information by name. For example, `out$coefficients`, `out$residuals`, and `out$fitted.values`
- Our `summary()` function selects from these elements and, per Healy (2018), shows them in a way that is "efficient" for sharing information but "untidy" for letting us manipulate that information
- So: we need to do some work with our model data before we can visualize it with ggplot

(Healy, 2018)

Generate predictions to graph

predict()

- We will work through an example case where our objective is to visualize the estimates our model produces over some range for a given variable of interest
- We do this by using the `predict()` function
- **But** for `predict()` to calculate our estimates, it needs new data to which it can fit our model, so we generate a new dataframe with the same columns as the model's original data, but new values for the rows

(Healy, 2018)

Making our new dataframe - expand.grid()

- Still working with our `gapminder` dataset, we will aim to create a dataframe containing the input variables for our linear model:
  - ‘Per capita GDP’ is the hundred evenly spaced elements between the minimum and maximum value from our existing dataset
  - ‘Population’ is held constant at its median value from our existing dataset
  - ‘Continent’ can be all of its five available values
- We use the `expand.grid()` function to generate a new dataframe populated by all combinations of values we give it

(Healy, 2018)

Making our new dataframe - expand.grid()

```
min_gdp <- min(gapminder$gdpPercap)

max_gdp <- max(gapminder$gdpPercap)

med_pop <- median(gapminder$pop)

pred_df <- expand.grid(gdpPercap = (seq(from = min_gdp,
to = max_gdp, length.out = 100)), pop = med_pop,
continent = c("Africa", "Americas", "Asia", "Europe",
"Oceania"))
```

(Healy, 2018)

Making our new dataframe - expand.grid()

- We can view the dimensions of our new dataframe with

```
dim(pred_df)
```

- Our output shows us that our dataframe has 3 columns and 500 rows
  - 3 columns from our three chosen variables
  - 500 rows for our 100 per capita GDP values x 5 possible continents

(Healy, 2018)

Predicting values

- Now we can use `predict()` to input our new variables into our `out` model and calculate the fitted life expectancy value
- By setting `interval = “predict”` as an argument, our output will calculate 95% prediction intervals, as well as the prediction for each data point

```
pred_out <- predict(object = out, newdata = pred_df,
interval = "predict")

head(pred_out)
```

(Healy, 2018)

Binding our data frames

- **Only because we just designed our two dataframes to correspond row by row**, we can bind them together by column into one single dataframe

```
pred_df <- cbind(pred_df, pred_out)

head(pred_df)
```

- Now that our predicted values are in a neat dataframe, we can use our existing ggplot knowledge to plot all or a subset of these values

(Healy, 2018)

Example

- As an example, we can create a visualization to explore our question "How does per capita GDP affect life expectancy in Europe and Africa?"

```
p <- ggplot(data = subset(pred_df, continent %in% c("Europe",
"Africa")), aes(x = gdpPercap, y = fit, ymin = lwr, ymax = upr, color
= continent, fill = continent, group = continent))

p + geom_point(data = subset(gapminder, continent %in% c("Europe",
"Africa")), aes(x = gdpPercap, y = lifeExp, color = continent), alpha
= 0.5, inherit.aes = FALSE) +

  geom_line() +

  geom_ribbon(alpha = 0.2, color = FALSE) +

  scale_x_log10(labels = scales::dollar)
```

(Healy, 2018)

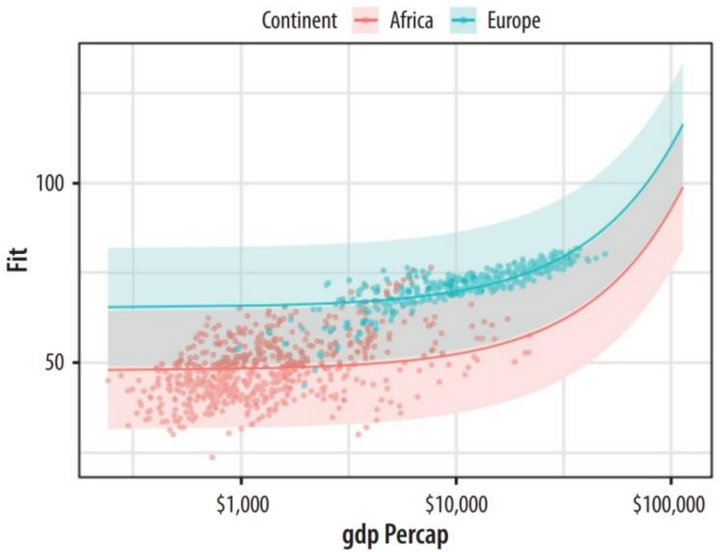
Example - geom_ribbon()

- We use the new function `geom_ribbon()`, which takes the `ymin` and `ymax` arguments defined in our ggplot mapping to define the upper and lower limits of the prediction interval

```
p <- ggplot(data = subset(pred_df, continent %in% c("Europe",
"Africa")), aes(x = gdpPercap, y = fit, ymin = lwr, ymax = upr, color
= continent, fill = continent, group = continent))

p + geom_point(data = subset(gapminder, continent %in% c("Europe",
"Africa")), aes(x = gdpPercap, y = lifeExp, color = continent), alpha
= 0.5, inherit.aes = FALSE) +

  geom_line() +

  geom_ribbon(alpha = 0.2, color = FALSE) +

  scale_x_log10(labels = scales::dollar)
```

(Healy, 2018)

Example - geom_ribbon()




```
p <- ggplot(data = subset(pred_df, continent %in% c("Europe", "Africa")), aes(x = gdpPercap, y
= fit, ymin = lwr, ymax = upr, color = continent, fill = continent, group = continent))

p + geom_point(data = subset(gapminder, continent %in% c("Europe", "Africa")), aes(x =
gdpPercap, y = lifeExp, color = continent), alpha = 0.5, inherit.aes = FALSE) + geom_line() +
geom_ribbon(alpha = 0.2, color = FALSE) + scale_x_log10(labels = scales::dollar)
```

(Healy, 2018)

Activity

- **Decide a research question** to explore visually using our pred_df data (for example, “How does per capita GDP affect life expectancy in Europe and Africa?”)
- **Use ggplot** to generate a visualization from pred_df that can help you answer your question
- **Rejoin the class** and discuss your data visualization and answer to your research question
    - Did you all produce similar graphs? If not, why did you choose one type of graph over another?

Tidy model objects with Broom

Broom

- We can use the `broom` package to go from model outputs to numbers in dataframes that we can easily plot

```
install.packages("broom")

library(broom)
```

- Broom extracts three kinds of information from our models:
  - **Component level** (coefficients, t-statistics)
  - **Observation level** (fitted values and residuals)
  - **Model level** (F statistic, r-squared)

(Healy, 2018)

Component level statistics with tidy()

- The `tidy()` function takes a model object and returns a dataframe of component-level information
- For example, with an added step to round numeric columns to two decimal places:

```
out_conf <- tidy(out)

out_conf |> round_df()
```

```
# A tibble: 7 x 5
  term              estimate std.error statistic p.value
  <chr>                <dbl>     <dbl>     <dbl>   <dbl>
1 (Intercept)          47.8       0.34    141.         0
2 gdpPercap             0         0        19.2        0
3 pop                   0         0         3.33       0
4 continentAmericas    13.5       0.6      22.5        0
5 continentAsia         8.19      0.57     14.3        0
6 continentEurope      17.5       0.62     28.0        0
7 continentOceania     18.1       1.78     10.2        0
```

(Healy, 2018)

Component level statistics with tidy()

- Now we can create a data visualization from this dataframe, the way that we have been doing all along

```
p <- ggplot(out_conf, mapping = aes(x = term, y = estimate))
p + geom_point() + coord_flip()
```

- We can use the `confint()` function to calculate confidence intervals for our estimates

```
out_conf <- tidy(out, conf.int = TRUE)
out_conf %>% round_df()
```

(Healy, 2018)

Component level statistics with tidy()

- We can clean up our visualization by
  - Removing the '(Intercept)' term
  - Removing the word 'continent' from our variables (so we will see 'Americas' instead of 'continentAmericas')

```
out_conf <- subset(out_conf, term %nin% "(Intercept)")

out_conf$nicelabs <- prefix_strip(out_conf$term, "continent")
```

(Healy, 2018)

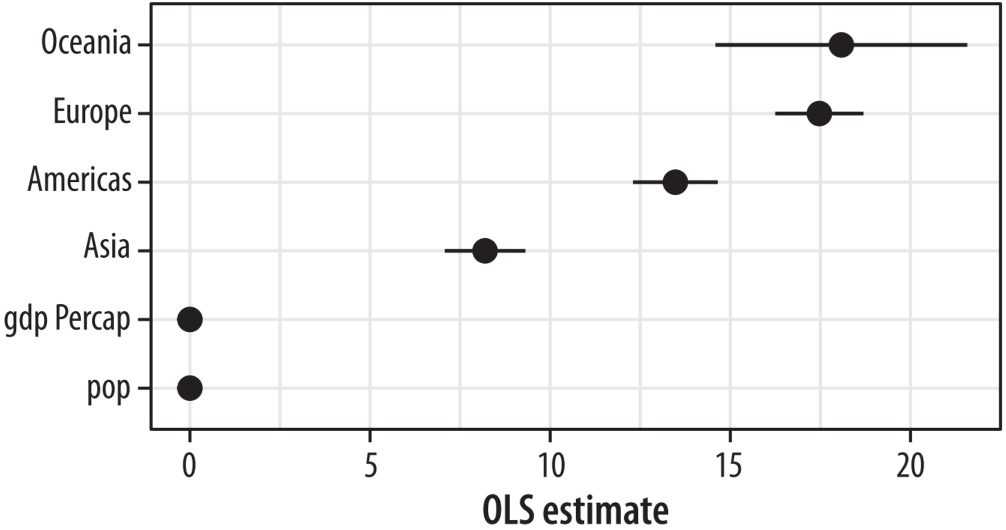
Component level statistics with tidy()

- Finally, we use geom_pointrange() to to make a figure that includes our confidence intervals and is ordered from largest to smallest in magnitude

```
p <- ggplot(out_conf, mapping = aes(x = reorder(nicelabs, estimate), y = estimate, ymin = conf.low, ymax = conf.high))

p + geom_pointrange() + coord_flip() + labs(x = "", y = "OLS Estimate")
```

(Healy, 2018)

Component level statistics with tidy() - Result




```
p <- ggplot(out_conf, mapping = aes(x = reorder(nicelabs, estimate), y =
estimate, ymin = conf.low, ymax = conf.high))

p + geom_pointrange() + coord_flip() + labs(x = "", y = "OLS Estimate")
```

(Healy, 2018)

Observation level statistics with augment()

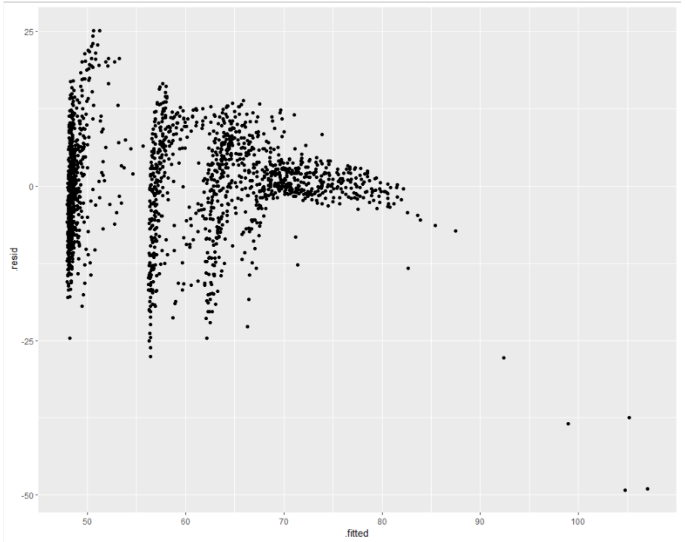
- The `augment()` function returns statistics calculated for each individual observation, so they can be added to the dataframe that our model is based on. These statistics include:
  - **.fitted** → fitted values of the model
  - **.se.fit** → standard errors of the fitted values
  - **.resid** → residuals
  - **.cooksd** → Cook’s distance
  - **.std.resid** → standardized residuals

```
out_aug <- augment(out)
head(out_aug) |> round_df()
```

(Healy, 2018)

Observation level statistics with augment()

- We can use augment() to plot, for example, residuals vs fitted values




```
p <- ggplot(data = out_aug, mapping = aes(x = .fitted, y =
.resid))

p + geom_point()
```

(Healy, 2018)

Model level statistics with glance()

- The `glance()` function returns a table of the statistics generated by our `summary()` output - r squared, p value, etc

```
glance(out) |> round_df()
```

(Healy, 2018)

Grouped analysis and list columns

Using Broom for grouped analysis

- Part of the benefit of Broom is that it lets us quickly fit models to different subsets of our data
- For example, if we want to use our original gapminder dataset to explore the relationship between life expectancy and GDP in Europe, in the year 1977, we could do:

```
eu77 <- gapminder |> filter(continent == "Europe", year
== 1977)

fit <- lm(lifeExp ~ log(gdpPercap), data = eu77)

summary(fit)
```

(Healy, 2018)

Nesting data

- We can use dplyr and Broom to sort the data into groups of continent-year slices, then use the `nest()` function to nest the data contained in each group
- `nest()` lets us create a list column (essentially a table within a table)

```
out_le <- gapminder |>
    group_by(continent, year) |>
    nest()
out_le
```

(Healy, 2018)

Nesting data

- We can view specific information (such as our Europe, 1977 data) by filtering and then unnesting our list column, like so:

```
out_le |> filter(continent == "Europe" & year ==
1977) |> unnest()
```

- Now we can easily and compactly apply our regression analyses to every continent-year combination in our dataset

(Healy, 2018)

Nesting data

- We do this by creating a function that fits our model to a dataframe, then mapping that function to each of our list column's rows, one at a time:

```
fit_ols <- function(df) {lm(lifeExp ~ log(gdpPercap), data = df)}
out_le <- gapminder |>
  group_by(continent, year) |>
  nest() |>
  mutate(model = map(data, fit_ols))
out_le
```

(Healy, 2018)

Nested and tidied data

- Now we have two list columns: ‘data’ and ‘model’; inside each ‘model’ element is a linear model for that continent-year pairing
- We can run our code again, this time cleaning up our data by removing Intercept terms and the outlier observations from Oceania and extracting summary statistics from each model with `tidy()`

```
fit_ols <- function(df) { lm(lifeExp ~ log(gdpPercap), data = df)}

out_tidy <- gapminder |> group_by(continent, year) |> nest() |>
mutate(model = map(data, fit_ols), tidied = map(model, tidy)) |>
unnest(tidied) |> filter(term %nin% "(Intercept)" & continent %nin%
"Oceania")
```

(Healy, 2018)

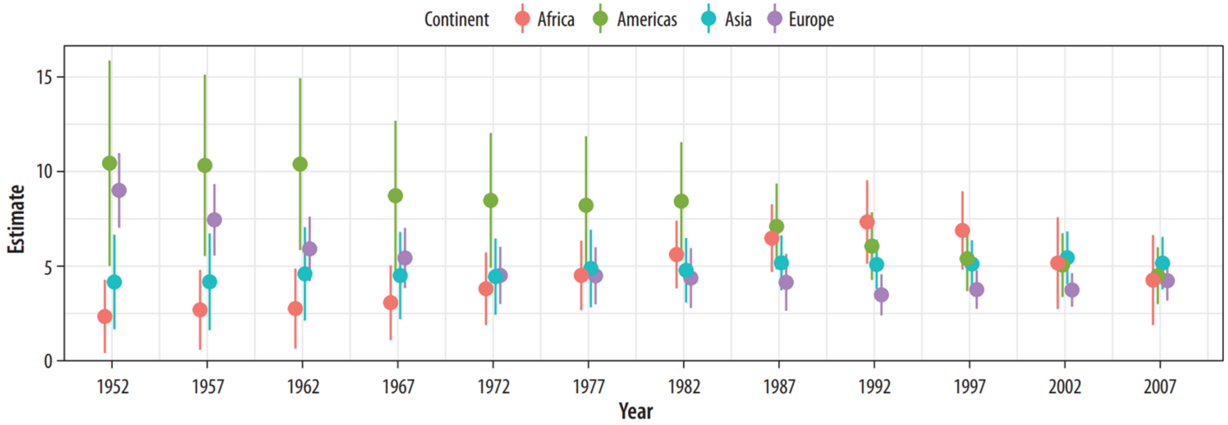
Plotting nested and tidied data

- Now we can use ggplot to plot our regression outputs, with error bars, for each continent-year pairing

```
p <- ggplot(data = out_tidy, mapping = aes(x = year, y = estimate,
ymin = estimate - 2*std.error, ymax = estimate + 2*std.error, group
= continent, color = continent))

p + geom_pointrange(position = position_dodge(width = 1)) +

  scale_x_continuous(breaks = unique(gapminder$year)) +

  theme(legend.position = "top") +

  labs(x = "Year", y = "Estimate", color = "Continent")
```

(Healy, 2018)

Plotting nested and tidied data




```
p <- ggplot(data = out_tidy, mapping = aes(x = year, y = estimate, ymin = estimate - 2*std.error, ymax
= estimate + 2*std.error, group = continent, color = continent))

p + geom_pointrange(position = position_dodge(width = 1)) +

  scale_x_continuous(breaks = unique(gapminder$year)) +

  theme(legend.position = "top") +

  labs(x = "Year", y = "Estimate", color = "Continent")
```

(Healy, 2018)

Get model-based graphics right

Data visualization from models

- Data visualization from statistical models has an extra burden of interpretation - which means, if we are not careful, an extra opportunity to confuse ourselves and our audience
- When creating visualizations from statistical models, we need to remember the tips and best practices from previous lessons in the course, but model-based graphics come with their own guidelines

(Healy, 2018)

Data visualization from models

- Data visualization from statistical models has an extra burden of interpretation - which means, if we are not careful, an extra opportunity to confuse ourselves and our audience
- When creating visualizations from statistical models, we need to remember the tips and best practices from previous lessons in the course, but model-based graphics come with their own guidelines
- As with data visualization in general, these are general rules, not hard-and-fast absolutes - remember: **visualizing data means making situational decisions**

(Healy, 2018)

Data visualization from models - Guidelines

1. **Present your findings in substantive terms** → Show your results in a way that is directly meaningful to the questions your analysis is trying to answer. Think: is there a certain percentile range that matters? What scale should I use?
2. **Show your degree of confidence** → Models come with an estimate of precision, confidence, or significance - communicate this clearly!
3. **Show your data when you can** → Recall our examples showing both the model estimate (eg. a regression line) and the underlying data points (eg. a scatterplot)

(Healy, 2018)

Next...

- Data viz for/as advocacy!