Columbia University in the City of New York

Lecture 8: Modeling II

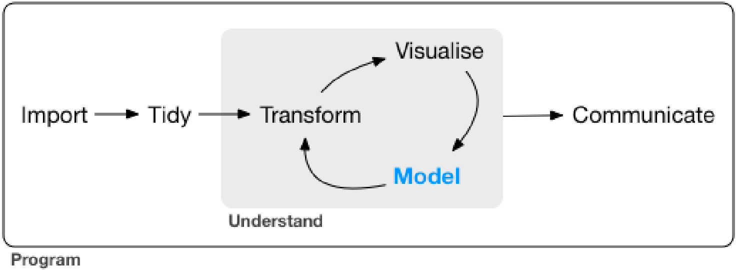
Data Science for Business Analytics

Thibault Vatter

Department of Statistics, Columbia University and HEC Lausanne, UNIL

4/9/2018

Today



- This morning:
  - how models work mechanistically (focus on linear models),
  - how to use models to find patterns in real data.
- This afternoon:
  - how to use **many** simple models,
  - how to combine modeling and programming tools.

As usual, most of the material is borrowed from R for data science.

Many models

To work with large numbers of models, use

1. many simple models to better understand complex datasets.
2. list-columns to store arbitrary data structures in a data frame.
3. the **broom** package to turn models into tidy data.

Note that this part:

- is harder than the others,
- and requires a deeper internalization of ideas (e.g., about modelling, data structures, and iteration).

Outline

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

Reducing code duplication

Three main benefits:

1. Easier to see the intent of your code.
2. Easier to respond to changes in requirements.
3. Likely to have fewer bugs.

Main tools for reducing duplication:

- **Functions** identify repeated patterns of code and extract them out into independent pieces.
- **Iteration** helps you when you need to do the same thing to multiple inputs.

For loops

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

```r
df <- tibble(a = rnorm(10), b = rnorm(10), c = rnorm(10), d = rnorm(10))

c(median(df$a), median(df$b), median(df$c), median(df$d))
```

```
## [1]  0.20443713 -0.35536298  0.09003127 -0.25752642
```

```r
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
```

```
## [1]  0.20443713 -0.35536298  0.09003127 -0.25752642
```

A side note:

```r
y <- vector("double", 0)
seq_along(y)
```

```
## integer(0)
```

```r
1:length(y)
```

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

Modifying an existing object

```r
df <- tibble(a = rnorm(10),
             b = rnorm(10),
             c = rnorm(10),
             d = rnorm(10))

rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)

for (i in seq_along(df)) {
    df[[i]] <- rescale01(df[[i]])
}
```

For loops vs. functionals

To compute the mean of every column:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
```

As a function:

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

How about other quantities?

```r
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

What's “wrong” here?

What's "wrong" here?


```r
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

What's "wrong" here?

```r
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

Without duplication:

```r
f <- function(x, i) abs(x - mean(x)) ^ i
```

Back to column summaries

```r
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
```

```
## [1] 0.3684051 0.4523108 0.8108251 0.5468810
```

```r
col_summary(df, mean)
```

```
## [1] 0.4406485 0.4485883 0.6825104 0.5249303
```

Functional programming

- Use **functions that return functions** as output.
- Pass functions as arguments to others function.

The **purrr** package:

- Functions eliminating the need for many common for loops.
- Similar to the apply family in base R (`apply()`, `lapply()`, `tapply()`, etc), but more consistent and easier to learn.

The goal is to break code into independent pieces:

1. Solve a problem for a single element of the list.
   - Once this is done, purrr generalizes to every element in the list.
2. Break a complex problem down into bite-sized pieces.
   - With purrr, small pieces are composed together with the pipe.

The map functions

- `map()` makes a list.
- `map_lgl()` makes a logical vector.
- `map_int()` makes an integer vector.
- `map_dbl()` makes a double vector.
- `map_chr()` makes a character vector.

Each function:

1. Takes a vector as input.
2. Applies a function to each piece.
3. Returns a new vector that's the same length (and has the same names) as the input.

The return type is determined by the suffix.

Using map functions

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

```r
map_dbl(df, mean)
map_dbl(df, median)
map_dbl(df, sd)
```

```
##         a         b         c         d
## 0.4406485 0.4485883 0.6825104 0.5249303
##         a         b         c         d
## 0.3684051 0.4523108 0.8108251 0.5468810
##         a         b         c         d
## 0.3362143 0.2640144 0.3516401 0.2929669
```

Using the pipe:

```r
df %>% map_dbl(mean)
df %>% map_dbl(median)
df %>% map_dbl(sd)
```

```
##         a         b         c         d
## 0.4406485 0.4485883 0.6825104 0.5249303
##         a         b         c         d
## 0.3684051 0.4523108 0.8108251 0.5468810
##         a         b         c         d
## 0.3362143 0.2640144 0.3516401 0.2929669
```

map_*() vs col_summary()

- All purrr functions are implemented in C (i.e., slightly faster).
- The second argument, .f, the function to apply, can be a formula, a character vector, or an integer vector.
- map_*() uses ... ([dot dot dot]) to pass along additional arguments to .f each time it's called:

```
map_dbl(df, mean, trim = 0.5)
```

```
##         a         b         c         d
## 0.3684051 0.4523108 0.8108251 0.5468810
```

- The map functions preserve names:

```
z <- list(x = 1:3, y = 4:5)
map_int(z, length)
```

```
## x y
## 3 2
```

Shortcut 1

Splits the `mtcars` dataset into three pieces and fits the same linear model to each piece:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

Using a one-sided formula:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

Shortcut 2

Columbia University
In the City of New York

Extract a summary statistic like the R^2:

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

Use a string:

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

Shortcut 3

Use an integer:

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
```

```
## [1] 2 5 8
```

Dealing with failure using `safely()`

- `safely()` is an adverb: it takes a function (a verb) and returns a modified version.
- The modified function always returns a list with two elements:
    1. `result` is the original result.
    2. `error` is an error object.

```
safe_log <- safely(log)
str(safe_log(10))
```

```
## List of 2
##  $ result: num 2.3
##  $ error : NULL
```

```
str(safe_log("a"))
```

```
## List of 2
##  $ result: NULL
##  $ error :List of 2
##   ..$ message: chr "non-numeric argument to mathematical function"
##   ..$ call   : language log(x = x, base = base)
##   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

safely() and map()

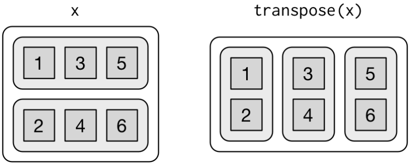
```r
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
```

```
## List of 3
##  $ :List of 2
##   ..$ result: num 0
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: num 2.3
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: NULL
##   ..$ error :List of 2
##   .. ..$ message: chr "non-numeric argument to mathematical function"
##   .. ..$ call   : language log(x = x, base = base)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

transpose()


```r
y <- y %>% transpose()
str(y)
```

```
## List of 2
##  $ result:List of 3
##   ..$ : num 0
##   ..$ : num 2.3
##   ..$ : NULL
##  $ error :List of 3
##   ..$ : NULL
##   ..$ : NULL
##   ..$ :List of 2
##   .. ..$ message: chr "non-numeric argument to mathematical function"
##   .. ..$ call   : language log(x = x, base = base)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```



Typical use


```r
is_ok <- y$error %>% map_lgl(is_null)
x[!is_ok]
```

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

```r
y$result[is_ok] %>% flatten_dbl()
```

```
## [1] 0.000000 2.302585
```

Two other useful adverbs

- `possibly()`: "simpler" than `safely()`, because you give it a default value to return when there is an error.

```
list(1, 10, "a") %>% map_dbl(possibly(log, NA_real_))
```

```
## [1] 0.000000 2.302585       NA
```

- `quietly()`: instead of capturing errors, it captures printed output, messages, and warnings.

```
list(1, -1) %>% map(quietly(log)) %>% str()
```

```
## List of 2
##  $ :List of 4
##   ..$ result  : num 0
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
##  $ :List of 4
##   ..$ result  : num NaN
##   ..$ output  : chr ""
##   ..$ warnings: chr "NaNs produced"
##   ..$ messages: chr(0)
```

Mapping over multiple arguments


```r
mu <- list(5, 10, -3)
mu %>% map(rnorm, n = 5) %>% str()
```

```
## List of 3
##  $ : num [1:5] 4.2 6.25 5.77 4.78 4.58
##  $ : num [1:5] 9.58 11 9.72 11.26 10.65
##  $ : num [1:5] -1.7 -3.87 -2.99 -3.88 -2.4
```

What if you also want to vary the standard deviation?

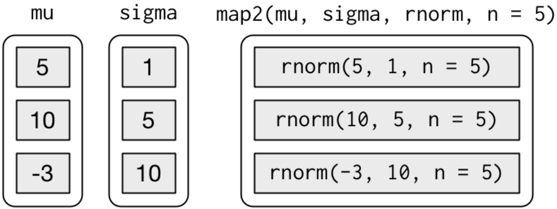
```r
sigma <- list(1, 5, 10)
seq_along(mu) %>% map(~rnorm(5, mu[[.]], sigma[[.]])) %>% str()
```

```
## List of 3
##  $ : num [1:5] 5.12 4.72 6.46 5.23 6
##  $ : num [1:5] 13.91 6.12 6.92 10.23 4.35
##  $ : num [1:5] 2.77 -15.81 13.25 -8.01 13.78
```

map2()


```r
map2(mu, sigma, rnorm, n = 5) %>%
    str()
```

```
## List of 3
##  $ : num [1:5] 4.59 4.03 5.03 5.03 3.32
##  $ : num [1:5] 15.3 4.4 11.7 12.5 10.7
##  $ : num [1:5] -4.19 -1.02 -13.69 -11.03 -14.14
```



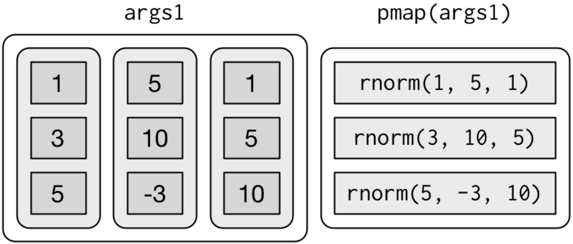
Note that the arguments that

- vary for each call come *before* the function,
- are the same for every call come *after* (using ...).

pmap()

```r
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma)
args1 %>%
    pmap(rnorm) %>%
    str()
```

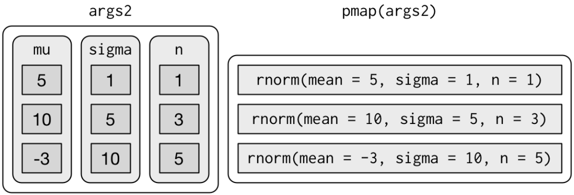
```
## List of 3
##  $ : num 6.58
##  $ : num [1:3] 17.49 11.31 3.84
##  $ : num [1:5] -3.04 12.12 -7.76 4.98 -12.74
```



pmap() cont'd

```r
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>%
    pmap(rnorm) %>%
    str()
```

```
## List of 3
##  $ : num 5.69
##  $ : num [1:3] 5.22 3.84 5.22
##  $ : num [1:5] -11.7 -12.11 4.41 -2.31 -6.24
```



Or better

Columbia University in the City of New York

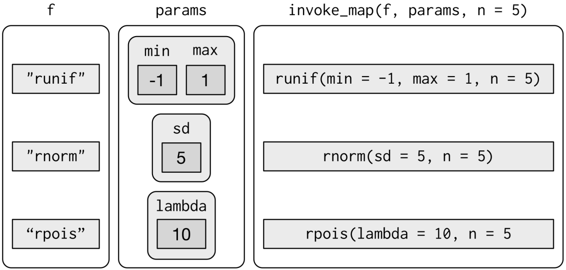
```r
params <- tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5)
params %>%
  pmap(rnorm)
```

```
## [[1]]
## [1] 3.913497
##
## [[2]]
## [1] 4.920355 6.161049 4.401400
##
## [[3]]
## [1]  -7.481742   1.717364 -14.804907  11.702570 -16.114206
```

Invoking different functions


```r
f <- c("runif", "rnorm", "rpois")
param <- list(list(min = -1, max = 1), list(sd = 5), list(lambda = 10))
invoke_map(f, param, n = 5) %>% str()
```

```
## List of 3
##  $ : num [1:5] -0.0769 -0.2496 0.9822 -0.6473 0.6269
##  $ : num [1:5] -7.44 -5.38 5 -3.11 -6.92
##  $ : int [1:5] 15 11 9 13 12
```



Invoking different cont'd

```r
sim <- tribble(~f,      ~params,
               "runif", list(min = -1, max = 1),
               "rnorm", list(sd = 5),
               "rpois", list(lambda = 10))
sim %>%
    mutate(sim = invoke_map(f, params, n = 10)) %>%
    str()
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  3 variables:
##  $ f     : chr  "runif" "rnorm" "rpois"
##  $ params:List of 3
##   ..$ :List of 2
##   .. ..$ min: num -1
##   .. ..$ max: num 1
##   ..$ :List of 1
##   .. ..$ sd: num 5
##   ..$ :List of 1
##   .. ..$ lambda: num 10
##  $ sim   :List of 3
##   ..$ : num  -0.464 0.524 0.973 -0.413 -0.201 ...
##   ..$ : num  1.038 11.54 0.529 2.285 -0.386 ...
##   ..$ : int  13 6 8 13 11 9 10 7 8 7
```

Outline

gapminder

- Summarizes the progression of countries over time using variables like life expectancy and GDP.
- Popularized by Hans Rosling, a Swedish doctor and statistician, in a short video filmed in conjunction with the BBC

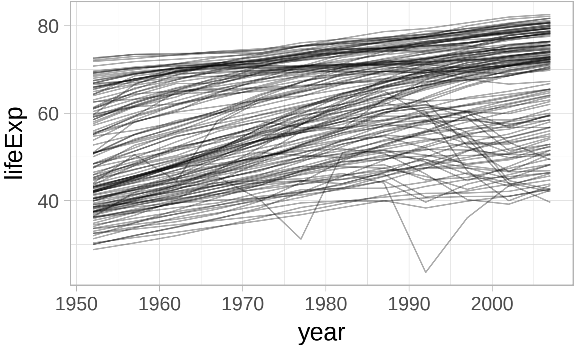
```r
library(gapminder)
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

Focus on three variables

How does life expectancy (`lifeExp`) change over time (`year`) for each country (`country`)?

```
gapminder %>%
  ggplot(aes(year, lifeExp, group = country)) +
    geom_line(alpha = 1/3)
```

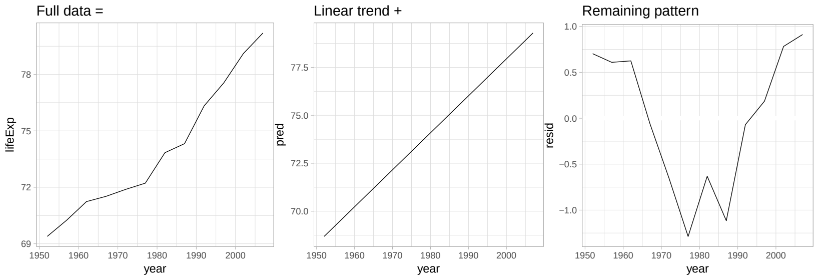


Model for a single country


```r
nz <- filter(gapminder, country == "New Zealand")
nz %>% ggplot(aes(year, lifeExp)) + geom_line() + ggtitle("Full data = ")

nz_mod <- lm(lifeExp ~ year, data = nz)
nz %>% add_predictions(nz_mod) %>%
  ggplot(aes(year, pred)) + geom_line() + ggtitle("Linear trend + ")

nz %>% add_residuals(nz_mod) %>%
  ggplot(aes(year, resid)) +
    geom_hline(yintercept = 0, colour = "white", size = 3) +
    geom_line() + ggtitle("Remaining pattern")
```



Nested data

```r
(by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest())
```

```
## # A tibble: 142 x 3
##    country     continent data
##    <fct>       <fct>     <list>
##  1 Afghanistan Asia      <tibble [12 x 4]>
##  2 Albania     Europe    <tibble [12 x 4]>
##  3 Algeria     Africa    <tibble [12 x 4]>
##  4 Angola      Africa    <tibble [12 x 4]>
##  5 Argentina   Americas  <tibble [12 x 4]>
##  6 Australia   Oceania   <tibble [12 x 4]>
##  7 Austria     Europe    <tibble [12 x 4]>
##  8 Bahrain     Asia      <tibble [12 x 4]>
##  9 Bangladesh  Asia      <tibble [12 x 4]>
## 10 Belgium     Europe    <tibble [12 x 4]>
## # ... with 132 more rows
```

- In a grouped data frame, each row is an observation.
- In a nested data frame, each row is a group.

List-columns

A model-fitting function applied to every country:

```
country_model <- function(df) lm(lifeExp ~ year, data = df)
models <- map(by_country$data, country_model)
```

Or add an additional list-column:

```
(by_country <- by_country %>% mutate(model = map(data, country_model)))
```

```
## # A tibble: 142 x 4
##    country     continent data              model
##    <fct>       <fct>     <list>            <list>
##  1 Afghanistan Asia      <tibble [12 x 4]> <S3: lm>
##  2 Albania     Europe    <tibble [12 x 4]> <S3: lm>
##  3 Algeria     Africa    <tibble [12 x 4]> <S3: lm>
##  4 Angola      Africa    <tibble [12 x 4]> <S3: lm>
##  5 Argentina   Americas  <tibble [12 x 4]> <S3: lm>
##  6 Australia   Oceania   <tibble [12 x 4]> <S3: lm>
##  7 Austria     Europe    <tibble [12 x 4]> <S3: lm>
##  8 Bahrain     Asia      <tibble [12 x 4]> <S3: lm>
##  9 Bangladesh  Asia      <tibble [12 x 4]> <S3: lm>
## 10 Belgium     Europe    <tibble [12 x 4]> <S3: lm>
## # ... with 132 more rows
```

Why bother?

Columbia University in the City of New York

- Avoid leaving the list of models as a free-floating object.
- No need to manually keep them in sync when using e.g. `filter` or `arrange`.

```
by_country %>% filter(continent == "Europe")
```

```
## # A tibble: 30 x 4
##    country                continent data              model
##    <fct>                  <fct>     <list>            <list>
##  1 Albania                Europe    <tibble [12 x 4]> <S3: lm>
##  2 Austria                Europe    <tibble [12 x 4]> <S3: lm>
##  3 Belgium                Europe    <tibble [12 x 4]> <S3: lm>
##  4 Bosnia and Herzegovina Europe    <tibble [12 x 4]> <S3: lm>
##  5 Bulgaria               Europe    <tibble [12 x 4]> <S3: lm>
##  6 Croatia                Europe    <tibble [12 x 4]> <S3: lm>
##  7 Czech Republic         Europe    <tibble [12 x 4]> <S3: lm>
##  8 Denmark                Europe    <tibble [12 x 4]> <S3: lm>
##  9 Finland                Europe    <tibble [12 x 4]> <S3: lm>
## 10 France                 Europe    <tibble [12 x 4]> <S3: lm>
## # ... with 20 more rows
```

Adding residuals


```
(by_country <- by_country %>%
  mutate(resids = map2(data, model, add_residuals)))
```

```
## # A tibble: 142 x 5
##    country     continent data              model    resids
##    <fct>       <fct>     <list>            <list>   <list>
##  1 Afghanistan Asia      <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  2 Albania     Europe    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  3 Algeria     Africa    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  4 Angola      Africa    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  5 Argentina   Americas  <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  6 Australia   Oceania   <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  7 Austria     Europe    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  8 Bahrain     Asia      <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
##  9 Bangladesh  Asia      <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
## 10 Belgium     Europe    <tibble [12 x 4]> <S3: lm> <tibble [12 x 5]>
## # ... with 132 more rows
```

Unnesting

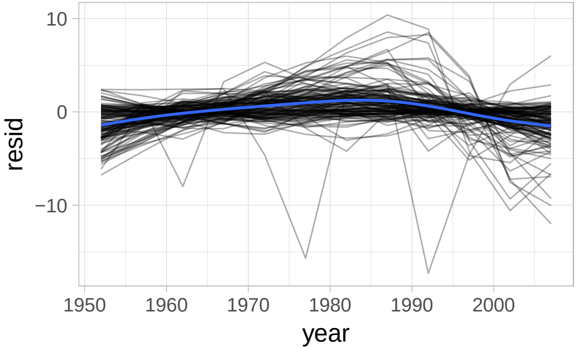
COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

```r
resids <- unnest(by_country, resids)
resids
```

```
## # A tibble: 1,704 x 7
##    country     continent  year lifeExp      pop gdpPercap   resid
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>   <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779. -1.11
##  2 Afghanistan Asia       1957    30.3  9240934      821. -0.952
##  3 Afghanistan Asia       1962    32.0 10267083      853. -0.664
##  4 Afghanistan Asia       1967    34.0 11537966      836. -0.0172
##  5 Afghanistan Asia       1972    36.1 13079460      740.  0.674
##  6 Afghanistan Asia       1977    38.4 14880372      786.  1.65
##  7 Afghanistan Asia       1982    39.9 12881816      978.  1.69
##  8 Afghanistan Asia       1987    40.8 13867957      852.  1.28
##  9 Afghanistan Asia       1992    41.7 16317921      649.  0.754
## 10 Afghanistan Asia       1997    41.8 22227415      635. -0.534
## # ... with 1,694 more rows
```

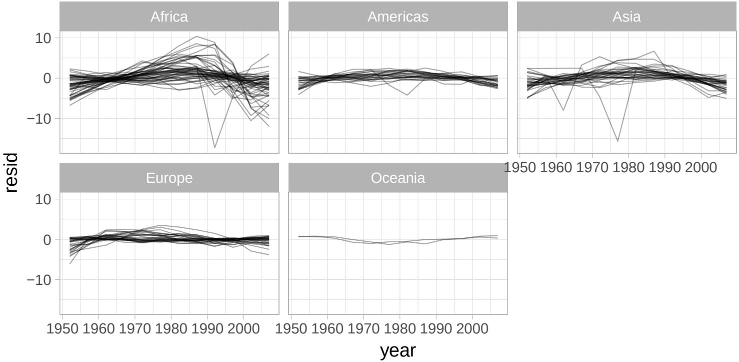
Visualizing the residuals


```r
resids %>%
  ggplot(aes(year, resid)) +
    geom_line(aes(group = country), alpha = 1 / 3) +
    geom_smooth(se = FALSE)
```



Visualizing the residuals cont'd

```r
resids %>% ggplot(aes(year, resid, group = country)) +
    geom_line(alpha = 1 / 3) + facet_wrap(~continent)
```



Model quality

```
broom::glance(nz_mod)
```

```
##   r.squared adj.r.squared     sigma statistic      p.value df
## 1 0.9535846     0.9489431 0.8043472  205.4459 5.407324e-08  2
##      logLik      AIC    BIC deviance df.residual
## 1 -13.32064 32.64128 34.096 6.469743          10
```

```
by_country %>% mutate(glance = map(model, glance)) %>% unnest(glance)
```

```
## # A tibble: 142 x 16
##    country continent data  model resids r.squared adj.r.squared sigma
##    <fct>   <fct>     <lis> <lis> <list>     <dbl>         <dbl> <dbl>
##  1 Afghan~ Asia      <tib~ <S3:~ <tibb~     0.948         0.942 1.22
##  2 Albania Europe    <tib~ <S3:~ <tibb~     0.911         0.902 1.98
##  3 Algeria Africa    <tib~ <S3:~ <tibb~     0.985         0.984 1.32
##  4 Angola  Africa    <tib~ <S3:~ <tibb~     0.888         0.877 1.41
##  5 Argent~ Americas  <tib~ <S3:~ <tibb~     0.996         0.995 0.292
##  6 Austra~ Oceania   <tib~ <S3:~ <tibb~     0.980         0.978 0.621
##  7 Austria Europe    <tib~ <S3:~ <tibb~     0.992         0.991 0.407
##  8 Bahrain Asia      <tib~ <S3:~ <tibb~     0.967         0.963 1.64
##  9 Bangla~ Asia      <tib~ <S3:~ <tibb~     0.989         0.988 0.977
## 10 Belgium Europe    <tib~ <S3:~ <tibb~     0.995         0.994 0.293
## # ... with 132 more rows, and 8 more variables: statistic <dbl>,
## #   p.value <dbl>, df <int>, logLik <dbl>, AIC <dbl>, BIC <dbl>,
## #   deviance <dbl>, df.residual <int>
```

Or better


```r
glance <- by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance, .drop = TRUE)
glance
```

```
## # A tibble: 142 x 13
##    country continent r.squared adj.r.squared sigma statistic  p.value
##    <fct>   <fct>         <dbl>         <dbl> <dbl>     <dbl>    <dbl>
##  1 Afghan~ Asia          0.948         0.942 1.22      181.  9.84e- 8
##  2 Albania Europe        0.911         0.902 1.98      102.  1.46e- 6
##  3 Algeria Africa        0.985         0.984 1.32      662.  1.81e-10
##  4 Angola  Africa        0.888         0.877 1.41       79.1 4.59e- 6
##  5 Argent~ Americas      0.996         0.995 0.292    2246.  4.22e-13
##  6 Austra~ Oceania       0.980         0.978 0.621     481.  8.67e-10
##  7 Austria Europe        0.992         0.991 0.407    1261.  7.44e-12
##  8 Bahrain Asia          0.967         0.963 1.64      291.  1.02e- 8
##  9 Bangla~ Asia          0.989         0.988 0.977     930.  3.37e-11
## 10 Belgium Europe        0.995         0.994 0.293    1822.  1.20e-12
## # ... with 132 more rows, and 6 more variables: df <int>,
## #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
## #   df.residual <int>
```

Which models don't fit well?

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

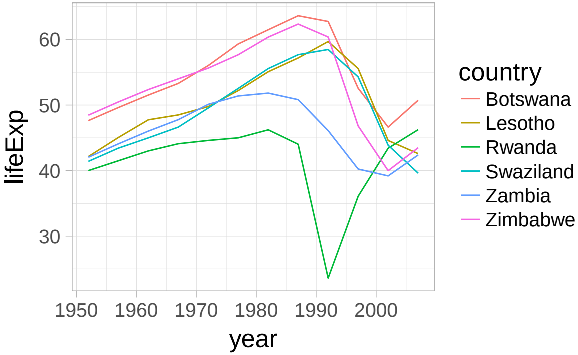
```r
glance %>%
  arrange(r.squared)
```

```
## # A tibble: 142 x 13
##    country  continent r.squared adj.r.squared sigma statistic p.value
##    <fct>    <fct>         <dbl>         <dbl> <dbl>     <dbl>   <dbl>
##  1 Rwanda   Africa       0.0172      -0.0811   6.56     0.175  0.685
##  2 Botswana Africa       0.0340      -0.0626   6.11     0.352  0.566
##  3 Zimbabwe Africa       0.0562      -0.0381   7.21     0.596  0.458
##  4 Zambia   Africa       0.0598      -0.0342   4.53     0.636  0.444
##  5 Swazila~ Africa       0.0682      -0.0250   6.64     0.732  0.412
##  6 Lesotho  Africa       0.0849      -0.00666  5.93     0.927  0.358
##  7 Cote d'~ Africa       0.283        0.212    3.93     3.95   0.0748
##  8 South A~ Africa       0.312        0.244    4.74     4.54   0.0588
##  9 Uganda   Africa       0.342        0.276    3.19     5.20   0.0457
## 10 Congo, ~ Africa       0.348        0.283    2.43     5.34   0.0434
## # ... with 132 more rows, and 6 more variables: df <int>,
## #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
## #   df.residual <int>
```

Visualize


```r
bad_fit <- filter(glance, r.squared < 0.25)

gapminder %>% semi_join(bad_fit, by = "country") %>%
  ggplot(aes(year, lifeExp, colour = country)) + geom_line()
```



Outline

List-columns

```r
tibble(x = list(1:3, 3:5),
       y = c("1, 2", "3, 4, 5"))
```

```
## # A tibble: 2 x 2
##   x         y
##   <list>    <chr>
## 1 <int [3]> 1, 2
## 2 <int [3]> 3, 4, 5
```

```r
tribble(~x, ~y,
        1:3, "1, 2",
        3:5, "3, 4, 5")
```

```
## # A tibble: 2 x 2
##   x         y
##   <list>    <chr>
## 1 <int [3]> 1, 2
## 2 <int [3]> 3, 4, 5
```

An effective list-column pipeline

1. Create the list-column using one of `nest()`, `summarise()` + `list()`, or `mutate()` + a map function.
2. Create other intermediate list-columns by transforming existing list columns with `map()`, `map2()` or `pmap()`.
3. Simplify the list-column back down to a data frame or atomic vector.

Creating list-columns

1. With `nest()` to convert a grouped data frame into a nested data frame where you have list-column of data frames.
2. With `mutate()` and vectorised functions that return a list.
3. With `summarise()` and summary functions that return multiple results.

Create with nesting I


```
gapminder %>% group_by(country, continent) %>% nest()
```

```
## # A tibble: 142 x 3
##    country     continent data
##    <fct>       <fct>     <list>
##  1 Afghanistan Asia      <tibble [12 x 4]>
##  2 Albania     Europe    <tibble [12 x 4]>
##  3 Algeria     Africa    <tibble [12 x 4]>
##  4 Angola      Africa    <tibble [12 x 4]>
##  5 Argentina   Americas  <tibble [12 x 4]>
##  6 Australia   Oceania   <tibble [12 x 4]>
##  7 Austria     Europe    <tibble [12 x 4]>
##  8 Bahrain     Asia      <tibble [12 x 4]>
##  9 Bangladesh  Asia      <tibble [12 x 4]>
## 10 Belgium     Europe    <tibble [12 x 4]>
## # ... with 132 more rows
```

Create with nesting II

Columbia University in the City of New York

```r
gapminder %>% nest(year:gdpPercap)
```

```
## # A tibble: 142 x 3
##    country     continent data
##    <fct>       <fct>     <list>
##  1 Afghanistan Asia      <tibble [12 x 4]>
##  2 Albania     Europe    <tibble [12 x 4]>
##  3 Algeria     Africa    <tibble [12 x 4]>
##  4 Angola      Africa    <tibble [12 x 4]>
##  5 Argentina   Americas  <tibble [12 x 4]>
##  6 Australia   Oceania   <tibble [12 x 4]>
##  7 Austria     Europe    <tibble [12 x 4]>
##  8 Bahrain     Asia      <tibble [12 x 4]>
##  9 Bangladesh  Asia      <tibble [12 x 4]>
## 10 Belgium     Europe    <tibble [12 x 4]>
## # ... with 132 more rows
```

Create from vectorized functions

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

```
df <- tribble(~x1, "a,b,c", "d,e,f,g")

df %>% mutate(x2 = stringr::str_split(x1, ","))
```

```
## # A tibble: 2 x 2
##   x1      x2
##   <chr>   <list>
## 1 a,b,c   <chr [3]>
## 2 d,e,f,g <chr [4]>
```

```
sim <- tribble(~f,       ~params,
               "runif", list(min = -1, max = -1),
               "rnorm", list(sd = 5),
               "rpois", list(lambda = 10))

sim %>% mutate(sims = invoke_map(f, params, n = 10))
```

```
## # A tibble: 3 x 3
##   f     params     sims
##   <chr> <list>     <list>
## 1 runif <list [2]> <dbl [10]>
## 2 rnorm <list [1]> <dbl [10]>
## 3 rpois <list [1]> <int [10]>
```

Create from multivalued summaries II

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

What's wrong here?

```
mtcars %>% group_by(cyl) %>% summarise(q = quantile(mpg))
```

```
## Error in summarise_impl(.data, dots): Column `q` must be length 1 (a summary
```

Use list-columns:

```
mtcars %>% group_by(cyl) %>% summarise(q = list(quantile(mpg)))
```

```
## # A tibble: 3 x 2
##     cyl q
##   <dbl> <list>
## 1    4. <dbl [5]>
## 2    6. <dbl [5]>
## 3    8. <dbl [5]>
```

Create from multivalued summaries II

```r
probs <- c(0.01, 0.25, 0.5, 0.75, 0.99)
mtcars %>% group_by(cyl) %>%
  summarise(p = list(probs), q = list(quantile(mpg, probs))) %>%
  unnest()
```

```
## # A tibble: 15 x 3
##      cyl      p     q
##    <dbl>  <dbl> <dbl>
##  1    4. 0.0100  21.4
##  2    4. 0.250   22.8
##  3    4. 0.500   26.0
##  4    4. 0.750   30.4
##  5    4. 0.990   33.8
##  6    6. 0.0100  17.8
##  7    6. 0.250   18.6
##  8    6. 0.500   19.7
##  9    6. 0.750   21.0
## 10    6. 0.990   21.4
## 11    8. 0.0100  10.4
## 12    8. 0.250   14.4
## 13    8. 0.500   15.2
## 14    8. 0.750   16.2
## 15    8. 0.990   19.1
```

Simplifying list-columns

1. If you want a single value, use `mutate()` with `map_lgl()`, `map_int()`, `map_dbl()`, and `map_chr()` to create an atomic vector.
2. If you want many values, use `unnest()` to convert list-columns back to regular columns, repeating the rows as many times as necessary.

List to vector

```
df <- tribble(~x, letters[1:5], 1:3, runif(5))

df %>% mutate(type = map_chr(x, typeof), length = map_int(x, length))
```

```
## # A tibble: 3 x 3
##   x         type      length
##   <list>    <chr>      <int>
## 1 <chr [5]> character      5
## 2 <int [3]> integer        3
## 3 <dbl [5]> double         5
```

Unnesting


```
tibble(x = 1:2, y = list(1:4, 1)) %>% unnest(y)
```

```
## # A tibble: 5 x 2
##       x     y
##   <int> <dbl>
## 1     1    1.
## 2     1    2.
## 3     1    3.
## 4     1    4.
## 5     2    1.
```

When columns contain different number of elements:

```
tribble(~x, ~y,           ~z,
        1, "a",           1:2,
        2, c("b", "c"), 3) %>% unnest(y, z)
```

```
## Error: All nested columns must have the same number of elements.
```

Making tidy data with broom

1. `broom::glance(model)`
   - A row for each model.
   - Columns give a model summary (measure of model quality, complexity, or combination of both).
2. `broom::tidy(model)`
   - A row for each coefficient in the model.
   - Columns give information about the estimate or its variability.
3. `broom::augment(model, data)`
   - A row for each row in `data`.
   - Adds extra values like residuals, and influence statistics.