14 How to run many models with the new dplyr grouping functions

What will this tutorial cover?

In this tutorial, you will learn six ways to run many models simultaneously using a set of Tidyverse functions. We will create 142 linear regression models and figure out how to extract the model parameters and test statistics from them. Also, you will get to know some new grouping function introduced to dplyr in 2019.

Who do I have to thank?

I would like to thank Mara Averick, Chris Etìenne , and Indrajeet Patil ( ) for their insightful tweets on this topic and helping me get started.

Hadley Wickham showed us in 2016 that you can run many models at once with a few Tidyverse functions (see also this chapter from the R for Data Science book). Using many models can be a powerful technique to gain insights from your data. A classic example is the Gapminder dataset. Suppose you want to find out whether life expectancy has shown a linear trend over the past 50 years in countries around the world.

First, we need access to the data. Fortunately, the gapminder package contains a data frame with the life expectancy of each country on each continent from 1952 to 2007:

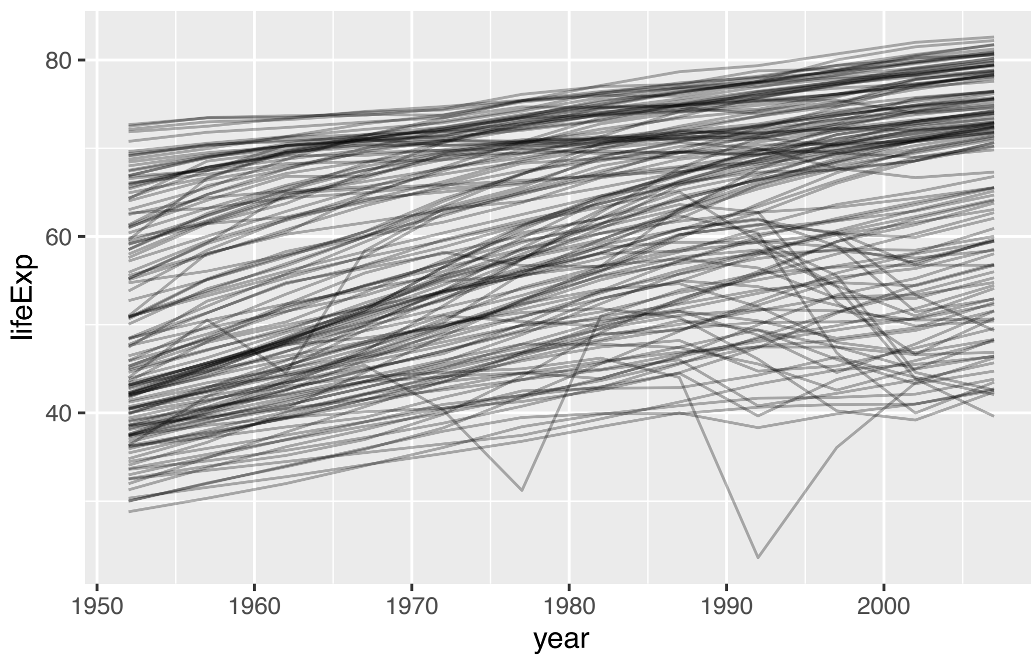
```
library(gapminder)
gapminder
```

```
# 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
```

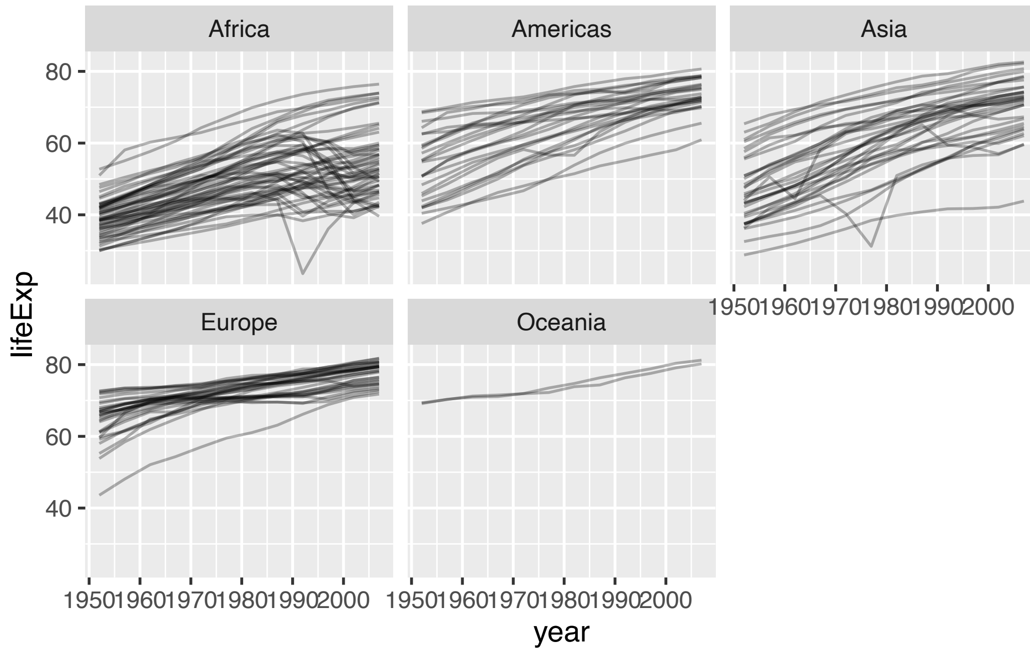
We can trace the development of life expectancy in these countries using a line chart:

```r
ggplot(gapminder,
       aes(x = year, y = lifeExp, group = country)) +
  geom_line(alpha = .3)
```



What we see is that on average life expectancy has risen from 1952 to 2007. Some countries have experienced a drastic decline in life expectancy. We can get a better picture if we split the plot by continent.

```r
ggplot(gapminder,
       aes(x = year, y = lifeExp, group = country)) +
  geom_line(alpha = .3) +
  facet_wrap(vars(continent))
```



We can see that there is one major decline in Africa and two major declines in Asia.

14.1 Building a single linear model

To find out how well life expectancy has followed a linear trend over the past 50 years we build a linear regression model with year as the independent variable and life expectancy as the dependent variable :

```
model <- lm(lifeExp ~ year, data = gapminder)
```

We tell the function `lm` that `year` is our independent variable and `lifeExp` is our dependent variable.

Once we have created the model, we can retrieve the results parameters and test statistics with the `summary` function:

```
summary(model)
```

```
Call:
lm(formula = lifeExp ~ year, data = gapminder)
```

```
Residuals:
    Min      1Q  Median      3Q     Max 
-39.949  -9.651   1.697  10.335  22.158 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept) -585.65219   32.31396  -18.12   <2e-16 ***
year           0.32590    0.01632   19.96   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.63 on 1702 degrees of freedom
Multiple R-squared:  0.1898,    Adjusted R-squared:  0.1893 
F-statistic: 398.6 on 1 and 1702 DF,  p-value: < 2.2e-16
```

We see that our regression coefficient for the independent variable year is positive (year = 0.32590), which means that life expectancy has increased over the years.

To perform further analysis with the parameters of our model, we can pipe the model into the `tidy` function from the broom package:

```r
library(broom)

model %>%
  tidy()
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) -586.      32.3        -18.1 2.90e-67
2 year           0.326    0.0163      20.0 7.55e-80
```

Similarly, we can get the test statistics of our model with `glance`:

```r
model %>%
  broom::glance()
```

```
# A tibble: 1 x 12
  r.squared adj.r.sq~1 sigma stati~2  p.value    df logLik    AIC    BIC devia~3
      <dbl>      <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
```

```
1     0.190      0.189  11.6    399. 7.55e-80     1 -6598. 13202. 13218. 230229.
# ... with 2 more variables: df.residual <int>, nobs <int>, and abbreviated
#   variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

That's all well and good, but how would we apply the same model to each country?

14.2 The split > apply > combine technique

The solution to this problem is the split > apply > combine technique. You have already come across this idea twice:

The combination of `group_by` and `summarise` is a method of split > apply > combine. For example, we could split the data by continent and year, calculate the mean for each group (apply), and combine the results in a data frame:

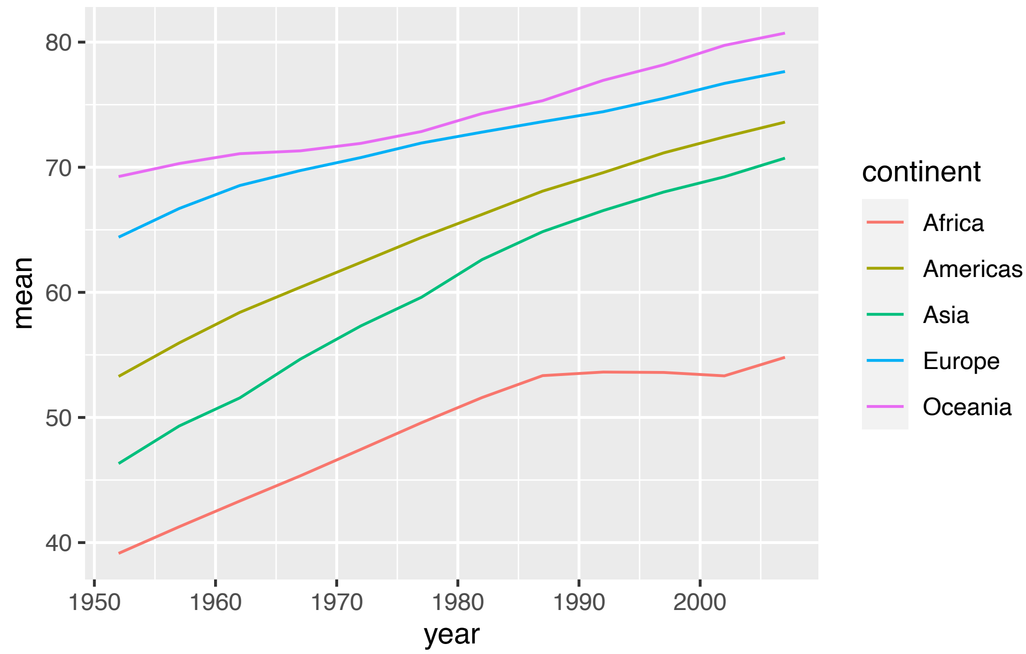
```
(group_by_summarise_example <- gapminder %>%
  group_by(continent, year) %>%
  summarise(mean = mean(lifeExp, na.rm = TRUE)) %>%
  ungroup())
```

```
`summarise()` has grouped output by 'continent'. You can override using the
`.groups` argument.
```

```
# A tibble: 60 x 3
   continent  year  mean
   <fct>     <int> <dbl>
 1 Africa     1952  39.1
 2 Africa     1957  41.3
 3 Africa     1962  43.3
 4 Africa     1967  45.3
 5 Africa     1972  47.5
 6 Africa     1977  49.6
 7 Africa     1982  51.6
 8 Africa     1987  53.3
 9 Africa     1992  53.6
10 Africa     1997  53.6
# ... with 50 more rows
```

With this data, we can plot the development of life expectancy on the five continents:

```r
group_by_summarise_example %>%
  ggplot(aes(x = year, y = mean)) +
  geom_line(aes(color = continent))
```



Another method for slice > apply > combine is to use `group_by` with `slice_max`, and `ungroup`:

```r
(group_by_with_slice_max <- gapminder %>%
  group_by(continent) %>%
  slice_max(lifeExp, n = 1) %>%
  ungroup())
```

```
# A tibble: 5 x 6
  country   continent  year lifeExp       pop gdpPercap
  <fct>     <fct>     <int>   <dbl>     <int>     <dbl>
1 Reunion   Africa     2007    76.4    798094     7670.
2 Canada    Americas   2007    80.7  33390141    36319.
3 Japan     Asia       2007    82.6 127467972    31656.
4 Iceland   Europe     2007    81.8    301931    36181.
5 Australia Oceania    2007    81.2  20434176    34435.
```

In this example, for each continent, we found the year in which life expectancy was highest over the last 50 years. In Africa, for example, the highest life expectancy ever measured was in Reunion in 2007.

14.3 Split > apply > combine for running many models

Now that you have an idea of what the split > apply > combine does, let's use it to run many models. And let's run the same linear model we just created for each country. From the results of these models, we can get an idea of where life expectancy is not following a linear trend.

Here is an example of how that might work. I will explain the code in a second.

```
(test_statistics <- gapminder %>%
  split(.$country) %>%
  map_dfr(\(.x) lm(lifeExp ~ year, .x) %>% broom::glance()))
```

```
# A tibble: 142 x 12
   r.squared adj.r.squ~1 sigma stati~2  p.value    df logLik   AIC   BIC devia~3
       <dbl>       <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>
 1     0.948       0.942 1.22    181.  9.84e- 8     1 -18.3  42.7  44.1   15.0
 2     0.911       0.902 1.98    102.  1.46e- 6     1 -24.1  54.3  55.8   39.3
 3     0.985       0.984 1.32    662.  1.81e-10     1 -19.3  44.6  46.0   17.5
 4     0.888       0.877 1.41     79.1 4.59e- 6     1 -20.0  46.1  47.5   19.8
 5     0.996       0.995 0.292  2246.  4.22e-13     1  -1.17  8.35  9.80   0.854
 6     0.980       0.978 0.621   481.  8.67e-10     1 -10.2  26.4  27.9    3.85
 7     0.992       0.991 0.407  1261.  7.44e-12     1  -5.16 16.3  17.8    1.66
 8     0.967       0.963 1.64    291.  1.02e- 8     1 -21.9  49.7  51.2   26.9
 9     0.989       0.988 0.977   930.  3.37e-11     1 -15.7  37.3  38.8    9.54
10     0.995       0.994 0.293  1822.  1.20e-12     1  -1.20  8.40  9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

First, we use the `split` function from base R to split the data frame into a list. Each list element contains a data frame of one country:

```
list_of_countries <- gapminder %>%
  split(.$country)
```

```
list_of_countries %>% length
```

```
[1] 142
```

The list contains 142 elements, which corresponds to the number of countries in the data frame:

```
gapminder$country %>% unique() %>% length()
```

```
[1] 142
```

Next, we apply two functions to each element of the list. We loop over the data frame of each country with `map_dfr` because we know that the output of the two functions will be a data frame (hence `dfr`). First we run a linear model for each list element. Perhaps you have stumbled across this code:

```
\(.x) lm(lifeExp ~ year, .x)
```

`\(.x)` is a shorthand option for an anonymous function in R. It was introduced with R 4.1.0 (Keith McNulty wrote a nice blog post about this).

Here is an simple example:

```
(\(.x) paste(.x, "loves R"))("Christian")
```

```
[1] "Christian loves R"
```

In our case the anonymous function returns the model object. This is how it looks for the first country in our list:

```
list_of_countries[[1]] %>%
  {lm(lifeExp ~ year, .)}
```

```

Call:
lm(formula = lifeExp ~ year, data = .)

Coefficients:
(Intercept)         year
  -507.5343       0.2753
```

With `glance` we extract the test statistics from the model object:

```r
list_of_countries[[1]] %>%
  {lm(lifeExp ~ year, .)} %>%
  glance()
```

```
# A tibble: 1 x 12
  r.squ~1 adj.r~2 sigma stati~3 p.value    df logLik   AIC   BIC devia~4 df.re~5
    <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>   <int>
1   0.948   0.942  1.22    181. 9.84e-8     1  -18.3  42.7  44.1    15.0      10
# ... with 1 more variable: nobs <int>, and abbreviated variable names
#   1: r.squared, 2: adj.r.squared, 3: statistic, 4: deviance, 5: df.residual
```

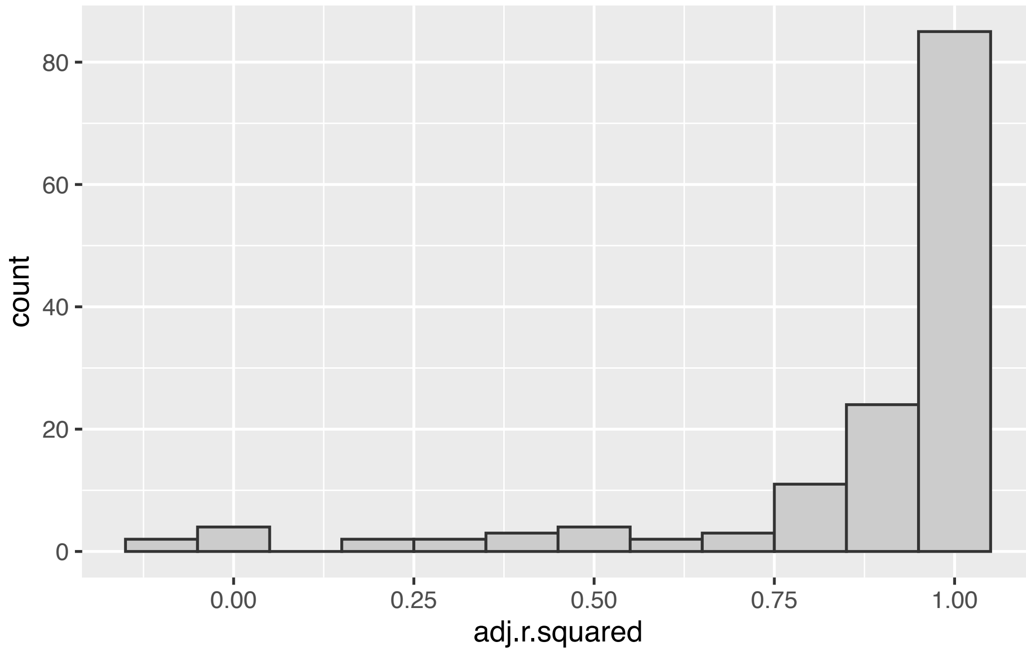
Since this gives us a data frame, `map_dfr` can combine the results into a single data frame:

```r
test_statistics
```

```
# A tibble: 142 x 12
   r.squared adj.r.squ~1 sigma stati~2  p.value    df logLik   AIC   BIC devia~3
       <dbl>       <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>
 1     0.948       0.942 1.22    181.  9.84e- 8     1 -18.3  42.7  44.1   15.0
 2     0.911       0.902 1.98    102.  1.46e- 6     1 -24.1  54.3  55.8   39.3
 3     0.985       0.984 1.32    662.  1.81e-10     1 -19.3  44.6  46.0   17.5
 4     0.888       0.877 1.41     79.1 4.59e- 6     1 -20.0  46.1  47.5   19.8
 5     0.996       0.995 0.292  2246.  4.22e-13     1  -1.17  8.35  9.80   0.854
 6     0.980       0.978 0.621   481.  8.67e-10     1 -10.2  26.4  27.9    3.85
 7     0.992       0.991 0.407  1261.  7.44e-12     1  -5.16 16.3  17.8    1.66
 8     0.967       0.963 1.64    291.  1.02e- 8     1 -21.9  49.7  51.2   26.9
 9     0.989       0.988 0.977   930.  3.37e-11     1 -15.7  37.3  38.8    9.54
10     0.995       0.994 0.293  1822.  1.20e-12     1  -1.20  8.40  9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

Now that we have run the same model for each country, we can see how well a linear model fits our data by looking at the adjusted R-squared:

```r
test_statistics %>%
  ggplot(aes(x = adj.r.squared)) +
  geom_histogram(binwidth = .1, fill = "grey80", color = "gray20")
```



Looking at the histogram, most models appear to follow a linear trend (as many countries have an r-squared of > .8), but in a few countries, life expectancy clearly did not follow a linear trend.

Unfortunately, the data frame no longer contains the countries and continents, so we cannot extract the countries that appear to be outliers of our linear trend. Fortunately, there are many other ways to achieve the same results. I call it the Wild West of split > apply > combine for running many models.

14.4 The Wild West of split > apply > combine for running many models

It turns out that there is not just one way to run many models with the Tidyverse, but many. Here's an overview of the options I found. We'll go through them in more detail in a minute.

ID	Split type	split	apply	combine
1	Groups	purrr:group_by	dplyr::group_map	purrr::map_dfr
2	Groups	purrr:group_by	dplyr::group_modify	dplyr::ungroup
3	Lists	base::split	purrr:map_dfr	-
4	Lists	base::split	purrr::map2_dfr	-
5	Lists	dplyr::group_split	purrr:map_dfr	-

ID	Split type	split	apply	combine
6	Nested data	dplyr::group_nest	dplyr::mutate + purrr::map	tidyr::unnest

First of all, we can differentiate between the three split types:

- *group*: When I talk about groups, I mean that the data is in the form of a grouped tibble.
- *lists*: By lists I mean the native list data type
- *nested_data*: By nested data, I mean nested data frames in which columns of a data frame contain lists or data frames.

Some methods combine the apply > combine step into one function (split > map_dfr and split -> map2_dfr). Therefore I added a hyphen (-) for the combine phase.

All methods have in common that a data frame comes in and a data frame goes out. What kind of data frame is returned depends on what we do in the apply phase. In our case, we chose to output the test statistics or parameters of our models. Similarly, we could also output the predictions of our models.

We will go through each of these examples next. Some of these examples use functions that are still in the experimental stage (for example, group_modify). So keep in mind that these functions may not be available forever. Either way, it's a good exercise to get familiar with the new grouping functions in dplyr.

14.5 #1 group_by > group_map > map_dfr

Our first method is using grouped data. This is how it looks like:

```
gapminder %>%
  group_by(country) %>%
  group_map(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .) %>% glance()
    ) %>%
  map_dfr(~ .)
```

```
# A tibble: 142 x 12
   r.squared adj.r.squ~1 sigma stati~2  p.value    df logLik   AIC   BIC devia~3
       <dbl>       <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>
 1     0.948       0.942 1.22     181.  9.84e- 8     1 -18.3  42.7  44.1    15.0
 2     0.911       0.902 1.98     102.  1.46e- 6     1 -24.1  54.3  55.8    39.3
```

```
 3     0.985        0.984 1.32    662.  1.81e-10     1 -19.3  44.6  46.0   17.5
 4     0.888        0.877 1.41     79.1 4.59e- 6     1 -20.0  46.1  47.5   19.8
 5     0.996        0.995 0.292  2246.  4.22e-13     1  -1.17  8.35  9.80   0.854
 6     0.980        0.978 0.621   481.  8.67e-10     1 -10.2  26.4  27.9    3.85
 7     0.992        0.991 0.407  1261.  7.44e-12     1  -5.16 16.3  17.8    1.66
 8     0.967        0.963 1.64    291.  1.02e- 8     1 -21.9  49.7  51.2   26.9
 9     0.989        0.988 0.977   930.  3.37e-11     1 -15.7  37.3  38.8    9.54
10     0.995        0.994 0.293  1822.  1.20e-12     1  -1.20  8.40  9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

As you can see, the resulting data frame does not contain a country or continent column. The first interesting bit in the code is the `group_map` function. `group_map` function takes a grouped tibble as input and outputs a list. We can see the list if we run the code without `map_dfr`:

```r
gapminder %>%
  group_by(country) %>%
  group_map(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .)
    ) %>%
  head(n = 2)
```

```
[[1]]

Call:
lm(formula = lifeExp ~ year, data = .)

Coefficients:
(Intercept)         year
  -507.5343       0.2753


[[2]]

Call:
lm(formula = lifeExp ~ year, data = .)

Coefficients:
(Intercept)         year
  -594.0725       0.3347
```

Before this function, we could not simply iterate a function over groups in a grouped tibble. Instead, we split data frames into a list and applied purrr functions to the elements of that list (e.g., `map`).

If you like, `group_map` is the dplyr version of purrr's `map` functions.

Now that we have extracted the object of each model we can use `glance` to get the test statistics of these models:

```
gapminder %>%
  group_by(country) %>%
  group_map(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .) %>%
      glance()
    ) %>%
  head(n = 2)
```

```
[[1]]
# A tibble: 1 x 12
  r.squ~1 adj.r~2 sigma stati~3 p.value    df logLik   AIC   BIC devia~4 df.re~5
    <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>   <int>
1   0.948   0.942  1.22    181. 9.84e-8     1  -18.3  42.7  44.1    15.0      10
# ... with 1 more variable: nobs <int>, and abbreviated variable names
#   1: r.squared, 2: adj.r.squared, 3: statistic, 4: deviance, 5: df.residual

[[2]]
# A tibble: 1 x 12
  r.squ~1 adj.r~2 sigma stati~3 p.value    df logLik   AIC   BIC devia~4 df.re~5
    <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>   <int>
1   0.911   0.902  1.98    102. 1.46e-6     1  -24.1  54.3  55.8    39.3      10
# ... with 1 more variable: nobs <int>, and abbreviated variable names
#   1: r.squared, 2: adj.r.squared, 3: statistic, 4: deviance, 5: df.residual
```

Since this gives us a list of data frames, we need to combine them with `map_dfr(~ .)`.

```
gapminder %>%
  group_by(country) %>%
  group_map(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .) %>% glance()
    ) %>%
```

```
  map_dfr(~ .)
```

```
# A tibble: 142 x 12
   r.squared adj.r.squ~1 sigma stati~2  p.value    df logLik   AIC   BIC devia~3
       <dbl>       <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>   <dbl>
 1     0.948       0.942 1.22    181.  9.84e- 8     1 -18.3  42.7  44.1   15.0  
 2     0.911       0.902 1.98    102.  1.46e- 6     1 -24.1  54.3  55.8   39.3  
 3     0.985       0.984 1.32    662.  1.81e-10     1 -19.3  44.6  46.0   17.5  
 4     0.888       0.877 1.41     79.1 4.59e- 6     1 -20.0  46.1  47.5   19.8  
 5     0.996       0.995 0.292  2246.  4.22e-13     1  -1.17  8.35  9.80   0.854
 6     0.980       0.978 0.621   481.  8.67e-10     1 -10.2  26.4  27.9    3.85 
 7     0.992       0.991 0.407  1261.  7.44e-12     1  -5.16 16.3  17.8    1.66 
 8     0.967       0.963 1.64    291.  1.02e- 8     1 -21.9  49.7  51.2   26.9  
 9     0.989       0.988 0.977   930.  3.37e-11     1 -15.7  37.3  38.8    9.54 
10     0.995       0.994 0.293  1822.  1.20e-12     1  -1.20  8.40  9.85   0.858
# ... with 132 more rows, 2 more variables: df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: adj.r.squared, 2: statistic, 3: deviance
```

14.6 #2 group_by > group_modify > ungroup

In our next example we use the new function `group_modify`. Compared to `group_map`, this function also accepts a grouped tibble, but outputs a grouped tibble instead of a list. With `group_modify` you have to make sure that a data frame is returned, otherwise the function will yield an error. This is what it looks like:

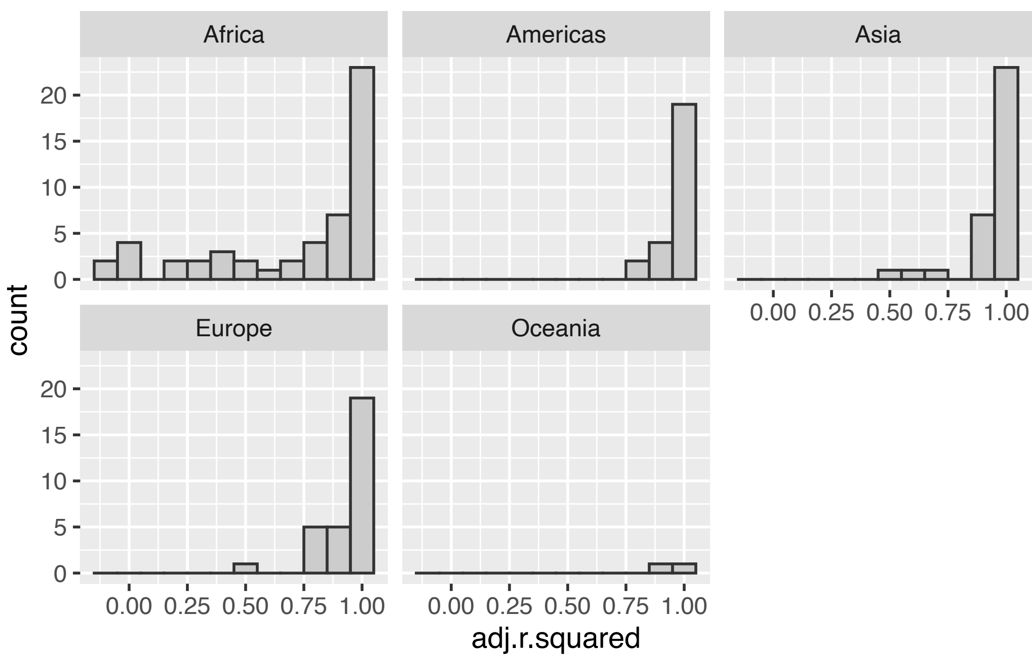
```
(method_two_results <- gapminder %>%
  group_by(country, continent) %>%
  group_modify(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .) %>% glance
    ) %>%
  ungroup())
```

```
# A tibble: 142 x 14
   country     conti~1 r.squ~2 adj.r~3 sigma stati~4  p.value    df logLik   AIC
   <fct>       <fct>     <dbl>   <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl>
 1 Afghanistan Asia      0.948   0.942 1.22    181.  9.84e- 8     1 -18.3   42.7
 2 Albania     Europe    0.911   0.902 1.98    102.  1.46e- 6     1 -24.1   54.3
 3 Algeria     Africa    0.985   0.984 1.32    662.  1.81e-10     1 -19.3   44.6
```

```
 4 Angola      Africa    0.888   0.877 1.41     79.1 4.59e- 6     1 -20.0  46.1
 5 Argentina   Americ~   0.996   0.995 0.292  2246.  4.22e-13     1  -1.17  8.35
 6 Australia   Oceania   0.980   0.978 0.621   481.  8.67e-10     1 -10.2  26.4
 7 Austria     Europe    0.992   0.991 0.407  1261.  7.44e-12     1  -5.16 16.3
 8 Bahrain     Asia      0.967   0.963 1.64    291.  1.02e- 8     1 -21.9  49.7
 9 Bangladesh  Asia      0.989   0.988 0.977   930.  3.37e-11     1 -15.7  37.3
10 Belgium     Europe    0.995   0.994 0.293  1822.  1.20e-12     1  -1.20  8.40
# ... with 132 more rows, 4 more variables: BIC <dbl>, deviance <dbl>,
#   df.residual <int>, nobs <int>, and abbreviated variable names 1: continent,
#   2: r.squared, 3: adj.r.squared, 4: statistic
```

The great advantage of this method is that we can keep the country and continent as columns. With these results, we can compare, for example, the adjusted R-squared between the five continents:

```
method_two_results %>%
  ggplot(aes(x = adj.r.squared)) +
  geom_histogram(binwidth = .1, fill = "grey80", color = "gray20") +
  facet_wrap(vars(continent))
```



The results clearly show that our linear models perform least well in African countries (since these countries have small R-squared values).

A look at the data shows us that the countries with the worst fit are Rwanda, Botswana, Zimbabwe, Zambia and Swaziland:

```r
method_two_results %>%
  slice_min(adj.r.squared, n = 5)
```

```
# A tibble: 5 x 14
  country conti~1 r.squ~2 adj.r~3 sigma stati~4 p.value    df logLik   AIC   BIC
  <fct>   <fct>     <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>
1 Rwanda  Africa   0.0172 -0.0811  6.56   0.175   0.685     1  -38.5  83.0  84.5
2 Botswa~ Africa   0.0340 -0.0626  6.11   0.352   0.566     1  -37.7  81.3  82.8
3 Zimbab~ Africa   0.0562 -0.0381  7.21   0.596   0.458     1  -39.6  85.3  86.7
4 Zambia  Africa   0.0598 -0.0342  4.53   0.636   0.444     1  -34.1  74.1  75.6
5 Swazil~ Africa   0.0682 -0.0250  6.64   0.732   0.412     1  -38.7  83.3  84.8
# ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>, and
#   abbreviated variable names 1: continent, 2: r.squared, 3: adj.r.squared,
#   4: statistic
```

The beauty of keeping the country and continent columns is that we can now compare the fit of our model to the actual life expectancies in these countries.

To get this data, we can use the `augment` function from the broom package. For each country, the function returns the fitted values of our dependent variable (in our case life expectancy), the residuals and the actual values of the dependent variable:

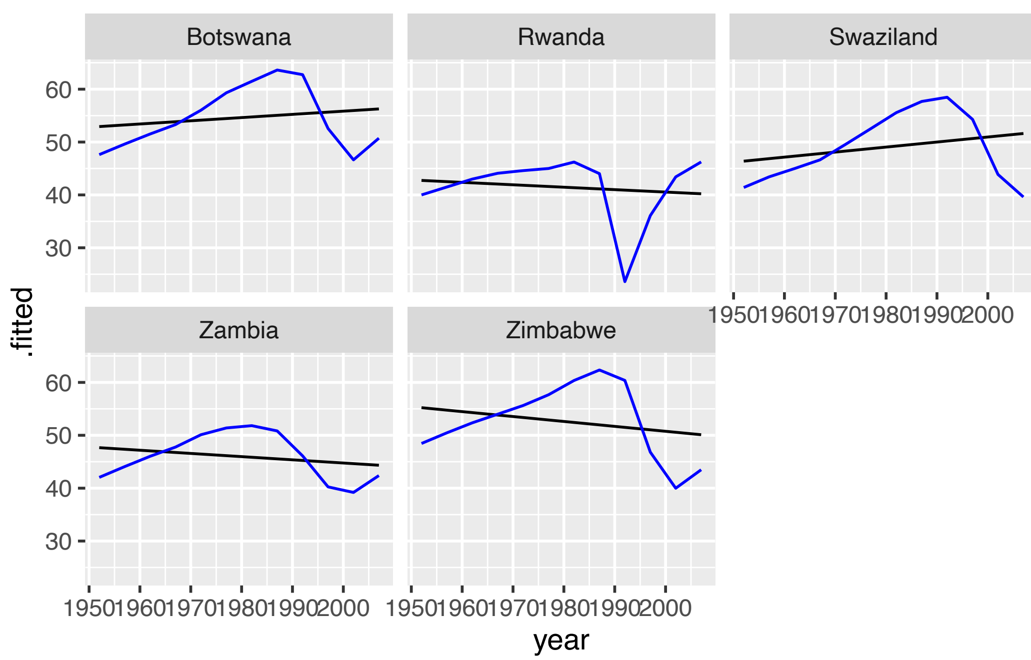
```r
(augmented_data <- gapminder %>%
  group_by(country, continent) %>%
  group_modify(
    .data = .,
    .f    = ~ lm(lifeExp ~ year, data = .) %>% augment()
    ) %>%
  ungroup())
```

```
# A tibble: 1,704 x 10
   country   conti~1 lifeExp  year .fitted  .resid   .hat .sigma .cooksd .std.~2
   <fct>     <fct>     <dbl> <int>   <dbl>   <dbl>  <dbl>  <dbl>   <dbl>   <dbl>
 1 Afghanis~ Asia       28.8  1952    29.9 -1.11   0.295    1.21 2.43e-1 -1.08
 2 Afghanis~ Asia       30.3  1957    31.3 -0.952  0.225    1.24 1.13e-1 -0.884
 3 Afghanis~ Asia       32.0  1962    32.7 -0.664  0.169    1.27 3.60e-2 -0.595
 4 Afghanis~ Asia       34.0  1967    34.0 -0.0172 0.127    1.29 1.65e-5 -0.0151
 5 Afghanis~ Asia       36.1  1972    35.4  0.674  0.0991   1.27 1.85e-2  0.581
```

```
 6 Afghanis~ Asia       38.4  1977    36.8  1.65   0.0851   1.15 9.23e-2  1.41 
 7 Afghanis~ Asia       39.9  1982    38.2  1.69   0.0851   1.15 9.67e-2  1.44 
 8 Afghanis~ Asia       40.8  1987    39.5  1.28   0.0991   1.21 6.67e-2  1.10 
 9 Afghanis~ Asia       41.7  1992    40.9  0.754  0.127    1.26 3.17e-2  0.660
10 Afghanis~ Asia       41.8  1997    42.3 -0.534  0.169    1.27 2.33e-2 -0.479
# ... with 1,694 more rows, and abbreviated variable names 1: continent,
#   2: .std.resid
```

Now we are ready to compare our regression models for these five countries with the actual data:

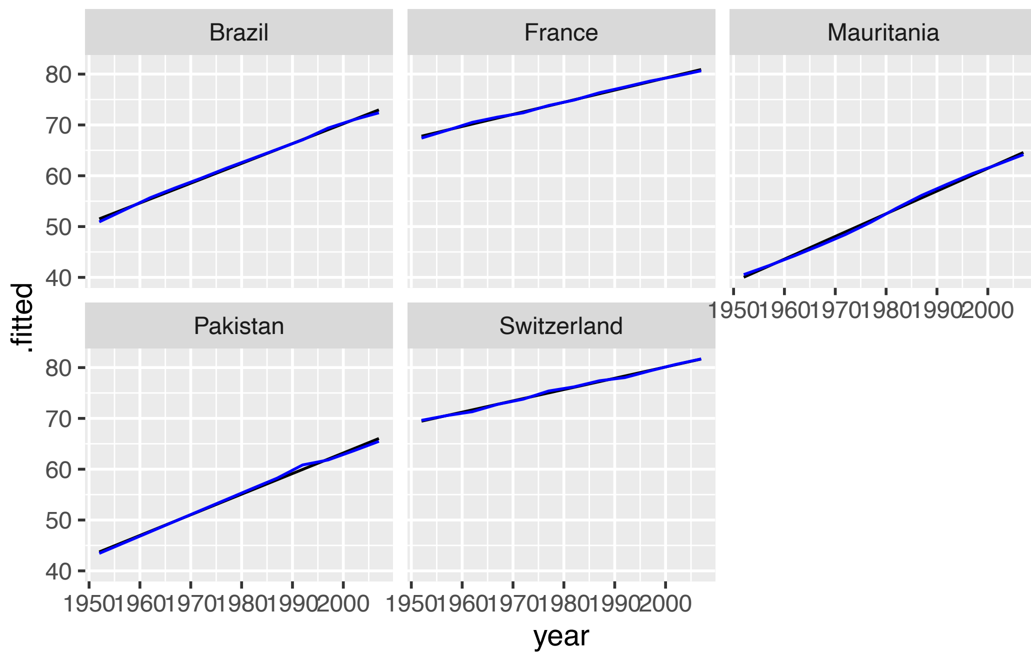
```
augmented_data %>%
  filter(country %in% (
    method_two_results %>% slice_min(adj.r.squared, n = 5) %>%
      pull(country)
  )) %>%
  ggplot(aes(x = year, y = .fitted)) +
  geom_line() +
  geom_line(aes(y = .fitted + .resid), color = "blue") +
  facet_wrap(vars(country))
```



The blue line represents the actual data and the black line our fitted regression model.

Similarly, we can look at the countries with the best fit (`slice_max` instead of `slice_min`):

```
augmented_data %>%
  filter(country %in% (
    method_two_results %>% slice_max(adj.r.squared, n = 5) %>%
      pull(country)
  )) %>%
  ggplot(aes(x = year, y = .fitted)) +
  geom_line() +
  geom_line(aes(y = .fitted + .resid), color = "blue") +
  facet_wrap(vars(country))
```



14.7 #4 split > map2_dfr

We have already covered on how to run many models using `split` and `map_dfr`. However, we have seen that this method does not preserve the country column. However, there is a trick to preserve it. And the trick is the function `map2_dfr`.

You may know that the function `names` returns the names of list objects:

```r
my_list <- list(
  Afghanistan = c(1, 2, 3),
  Germany     = 3
)

my_list %>% names()
```

```
[1] "Afghanistan" "Germany"
```

You may also know that `map2_dfr` allows us to iterate over two arguments simultaneously. By combining both functions we can add a new column to the data frame returned by the `tidy`, `glance` or `augment` function, representing the country column:

```r
gapminder %>%
  split(.$country) %>%
  map2_dfr(
    .x = .,
    .y = names(.),
    .f = ~ lm(lifeExp ~ year, data = .x) %>%
      tidy() %>% mutate(country = .y)
  )
```

```
# A tibble: 284 x 6
   term         estimate std.error statistic  p.value country
   <chr>           <dbl>     <dbl>     <dbl>    <dbl> <chr>
 1 (Intercept)  -508.      40.5        -12.5  1.93e- 7 Afghanistan
 2 year            0.275    0.0205      13.5  9.84e- 8 Afghanistan
 3 (Intercept)  -594.      65.7         -9.05 3.94e- 6 Albania
 4 year            0.335    0.0332      10.1  1.46e- 6 Albania
 5 (Intercept) -1068.      43.8        -24.4  3.07e-10 Algeria
 6 year            0.569    0.0221      25.7  1.81e-10 Algeria
 7 (Intercept)  -377.      46.6         -8.08 1.08e- 5 Angola
 8 year            0.209    0.0235       8.90 4.59e- 6 Angola
 9 (Intercept)  -390.       9.68       -40.3  2.14e-12 Argentina
10 year            0.232    0.00489     47.4  4.22e-13 Argentina
# ... with 274 more rows
```

With this method we keep the country name but not the continent column since we have split the data frame into a one-dimensional list.

However, we can use another trick to keep both columns. We know that the data frame we split still contains the country and continent columns. We also know that both columns contain the same values per column. We can therefore add the country and continent to the data frame returned by the `tidy` function. Also we don't even need `map2_dfr` and can use `map_dfr` instead.

```
gapminder %>%
  split(.$country) %>%
  map_dfr(
    .x = .,
    .f = ~ lm(lifeExp ~ year, data = .x) %>%
      tidy() %>%
      mutate(
        country = .x$country[1],
        continent = .x$continent[1])
  )
```

```
# A tibble: 284 x 7
   term         estimate std.error statistic  p.value country     continent
   <chr>           <dbl>     <dbl>     <dbl>    <dbl> <fct>       <fct>
 1 (Intercept)  -508.     40.5        -12.5  1.93e- 7 Afghanistan Asia
 2 year            0.275   0.0205      13.5  9.84e- 8 Afghanistan Asia
 3 (Intercept)  -594.     65.7         -9.05 3.94e- 6 Albania     Europe
 4 year            0.335   0.0332      10.1  1.46e- 6 Albania     Europe
 5 (Intercept) -1068.     43.8        -24.4  3.07e-10 Algeria     Africa
 6 year            0.569   0.0221      25.7  1.81e-10 Algeria     Africa
 7 (Intercept)  -377.     46.6         -8.08 1.08e- 5 Angola      Africa
 8 year            0.209   0.0235       8.90 4.59e- 6 Angola      Africa
 9 (Intercept)  -390.      9.68       -40.3  2.14e-12 Argentina   Americas
10 year            0.232   0.00489     47.4  4.22e-13 Argentina   Americas
# ... with 274 more rows
```

14.8 #5 group_split > map_dfr

The combination of `group_split` and `map_dfr` is just an alternative version of `split` and `map_dfr`. `group_split` was introduced in dplyr for version 0.8.0 in 2019. It is still in an experimental state. The difference with `split` is that `group_split` does not name the elements of the returned list. Other than that, I don't see a good use case for using `group_split` over `split` for our use case.

Here is how it works:

```r
gapminder %>%
  group_split(country) %>%
  map_dfr(
    .x = .,
    .f = ~ lm(lifeExp ~ year, data = .x) %>%
      tidy() %>%
      mutate(
        country = .x$country[1],
        continent = .x$continent[1])
  )
```

```
# A tibble: 284 x 7
   term        estimate std.error statistic  p.value country     continent
   <chr>          <dbl>     <dbl>     <dbl>    <dbl> <fct>       <fct>
 1 (Intercept)  -508.     40.5        -12.5  1.93e- 7 Afghanistan Asia
 2 year            0.275   0.0205      13.5  9.84e- 8 Afghanistan Asia
 3 (Intercept)  -594.     65.7         -9.05 3.94e- 6 Albania     Europe
 4 year            0.335   0.0332      10.1  1.46e- 6 Albania     Europe
 5 (Intercept) -1068.     43.8        -24.4  3.07e-10 Algeria     Africa
 6 year            0.569   0.0221      25.7  1.81e-10 Algeria     Africa
 7 (Intercept)  -377.     46.6         -8.08 1.08e- 5 Angola      Africa
 8 year            0.209   0.0235       8.90 4.59e- 6 Angola      Africa
 9 (Intercept)  -390.      9.68       -40.3  2.14e-12 Argentina   Americas
10 year            0.232   0.00489     47.4  4.22e-13 Argentina   Americas
# ... with 274 more rows
```

14.9 #6 group_nest > mutate/map > unnest

At last we have the combination of `group_nest`, `mutate` and `map` and `unnest`. I'll show you the code first and we'll go through it afterwards:

```r
gapminder %>%
  group_nest(continent, country) %>%
  mutate(
    model = map(data, ~ lm(lifeExp ~ year, data = .) %>%
      tidy())
  ) %>%
  select(-data) %>%
  unnest(model)
```

```
# A tibble: 284 x 7
   continent country      term        estimate std.error statistic  p.value
   <fct>     <fct>        <chr>          <dbl>     <dbl>     <dbl>    <dbl>
 1 Africa    Algeria      (Intercept) -1068.     43.8      -24.4   3.07e-10
 2 Africa    Algeria      year            0.569   0.0221    25.7   1.81e-10
 3 Africa    Angola       (Intercept)  -377.     46.6       -8.08  1.08e- 5
 4 Africa    Angola       year            0.209   0.0235     8.90  4.59e- 6
 5 Africa    Benin        (Intercept)  -613.     38.9      -15.8   2.18e- 8
 6 Africa    Benin        year            0.334   0.0196    17.0   1.04e- 8
 7 Africa    Botswana     (Intercept)   -65.5   202.        -0.324 7.53e- 1
 8 Africa    Botswana     year            0.0607  0.102      0.593 5.66e- 1
 9 Africa    Burkina Faso (Intercept)  -676.     67.8       -9.97  1.63e- 6
10 Africa    Burkina Faso year            0.364   0.0342    10.6   9.05e- 7
# ... with 274 more rows
```

Let's break it down. The **group_nest** function was also introduced with the dplyr version 0.8.0 and works pretty similar to **nest**. Compared to **nest**, with **group_nest** you name the columns that should *not* be nested instead of the columns that should be nested.

```
gapminder %>%
  group_nest(continent, country)
```

```
# A tibble: 142 x 3
   continent country                                data
   <fct>     <fct>                    <list<tibble[,4]>>
 1 Africa    Algeria                            [12 x 4]
 2 Africa    Angola                             [12 x 4]
 3 Africa    Benin                              [12 x 4]
 4 Africa    Botswana                           [12 x 4]
 5 Africa    Burkina Faso                       [12 x 4]
 6 Africa    Burundi                            [12 x 4]
 7 Africa    Cameroon                           [12 x 4]
 8 Africa    Central African Republic           [12 x 4]
 9 Africa    Chad                               [12 x 4]
10 Africa    Comoros                            [12 x 4]
# ... with 132 more rows
```

Next, we create a new column that contains our model results:

```
(model_results_nested <- gapminder %>%
  group_nest(continent, country) %>%
```

```
  mutate(
    model = map(data, ~ lm(lifeExp ~ year, data = .) %>%
      tidy())
  ))
```

```
# A tibble: 142 x 4
   continent country                                    data model
   <fct>     <fct>                       <list<tibble[,4]>> <list>
 1 Africa    Algeria                               [12 x 4] <tibble [2 x 5]>
 2 Africa    Angola                                [12 x 4] <tibble [2 x 5]>
 3 Africa    Benin                                 [12 x 4] <tibble [2 x 5]>
 4 Africa    Botswana                              [12 x 4] <tibble [2 x 5]>
 5 Africa    Burkina Faso                          [12 x 4] <tibble [2 x 5]>
 6 Africa    Burundi                               [12 x 4] <tibble [2 x 5]>
 7 Africa    Cameroon                              [12 x 4] <tibble [2 x 5]>
 8 Africa    Central African Republic              [12 x 4] <tibble [2 x 5]>
 9 Africa    Chad                                  [12 x 4] <tibble [2 x 5]>
10 Africa    Comoros                               [12 x 4] <tibble [2 x 5]>
# ... with 132 more rows
```

Each value of the model column contains the results of the `tidy` function. For example, the model results for Algeria:

```
model_results_nested$model[[1]]
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) -1068.      43.8       -24.4 3.07e-10
2 year            0.569    0.0221     25.7 1.81e-10
```

A nice feature of this method is that we can store the results of the `tidy` and `glance` functions in one data frame:

```
gapminder %>%
  group_nest(continent, country) %>%
  mutate(
    model                 = map(data, ~ lm(lifeExp ~ year, data = .)),
    model_parameters      = map(model, broom::tidy),
    model_test_statistics = map(model, broom::glance)
```

```
)
```

```
# A tibble: 142 x 6
   continent country                              data model model_~1 model_~2
   <fct>     <fct>                    <list<tibble[,4]>> <lis> <list>   <list>
 1 Africa    Algeria                           [12 x 4] <lm>  <tibble> <tibble>
 2 Africa    Angola                            [12 x 4] <lm>  <tibble> <tibble>
 3 Africa    Benin                             [12 x 4] <lm>  <tibble> <tibble>
 4 Africa    Botswana                          [12 x 4] <lm>  <tibble> <tibble>
 5 Africa    Burkina Faso                      [12 x 4] <lm>  <tibble> <tibble>
 6 Africa    Burundi                           [12 x 4] <lm>  <tibble> <tibble>
 7 Africa    Cameroon                          [12 x 4] <lm>  <tibble> <tibble>
 8 Africa    Central African Republic          [12 x 4] <lm>  <tibble> <tibble>
 9 Africa    Chad                              [12 x 4] <lm>  <tibble> <tibble>
10 Africa    Comoros                           [12 x 4] <lm>  <tibble> <tibble>
# ... with 132 more rows, and abbreviated variable names 1: model_parameters,
#   2: model_test_statistics
```

Once we have this data frame, we can use `unnest` and `select` to unpack the results of our models:

```
gapminder %>%
  group_nest(continent, country) %>%
  mutate(
    model                 = map(data, ~ lm(lifeExp ~ year, data = .)),
    model_parameters      = map(model, broom::tidy),
    model_test_statistics = map(model, broom::glance)
  ) %>%
  select(-model, -model_parameters, -data) %>%
  unnest(model_test_statistics)
```

```
# A tibble: 142 x 14
   continent country   r.squ~1 adj.r~2 sigma stati~3  p.value    df logLik   AIC
   <fct>     <fct>       <dbl>   <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl>
 1 Africa    Algeria    0.985   0.984   1.32 6.62e+2 1.81e-10     1 -19.3   44.6
 2 Africa    Angola     0.888   0.877   1.41 7.91e+1 4.59e- 6     1 -20.0   46.1
 3 Africa    Benin      0.967   0.963   1.17 2.89e+2 1.04e- 8     1 -17.9   41.7
 4 Africa    Botswana   0.0340 -0.0626  6.11 3.52e-1 5.66e- 1     1 -37.7   81.3
 5 Africa    Burkina ~  0.919   0.911   2.05 1.13e+2 9.05e- 7     1 -24.5   55.1
 6 Africa    Burundi    0.766   0.743   1.61 3.27e+1 1.93e- 4     1 -21.7   49.3
 7 Africa    Cameroon   0.680   0.648   3.24 2.13e+1 9.63e- 4     1 -30.1   66.1
```

```
 8 Africa    Central ~  0.493   0.443  3.52  9.73e+0 1.09e- 2     1 -31.1   68.1
 9 Africa    Chad       0.872   0.860  1.83  6.84e+1 8.82e- 6     1 -23.2   52.4
10 Africa    Comoros    0.997   0.997  0.479 3.17e+3 7.63e-14     1  -7.09  20.2
# ... with 132 more rows, 4 more variables: BIC <dbl>, deviance <dbl>,
#   df.residual <int>, nobs <int>, and abbreviated variable names 1: r.squared,
#   2: adj.r.squared, 3: statistic
```

14.10 Conclusion

We have seen that there are a few ways to run many models using a number of functions from the Tidyverse package. Some of these methods make it more difficult than others to extract the names of groups or splits. Time will tell which approach works best for most people. In any case, some methods look promising. I was particularly impressed with the combination of `group_by`, `group_modify` and `ungroup`. Keep in mind, however, that many of the new grouping functions introduced in dplyr in 2019 are still in the experimental stage. They may not be with us forever.

> **Summary**
>
> Here's what you can take away from this tutorial.
>
> - There are three approaches to running many models. Those based on grouped tibbles, those based on lists and those based on nested data.
> - The combination of `group_by`, `group_modify` and `ungroup` seems to be one of the most elegant ways to run many models.
> - Many grouping functions that were introduced in dplyr in 2019 are still in an experimental stage and should be used with caution.