17 How to make a data frame wider

What will this tutorial cover?

In this tutorial, you will learn how to make data frames wider. Because this technique reduces the number of values in a data frame, it is often useful when you need to make your data frames human readable. Other use cases are also discussed: For example, how to use the technique for feature engineering in machine learning and how to deal with certain challenges.

Who do I have to thank?

Before writing this tutorial, I asked the Twitter community about specific challenges they have with the `pivot_wider` function. I would like to thank everyone who provided answers. In particular, the following people who inspired me to write this tutorial: Cole, Eric Stewart, Guillaume Loignon, Saurav Ghosh, Marc-Aurèle Rivière, neregauzak, John Paul Helveston.
I also have to thank the developers of the tidyverse documentation. I took some of the ideas from their official documentation of pivot_wider.

In the last tutorial, we said that many datasets are made longer in order to make them tidy. Tidy data is machine readable (thanks to Cole for this insight) in that it is optimized to be processed by data analytics tools. As a result, longer data frames have more values than shorter ones and they are easier to analyze with R.

Sometimes we want to do the opposite and make data frames wider. On Twitter, I asked the community to share their use cases. A good summary of most use cases is that wider data frames are more readable for humans. Either cognitively or for presentations. Here are the use cases we will cover in this tutorial based on the conversation on Twitter:

- How to use 'pivot_wider´ (the simplest example)
- How to use pivot_wider to calculate ratios/percentages (Julio and Eric Stewart)
- How to use pivot_wider to create tables of summary statistics (Proposal by Guillaume Loignon).
- How to make data frames wider for use in other software tools
- How to use pivot_wider to one-hot encode a factor (Saurav Ghosh, Marc-Aurèle Rivière)
- How to deal with multiple variable names stored in a column (neregauzak)

- How to pivot_wider without an id column

In the following chapters, we will go through each of these use cases in detail. All of the cases have a few things in common. They extend the data frame by increasing the number of columns and decreasing the number of rows. Also, each use case can be implemented with the `pivot_wider` function.

17.1 How to use `pivot_wider` (the simplest example)

The `fish_encounters` data frame contains information about various stations that monitor and record the amount of fish passing through these stations downstream.

```
fish_encounters
```

```
# A tibble: 114 x 3
   fish  station  seen
   <fct> <fct>   <int>
 1 4842  Release     1
 2 4842  I80_1       1
 3 4842  Lisbon      1
 4 4842  Rstr        1
 5 4842  Base_TD     1
 6 4842  BCE         1
 7 4842  BCW         1
 8 4842  BCE2        1
 9 4842  BCW2        1
10 4842  MAE         1
# ... with 104 more rows
```

The data frame has three columns. `fish` is an identifier for specific fish species. `station` is the name of the measuring station. `seen` indicates whether a fish was seen (1 if yes) or not seen (NA if not) at this station.

Suppose you would like to make this data frame wider because you would like to present the results in a human-readable table. To do this, you can use `pivot_wider` and provide arguments for its main parameters:

- `id_cols`: These columns are the identifiers for the observations. These column names remain unchanged in the data frame. Their values form the rows of the transformed data frame. By default, all columns except those specified in `names_from` and `values_from` become `id_cols`.

- `names_from`: These columns will be transformed into a wider format. Their values will be converted to columns. If you specify more than one column for `names_from`, the newly created column names will be a combination of the column values.
- `values_from`: The values of these columns will be used for the columns created with `names_from`.

Here is how the function looks like in action:

```r
(fish_encounters_wide <- fish_encounters %>%
  pivot_wider(
     # this column will be kept in the new data frame
    id_cols = fish,
    # the values of this column will be the new column names
    names_from = station,
    # the values of these column will be used for the newly
    # created columns
    values_from = seen
  ))
```

```
# A tibble: 19 x 12
   fish  Release I80_1 Lisbon  Rstr Base_TD   BCE   BCW  BCE2  BCW2   MAE   MAW
   <fct>   <int> <int>  <int> <int>   <int> <int> <int> <int> <int> <int> <int>
 1 4842        1     1      1     1       1     1     1     1     1     1     1
 2 4843        1     1      1     1       1     1     1     1     1     1     1
 3 4844        1     1      1     1       1     1     1     1     1     1     1
 4 4845        1     1      1     1       1    NA    NA    NA    NA    NA    NA
 5 4847        1     1      1    NA      NA    NA    NA    NA    NA    NA    NA
 6 4848        1     1      1     1      NA    NA    NA    NA    NA    NA    NA
 7 4849        1     1     NA    NA      NA    NA    NA    NA    NA    NA    NA
 8 4850        1     1     NA     1       1     1     1    NA    NA    NA    NA
 9 4851        1     1     NA    NA      NA    NA    NA    NA    NA    NA    NA
10 4854        1     1     NA    NA      NA    NA    NA    NA    NA    NA    NA
11 4855        1     1      1     1       1    NA    NA    NA    NA    NA    NA
12 4857        1     1      1     1       1     1     1     1     1    NA    NA
13 4858        1     1      1     1       1     1     1     1     1     1     1
14 4859        1     1      1     1       1    NA    NA    NA    NA    NA    NA
15 4861        1     1      1     1       1     1     1     1     1     1     1
16 4862        1     1      1     1       1     1     1     1     1    NA    NA
17 4863        1     1     NA    NA      NA    NA    NA    NA    NA    NA    NA
18 4864        1     1     NA    NA      NA    NA    NA    NA    NA    NA    NA
19 4865        1     1      1    NA      NA    NA    NA    NA    NA    NA    NA
```

A small improvement to the data frame could be to prefix the new column names:

```
fish_encounters %>%
  pivot_wider(
    id_cols = fish,
    names_from = station,
    values_from = seen,
    # The prefix is added to each newly created column
    names_prefix = "station_"
  )
```

```
# A tibble: 19 x 12
   fish  stati~1 stati~2 stati~3 stati~4 stati~5 stati~6 stati~7 stati~8 stati~9
   <fct>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>
 1 4842        1       1       1       1       1       1       1       1       1
 2 4843        1       1       1       1       1       1       1       1       1
 3 4844        1       1       1       1       1       1       1       1       1
 4 4845        1       1       1       1       1      NA      NA      NA      NA
 5 4847        1       1       1      NA      NA      NA      NA      NA      NA
 6 4848        1       1       1       1      NA      NA      NA      NA      NA
 7 4849        1       1      NA      NA      NA      NA      NA      NA      NA
 8 4850        1       1      NA       1       1       1       1      NA      NA
 9 4851        1       1      NA      NA      NA      NA      NA      NA      NA
10 4854        1       1      NA      NA      NA      NA      NA      NA      NA
11 4855        1       1       1       1       1      NA      NA      NA      NA
12 4857        1       1       1       1       1       1       1       1       1
13 4858        1       1       1       1       1       1       1       1       1
14 4859        1       1       1       1       1      NA      NA      NA      NA
15 4861        1       1       1       1       1       1       1       1       1
16 4862        1       1       1       1       1       1       1       1       1
17 4863        1       1      NA      NA      NA      NA      NA      NA      NA
18 4864        1       1      NA      NA      NA      NA      NA      NA      NA
19 4865        1       1       1      NA      NA      NA      NA      NA      NA
# ... with 2 more variables: station_MAE <int>, station_MAW <int>, and
#   abbreviated variable names 1: station_Release, 2: station_I80_1,
#   3: station_Lisbon, 4: station_Rstr, 5: station_Base_TD, 6: station_BCE,
#   7: station_BCW, 8: station_BCE2, 9: station_BCW2
```

Another way to do this would be to use `names_glue`:

```
fish_encounters %>%
  pivot_wider(
    id_cols = fish,
```

```
    names_from = station,
    values_from = seen,
    # The prefix is added to each newly created column
    names_glue = "station_{station}"
  )
```

```
# A tibble: 19 x 12
   fish  stati~1 stati~2 stati~3 stati~4 stati~5 stati~6 stati~7 stati~8 stati~9
   <fct>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>
 1 4842        1       1       1       1       1       1       1       1       1
 2 4843        1       1       1       1       1       1       1       1       1
 3 4844        1       1       1       1       1       1       1       1       1
 4 4845        1       1       1       1       1      NA      NA      NA      NA
 5 4847        1       1       1      NA      NA      NA      NA      NA      NA
 6 4848        1       1       1       1      NA      NA      NA      NA      NA
 7 4849        1       1      NA      NA      NA      NA      NA      NA      NA
 8 4850        1       1      NA       1       1       1       1      NA      NA
 9 4851        1       1      NA      NA      NA      NA      NA      NA      NA
10 4854        1       1      NA      NA      NA      NA      NA      NA      NA
11 4855        1       1       1       1       1      NA      NA      NA      NA
12 4857        1       1       1       1       1       1       1       1       1
13 4858        1       1       1       1       1       1       1       1       1
14 4859        1       1       1       1       1      NA      NA      NA      NA
15 4861        1       1       1       1       1       1       1       1       1
16 4862        1       1       1       1       1       1       1       1       1
17 4863        1       1      NA      NA      NA      NA      NA      NA      NA
18 4864        1       1      NA      NA      NA      NA      NA      NA      NA
19 4865        1       1       1      NA      NA      NA      NA      NA      NA
# ... with 2 more variables: station_MAE <int>, station_MAW <int>, and
#   abbreviated variable names 1: station_Release, 2: station_I80_1,
#   3: station_Lisbon, 4: station_Rstr, 5: station_Base_TD, 6: station_BCE,
#   7: station_BCW, 8: station_BCE2, 9: station_BCW2
```

`names_glue` takes a string with curly braces. Inside the curly braces you put the columns from `names_from`. The content inside the braces is replaced by the new column names.

17.2 How to use `pivot_wider` to calculate ratios/percentages

Suppose you have obtained the following data set on the rents and incomes of U.S. residents in U.S. states:

```
us_rent_income
```

```
# A tibble: 104 x 5
   GEOID NAME       variable estimate   moe
   <chr> <chr>      <chr>       <dbl> <dbl>
 1 01    Alabama    income      24476   136
 2 01    Alabama    rent          747     3
 3 02    Alaska     income      32940   508
 4 02    Alaska     rent         1200    13
 5 04    Arizona    income      27517   148
 6 04    Arizona    rent          972     4
 7 05    Arkansas   income      23789   165
 8 05    Arkansas   rent          709     5
 9 06    California income      29454   109
10 06    California rent         1358     3
# ... with 94 more rows
```

The variable `variable` contains two values: `income` and `rent`. The actual estimated rent and the estimated income are stored in the variable `estimate`. The value for `income` indicates the mean annual income. The values for `rent` indicate the mean monthly income. `moe` indicates the margin of error for these values.

Clearly, this data frame is un-tidy as `income` and `rent` should be in two columns. Let's say you want to find out what percentage of their income people in different states have left for their rent. A suboptimal way would be a combination of `mutate` with `case_when` and `lead`:

```
us_rent_income %>%
  select(-moe) %>%
  mutate(
    # Standardize both values -> median yearly income/rent
    estimate = case_when(
      variable == "income" ~ estimate,
      variable == "rent"   ~ estimate * 12
    ),
    lead_estimate = lead(estimate),
    rent_percentage = (lead_estimate / estimate) * 100
  )
```

```
# A tibble: 104 x 6
   GEOID NAME       variable estimate lead_estimate rent_percentage
   <chr> <chr>      <chr>       <dbl>         <dbl>           <dbl>
```

```
 1 01    Alabama    income     24476          8964            36.6
 2 01    Alabama    rent        8964         32940           367.
 3 02    Alaska     income     32940         14400            43.7
 4 02    Alaska     rent       14400         27517           191.
 5 04    Arizona    income     27517         11664            42.4
 6 04    Arizona    rent       11664         23789           204.
 7 05    Arkansas   income     23789          8508            35.8
 8 05    Arkansas   rent        8508         29454           346.
 9 06    California income     29454         16296            55.3
10 06    California rent       16296         32401           199.
# ... with 94 more rows
```

The percentages we were looking for can be found in the `rent_percentage` column. Their values tell us two things: What percentage of rent people keep relative to their previous income, and what percentage of their income was relative to their rent. Again, we created an un-tidy set. Another problem with this approach is that we make an assumption with `lead`. We assume that the values `income` and `rent` alternate in the `variable` column. We could prove this, but it requires an unnecessary amount of work.

A better option is to make the data frame wider and calculate the percentage from the wider data set:

```
us_rent_income %>%
  pivot_wider(
    id_cols = c(GEOID, NAME),
    names_from = "variable",
    values_from = "estimate"
  ) %>%
  mutate(
    rent = rent * 12,
    percentage_of_rent = (rent / income) * 100
  )
```

```
# A tibble: 52 x 5
   GEOID NAME                    income  rent percentage_of_rent
   <chr> <chr>                    <dbl> <dbl>              <dbl>
 1 01    Alabama                  24476  8964               36.6
 2 02    Alaska                   32940 14400               43.7
 3 04    Arizona                  27517 11664               42.4
 4 05    Arkansas                 23789  8508               35.8
 5 06    California               29454 16296               55.3
 6 08    Colorado                 32401 13500               41.7
```

```
 7 09    Connecticut          35326 13476                38.1
 8 10    Delaware             31560 12912                40.9
 9 11    District of Columbia 43198 17088                39.6
10 12    Florida              25952 12924                49.8
# ... with 42 more rows
```

This approach has three advantages. First, we obtain a tidy data set. Second, we do not depend on the assumption that the `income` and `rent` values alternate. Third, it is less cognitively demanding. Since each column contains a variable, we don't need to worry about what those values represent. We simply divide one value by the other and multiply by 100.

17.3 How to use `pivot_wider` to create tables of summary statistics

Summary statistics are usually presented in papers, posters, and presentations. Since there is a limited amount of space available in these formats, they are presented in wider tables. As you may have heard already, wider data frames have fewer values than longer ones. In the next example, we will reduce the number of values from 105 to 40 by making the data frame wider. In other words, we're making the summary statistics of the data more human readable.

Diamonds can be described by different characteristics. The `cut` of a diamond can have different qualities (Fair, Good, Very Good Premium, Ideal). The `color` of a diamond can be categorized by the letters D (best) to J (worst). A diamond with color "D" is completely colorless. A diamond with the color "J" has some color and would therefore be of lower quality.

Suppose you want to plot the average prices of diamonds with different cuts and colors. You calculate them with `group_by` and `summarise`:

```
(means_diamonds <- diamonds %>%
  group_by(cut, color) %>%
  summarise(
    mean = mean(price)
  ))
```

```
# A tibble: 35 x 3
# Groups:   cut [5]
   cut   color  mean
   <ord> <ord> <dbl>
 1 Fair  D     4291.
 2 Fair  E     3682.
```

```
 3 Fair   F       3827.
 4 Fair   G       4239.
 5 Fair   H       5136.
 6 Fair   I       4685.
 7 Fair   J       4976.
 8 Good   D       3405.
 9 Good   E       3424.
10 Good   F       3496.
# ... with 25 more rows
```

As you can see, the data frame has $35 * 3 = 105$ values. Not only can we reduce the data frame to 40 values, but we can also make it more readable so that readers can find each value quickly. Let's transform the data frame with `pivot_wider`:

```
means_diamonds %>%
  pivot_wider(
    id_cols = cut,
    names_from = color,
    values_from = mean,
    names_prefix = "mean_"
  )
```

```
# A tibble: 5 x 8
# Groups:   cut [5]
  cut       mean_D mean_E mean_F mean_G mean_H mean_I mean_J
  <ord>      <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Fair       4291.  3682.  3827.  4239.  5136.  4685.  4976.
2 Good       3405.  3424.  3496.  4123.  4276.  5079.  4574.
3 Very Good  3470.  3215.  3779.  3873.  4535.  5256.  5104.
4 Premium    3631.  3539.  4325.  4501.  5217.  5946.  6295.
5 Ideal      2629.  2598.  3375.  3721.  3889.  4452.  4918.
```

With this data frame, we can make comparisons more easily: How much more expensive are "ideal" diamonds compared to "fair" diamonds? What influence does color have on the price of diamonds?

There is another way to calculate the mean values of these variables. I would like to point out that I do not recommend this approach, but the example is helpful to explain another parameter of `pivot_wider`. Suppose we do not calculate the means from the beginning and instead `select` the relevant columns and convert this data frame to a wider format:

```r
(diamonds_means_as_lists <- diamonds %>%
  select(cut, color, price) %>%
  pivot_wider(
    id_cols = cut,
    names_from = color,
    values_from = price
  ))
```

```
# A tibble: 5 x 8
  cut       E              I              J            H      F      G      D
  <ord>     <list>         <list>         <list>       <list> <list> <list> <list>
1 Ideal     <int [3,903]> <int [2,093]> <int [896]> <int>  <int>  <int>  <int>
2 Premium   <int [2,337]> <int [1,428]> <int [808]> <int>  <int>  <int>  <int>
3 Good      <int [933]>   <int [522]>   <int [307]> <int>  <int>  <int>  <int>
4 Very Good <int [2,400]> <int [1,204]> <int [678]> <int>  <int>  <int>  <int>
5 Fair      <int [224]>   <int [175]>   <int [119]> <int>  <int>  <int>  <int>
```

What we get are columns that contain lists as values. Why? Because the rows were not uniquely identifiable. A row is uniquely identifiable if for each row there is only one value per column. In our case, all value combinations of `cut` and `color` appear more than once (e.g. "Fair" + "D" appears 163 times):

```r
diamonds %>%
  select(cut, color, price) %>%
  count(cut, color)
```

```
# A tibble: 35 x 3
   cut   color     n
   <ord> <ord> <int>
 1 Fair  D       163
 2 Fair  E       224
 3 Fair  F       312
 4 Fair  G       314
 5 Fair  H       303
 6 Fair  I       175
 7 Fair  J       119
 8 Good  D       662
 9 Good  E       933
10 Good  F       909
# ... with 25 more rows
```

Enter `values_fn`. The parameter takes a function and applies this function to each list cell. For example, the first cell of column `E` contains integers.

```
diamonds_means_as_lists$E[[1]] %>% head
```

```
[1]  326  554 2757 2761 2761 2762
```

From each of these lists we can calculate its mean:

```
diamonds %>%
  select(cut, color, price) %>%
  pivot_wider(
    id_cols = cut,
    names_from = color,
    values_from = price,
    values_fn = mean
  )
```

```
# A tibble: 5 x 8
  cut           E     I     J     H     F     G     D
  <ord>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Ideal     2598. 4452. 4918. 3889. 3375. 3721. 2629.
2 Premium   3539. 5946. 6295. 5217. 4325. 4501. 3631.
3 Good      3424. 5079. 4574. 4276. 3496. 4123. 3405.
4 Very Good 3215. 5256. 5104. 4535. 3779. 3873. 3470.
5 Fair      3682. 4685. 4976. 5136. 3827. 4239. 4291.
```

These are exactly the same values we calculated with `group_by`, `summarize` and `pivot_wider`. The only difference between the two data frames is that the order of the columns and the order of the values of `cut` are different.

Again, I do not recommend this approach. With `values_fn`, `pivot_longer` does two things at once. It makes the data frame wider and calculates summary statistics for the values. Separating the two steps makes the code easier to read and more comprehensible.

17.4 How to make data frames wider for use in other software tools

Suppose you conducted an experiment to test whether caffeine has an effect on the 100-meter time of runners. Two groups ran 100 meters twice with a break of 30 minutes. On the second run, the treatment group received a caffeine boost 10 minutes before the run, while the control group didn't. The runners' cadence was also measured, i.e., the number of steps they take in one minute. Each runner is uniquely identifiable by an `id`:

```
(runners_data <- tibble(
  id   = as.numeric(gl(6, 2)),
  group = c(rep("treatment", 6), rep("control", 6)),
  measurement = c(rep(c("pre", "post"), 6)),
  speed = rnorm(12, mean = 12, sd = 0.5),
  cadence = rnorm(12, mean = 160, 3)
))
```

```
# A tibble: 12 x 5
      id group     measurement speed cadence
   <dbl> <chr>     <chr>       <dbl>   <dbl>
 1     1 treatment pre          11.8    160.
 2     1 treatment post         11.9    158.
 3     2 treatment pre          12.0    153.
 4     2 treatment post         11.9    164.
 5     3 treatment pre          11.5    158.
 6     3 treatment post         11.3    163.
 7     4 control   pre          11.8    160.
 8     4 control   post         12.2    159.
 9     5 control   pre          12.4    157.
10     5 control   post         12.2    156.
11     6 control   pre          12.1    161.
12     6 control   post         11.6    157.
```

Let's say you need to analyze the data using GUI-based software. With such software and such experimental conditions, it is not uncommon that you need to make the data frame wider to compare the pre and post values of a variable.

If you pass only the `speed` column to `values_from`, you would lose all information about the cadence of the runners:

```
runners_data %>%
  pivot_wider(
    id_cols = id,
    names_from = measurement,
    values_from = speed
  )
```

```
# A tibble: 6 x 3
```

```
     id   pre  post
  <dbl> <dbl> <dbl>
1     1  11.8  11.9
2     2  12.0  11.9
3     3  11.5  11.3
4     4  11.8  12.2
5     5  12.4  12.2
6     6  12.1  11.6
```

To keep all the information from the data frame, we need to pass the columns `speed` and `cadence` to `values_from`:

```r
runners_data %>%
  pivot_wider(
    id_cols = c(id, group),
    names_from = measurement,
    values_from = c(speed, cadence)
  )
```

```
# A tibble: 6 x 6
     id group     speed_pre speed_post cadence_pre cadence_post
  <dbl> <chr>         <dbl>      <dbl>       <dbl>        <dbl>
1     1 treatment      11.8       11.9        160.         158.
2     2 treatment      12.0       11.9        153.         164.
3     3 treatment      11.5       11.3        158.         163.
4     4 control        11.8       12.2        160.         159.
5     5 control        12.4       12.2        157.         156.
6     6 control        12.1       11.6        161.         157.
```

The function created four new columns. Why four? First you count the number of unique values in the variable specified in `names_of`. Then you multiply this number by the number of columns specified in `values_from`: $2 * 2 = 4$.

We can aesthetically improve the names of these columns with `names_sep` and `names_glue`. Let's start with `names_sep`, since we haven't seen it yet. If you pass more than one column to `values_from`, the parameter specifies how you join the values. In our case with a dot `.`:

```r
runners_data %>%
  pivot_wider(
    id_cols = c(id, group),
    names_from = measurement,
    values_from = c(speed, cadence),
```

```
    names_sep = "."
  )
```

```
# A tibble: 6 x 6
     id group     speed.pre speed.post cadence.pre cadence.post
  <dbl> <chr>         <dbl>      <dbl>       <dbl>        <dbl>
1     1 treatment      11.8       11.9        160.         158.
2     2 treatment      12.0       11.9        153.         164.
3     3 treatment      11.5       11.3        158.         163.
4     4 control        11.8       12.2        160.         159.
5     5 control        12.4       12.2        157.         156.
6     6 control        12.1       11.6        161.         157.
```

You can also change the order of the values with `names_glue`:

```
runners_data %>%
  pivot_wider(
    id_cols = c(id, group),
    names_from = measurement,
    values_from = c(speed, cadence),
    names_glue = "{measurement}.{.value}"
  )
```

```
# A tibble: 6 x 6
     id group     pre.speed post.speed pre.cadence post.cadence
  <dbl> <chr>         <dbl>      <dbl>       <dbl>        <dbl>
1     1 treatment      11.8       11.9        160.         158.
2     2 treatment      12.0       11.9        153.         164.
3     3 treatment      11.5       11.3        158.         163.
4     4 control        11.8       12.2        160.         159.
5     5 control        12.4       12.2        157.         156.
6     6 control        12.1       11.6        161.         157.
```

`.value` needs an explanation. We have seen this particular string in our `pivot_longer` tutorial. `.value` is a placeholder for the column names specified in `values_from`. Since we passed `speed` and `cadence` to the parameter, `.value` is replaced by these two values.

Not only GUI software like SPSS sometimes needs wider data to run statistical tests. Also R has some statistical functions that need wider data (see `t.test`). `pivot_wider` is therefore often needed in teams that use R in combination with GUI-based programs or for statistical analyses.

17.5 How to deal with multiple variable names stored in a column

Here is a distinctly un-tidy data frame (thanks to neregauzak for providing this data set).

```
(overnight_stays <- read_csv("data/etrm_03h_2.csv"))
```

```
# A tibble: 4 x 146
  variable       zona ~1 categ~2 da de~3 2011--~4 2011--~5 2011--~6 2011--~7 2011--~8
  <chr>          <chr>   <chr>   <chr>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 Entradas       C.A. d~ Total   Total    1.20e5   1.40e5   1.78e5   2.18e5   2.08e5
2 Pernoctaciones C.A. d~ Total   Total    2.12e5   2.47e5   3.17e5   4.03e5   3.81e5
3 Grado de ocup~ C.A. d~ Total   Total    2.72e1   3.35e1   3.76e1   4.85e1   4.46e1
4 Grado de ocup~ C.A. d~ Total   Total    3.56e1   4.36e1   4.86e1   5.64e1   5.6 e1
# ... with 137 more variables: `2011-06` <dbl>, `2011-07` <dbl>,
#   `2011-08` <dbl>, `2011-09` <dbl>, `2011-10` <dbl>, `2011-11` <dbl>,
#   `2011-12` <dbl>, `2012-01` <dbl>, `2012-02` <dbl>, `2012-03` <dbl>,
#   `2012-04` <dbl>, `2012-05` <dbl>, `2012-06` <dbl>, `2012-07` <dbl>,
#   `2012-08` <dbl>, `2012-09` <dbl>, `2012-10` <dbl>, `2012-11` <dbl>,
#   `2012-12` <dbl>, `2013-01` <dbl>, `2013-02` <dbl>, `2013-03` <dbl>,
#   `2013-04` <dbl>, `2013-05` <dbl>, `2013-06` <dbl>, `2013-07` <dbl>, ...
```

The data frame contains data on entries, overnight stays and occupancy rates in hotel establishments in the Basque Country by geographic area, category (aggregated), day of the week and month.

You may have noticed the problem with the variable `variable`. The variable contains strings with actual variable names. These four values should be columns by themselves.

Also, the values of an underlying `date` column are spread across multiple columns:

```
overnight_stays %>%
  colnames %>%
  head(n = 20)
```

```
 [1] "variable"        "zona geografica" "categoria"       "da de la semana"
 [5] "2011-01"         "2011-02"         "2011-03"         "2011-04"
 [9] "2011-05"         "2011-06"         "2011-07"         "2011-08"
[13] "2011-09"         "2011-10"         "2011-11"         "2011-12"
[17] "2012-01"         "2012-02"         "2012-03"         "2012-04"
```

To make this data frame tidy, we need to combine `pivot_longer` with `pivot_wider`. First we make the data frame longer by creating a `date` variable:

```r
(overnight_stays_longer <- overnight_stays %>%
  pivot_longer(
    cols = matches("\\d{4}-\\d{2,}"),
    names_to = "date",
    values_to = "value"
  ) )
```

```
# A tibble: 568 x 6
   variable `zona geografica` categoria `da de la semana` date     value
   <chr>    <chr>             <chr>     <chr>             <chr>    <dbl>
 1 Entradas C.A. de Euskadi   Total     Total             2011-01 120035
 2 Entradas C.A. de Euskadi   Total     Total             2011-02 140090
 3 Entradas C.A. de Euskadi   Total     Total             2011-03 177734
 4 Entradas C.A. de Euskadi   Total     Total             2011-04 218319
 5 Entradas C.A. de Euskadi   Total     Total             2011-05 207706
 6 Entradas C.A. de Euskadi   Total     Total             2011-06 225072
 7 Entradas C.A. de Euskadi   Total     Total             2011-07 273814
 8 Entradas C.A. de Euskadi   Total     Total             2011-08 277775
 9 Entradas C.A. de Euskadi   Total     Total             2011-09 239742
10 Entradas C.A. de Euskadi   Total     Total             2011-10 217931
# ... with 558 more rows
```

Still, the four variable names stored in `variable` must be columns of their own. So let's make them wider with `pivot_wider`:

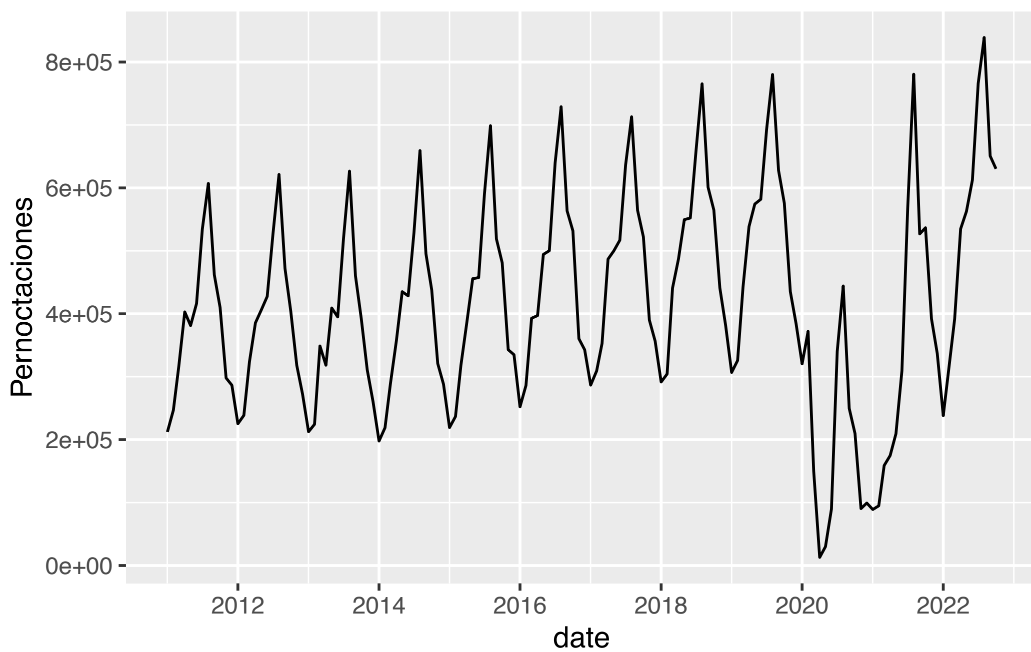
```r
(overnight_stays_tidy <- overnight_stays_longer %>%
  pivot_wider(
    names_from = variable,
    values_from = value
  ))
```

```
# A tibble: 142 x 8
   `zona geografica` categoria da de la ~1 date  Entra~2 Perno~3 Grado~4 Grado~5
   <chr>             <chr>     <chr>       <chr>   <dbl>   <dbl>   <dbl>   <dbl>
 1 C.A. de Euskadi   Total     Total       2011~  120035  212303    27.2    35.6
 2 C.A. de Euskadi   Total     Total       2011~  140090  246950    33.5    43.6
 3 C.A. de Euskadi   Total     Total       2011~  177734  316541    37.6    48.6
 4 C.A. de Euskadi   Total     Total       2011~  218319  403064    48.5    56.4
 5 C.A. de Euskadi   Total     Total       2011~  207706  381320    44.6    56
 6 C.A. de Euskadi   Total     Total       2011~  225072  416376    49.5    60.8
 7 C.A. de Euskadi   Total     Total       2011~  273814  534680    60.5    68.3
```

```
 8 C.A. de Euskadi   Total     Total        2011~  277775  607178    68.3    73.3
 9 C.A. de Euskadi   Total     Total        2011~  239742  462017    54.7    65.6
10 C.A. de Euskadi   Total     Total        2011~  217931  410032    47.8    57.7
# ... with 132 more rows, and abbreviated variable names 1: `da de la semana`,
#   2: Entradas, 3: Pernoctaciones, 4: `Grado de ocupacin por plazas`,
#   5: `Grado de ocupacin por habitaciones`
```

Now that the data frame is tidy, we can analyze the data properly. For example, we could plot the number of overnight stays over time:

```
overnight_stays_tidy %>%
  mutate(
    date = lubridate::ym(date)
  ) %>%
  ggplot(aes(x = date, y = Pernoctaciones)) +
  geom_line()
```



You can clearly see the onset of the Covid pandemic in 2020 and the seasonal trends within each year.

17.6 How to use `pivot_wider` to one-hot encode a factor

Marc-Aurèle Rivière provided me with this use case. One-hot encoding is a machine learning technique in which categorial values are converted so they are readable by machine learning algorithms. With this technique, categorical values are converted into multiple numbers such that the length of the set is equal to the number of categorical values and the numbers contain only 0s and 1s. Each set contains the number 1 only once. This technique is important for some machine learning algorithms because they can only work with numeric columns. People with a statistical background know this technique by a slightly different name: Dummy variables.

Let's say you have a data set with the sugar values of four fruits: Pineapple, watermelon, bananas and grapes. With one-hot encoding, your fruits would be represented as follows:

```
                pineapple   watermelon   bananas   grapes
pineapple               1            0         0        0
watermelon              0            1         0        0
bananas                 0            0         1        0
grapes                  0            0         0        1
```

For example, pineapples would be represented by the set $\{1, 0, 0, 0\}$. Watermelons by the set $\{0, 1, 0, 0\}$.

Let's look at the technique using a simple example data frame:

```
(sugar_in_fruits_per_100g <- tribble(
  ~id,  ~fruit,  ~sugar_level,
  1,    "pineapple", 10,
  2,    "watermelon", 6,
  3,    "banana", 12,
  4,    "grape", 16
))
```

```
# A tibble: 4 x 3
     id fruit      sugar_level
  <dbl> <chr>            <dbl>
1     1 pineapple           10
2     2 watermelon           6
3     3 banana              12
4     4 grape               16
```

The first step is to put the data into a wider format. The strange thing about this step is that we use the same column for `names_from` and `names_value`:

```r
sugar_in_fruits_per_100g %>%
  pivot_wider(
    names_from = fruit,
    values_from = fruit,
  )
```

```
# A tibble: 4 x 6
     id sugar_level pineapple watermelon banana grape
  <dbl>       <dbl> <chr>     <chr>      <chr>  <chr>
1     1          10 pineapple <NA>       <NA>   <NA>
2     2           6 <NA>      watermelon <NA>   <NA>
3     3          12 <NA>      <NA>       banana <NA>
4     4          16 <NA>      <NA>       <NA>   grape
```

Now that we have created a new column for each category or fruit, we need to convert the strings to the number 1. How can we do that? We know that the function `as.numeric` converts the value `TRUE` to the value `1`:

```r
as.numeric(TRUE)
```

```
[1] 1
```

So we have to convert the string to `TRUE`. We also know that any string is not a missing value:

```r
is.na("pineapple")
```

```
[1] FALSE
```

If we toggle this boolean value, we get `TRUE`:

```r
as.numeric(!is.na("pineapple"))
```

```
[1] 1
```

We can apply this transformation to any value of our newly created columns with `values_fn`:

```r
sugar_in_fruits_per_100g %>%
  pivot_wider(
    names_from = fruit,
    values_from = fruit,
    values_fn = \(x) as.numeric(!is.na(x))
  )
```

```
# A tibble: 4 x 6
     id sugar_level pineapple watermelon banana grape
  <dbl>       <dbl>     <dbl>      <dbl>  <dbl> <dbl>
1     1          10         1         NA     NA    NA
2     2           6        NA          1     NA    NA
3     3          12        NA         NA      1    NA
4     4          16        NA         NA     NA     1
```

Next we need to convert all `NA` to 0s. This can be done with `values_fill`. The parameter takes a value that will be used for each `NA` in the newly created columns. With `names_prefix` we can also add a prefix to the new columns:

```r
sugar_in_fruits_per_100g %>%
  pivot_wider(
    names_from = fruit,
    values_from = fruit,
    values_fn = \(x) as.numeric(!is.na(x)),
    values_fill = 0,
    names_prefix = "fruit_"
)
```

```
# A tibble: 4 x 6
     id sugar_level fruit_pineapple fruit_watermelon fruit_banana fruit_grape
  <dbl>       <dbl>           <dbl>            <dbl>        <dbl>       <dbl>
1     1          10               1                0            0           0
2     2           6               0                1            0           0
3     3          12               0                0            1           0
4     4          16               0                0            0           1
```

Voila! We have prepared the data frame for a machine learning algorithms using one-hot encoding.

17.7 How to use pivot_wider without an id column

This issue is discussed in the official pivot_wider vignette. Suppose you are faced with the challenge of cleaning up this data frame. You should make the data frame wider in that it should include a `name`, `company` and `email` column.

```
(contacts <- tribble(
  ~field, ~value,
  "name", "Jiena McLellan",
  "company", "Toyota",
  "name", "John Smith",
  "company", "google",
  "email", "john@google.com",
  "name", "Huxley Ratcliffe"
))
```

```
# A tibble: 6 x 2
  field   value
  <chr>   <chr>
1 name    Jiena McLellan
2 company Toyota
3 name    John Smith
4 company google
5 email   john@google.com
6 name    Huxley Ratcliffe
```

It seems like this data frame is a simple example of `pivot_wider`. But without an id column, the three new columns contain lists of characters:

```
contacts %>%
  pivot_wider(
    id_cols = NULL,
    names_from = field,
    values_from = value
  )
```

```
# A tibble: 1 x 3
  name        company     email
  <list>      <list>      <list>
1 <chr [3]> <chr [2]> <chr [1]>
```

Why is that? The `id` columns in `id_cols` are used to identify each observation. For each observation the function creates a new row. Since we don't have id columns, only one row is created.

The solution is simple. Create an id column. Since not every person in the data frame has a name, company, and email, we cannot iterate from 1 to 3. Doing it manually is not an option either, as the data frame can easily grow by hundreds of lines. A nice trick is to use `cumsum`.

```
(contacts_with_id <- contacts %>%
  mutate(
    id = cumsum(field == "name")
  ))
```

```
# A tibble: 6 x 3
  field   value               id
  <chr>   <chr>            <int>
1 name    Jiena McLellan       1
2 company Toyota               1
3 name    John Smith           2
4 company google               2
5 email   john@google.com      2
6 name    Huxley Ratcliffe     3
```

In the context of `mutate` `cumsum` goes through a column and increments a number by one each time a new value is reached. This technique is robust when some characteristics of a person are not present (e.g. the email).

The rest is straightforward. We convert the data into a wider format:

```
contacts_with_id %>%
  pivot_wider(
    id_cols = id,
    names_from = field,
    values_from = value
  )
```

```
# A tibble: 3 x 4
     id name             company email
  <int> <chr>            <chr>   <chr>
1     1 Jiena McLellan   Toyota  <NA>
2     2 John Smith       google  john@google.com
3     3 Huxley Ratcliffe <NA>    <NA>
```

Summary

- `pivot_wider` is often used to make data frames more readable (e.g. for posters or presentations).
- `pivot_wider` transforms data frames by reducing the number of rows and increasing the number of columns. It also reduces the total number of values within a data frame.
- `pivot_wider` can be used to implement one-hot encoding for a factor.