Data wrangling

Andrew Roth

2024-05-20

Introduction

In many cases you will need to do some manipulation to your data format before working with it. For example, generating plots in ggplot assumes your data is in tidy data format. Essentially this means each column represents one type of thing. In this tutorial we will work through some manipulations that frequently need to be done.

Tidying data

Let's start by generating some fake data to work with. Don't worry about the code for creating the data, just focus on the structure of the data for now.

```
mice_wide <- data.frame(
  id=sapply(1:50, function(x) paste("m", x, sep="")),
  pre=rnorm(10, 1, n=50),
  post=rnorm(12, 1, n=50),
  sex=c(rep("f", 25), rep("m", 25))
)

head(mice_wide)
```

```
##   id       pre     post sex
## 1 m1  9.712968 12.97262   f
## 2 m2 12.413857 11.71780   f
## 3 m3 10.876495 11.99379   f
## 4 m4 12.086012 12.58461   f
## 5 m5  9.266836 11.39080   f
## 6 m6  9.062422 13.12289   f
```

Our fake data has four columns:

- id - Unique identifier of the mouse.
- pre - Weight of the mouse before treatment.
- post - Weight of the mouse after treatment.
- sex - Sex of the mouse.

Now let's suppose we want to do some exploratory data analysis. The first thing we might want to do is visualize if there is a differnce in weight pre and post treatment. The challenge here is that our data is not in a format that ggplot can handle. Say we wanted to do a boxplot. Then we need an x column to separate the boxes and a y column for the weights. Currently our data is in a wide format because we have weights from different times represented in different columns. We are going to "tidy" our data so there is only one column for weight.

Here we will use a package called `tidyr` which is part of the tidyverse. Let's first load the package.

```
library(tidyr)
```

Now we will use a function `pivot_longer` which will help us put the data into long tidy form. The `pivot_longer` function effectively adds rows to our data frame by stacking some of the columns. It is easier to see than explain.

```
mice_long <- pivot_longer(mice_wide, c("pre", "post"))
head(mice_long)
```

```
## # A tibble: 6 x 4
##   id    sex   name  value
##   <chr> <chr> <chr> <dbl>
## 1 m1    f     pre    9.71
## 2 m1    f     post  13.0
## 3 m2    f     pre   12.4
## 4 m2    f     post  11.7
## 5 m3    f     pre   10.9
## 6 m3    f     post  12.0
```

Our new data frame now has a different set of columns: - id - Unique mouse identifier as before - sex - Mouse sex as before - name - Which has the name of the columns from the original dataset - value - Which has the corresponding value for column in with `name`

What we have done is told R to stack the pre and post columns together to "tidy" our data. We can see our new dataset has more rows.

```
nrow(mice_wide)
```

```
## [1] 50
```

```
nrow(mice_long)
```

```
## [1] 100
```

Now the default names for the new column of "name" and "value" are not very descriptive. We can fix them by passing some optional values to `pivot_long`.

```
mice_long <- pivot_longer(mice_wide, c("pre", "post"), names_to="timepoint", values_to="weight")
head(mice_long)
```

```
## # A tibble: 6 x 4
##   id    sex   timepoint weight
##   <chr> <chr> <chr>      <dbl>
## 1 m1    f     pre         9.71
## 2 m1    f     post       13.0
## 3 m2    f     pre        12.4
## 4 m2    f     post       11.7
## 5 m3    f     pre        10.9
## 6 m3    f     post       12.0
```

Now the column names are more descriptive. Before plotting let's do a bit more cleanup here and let R now we have factors.

```
mice_long$id <- factor(mice_long$id)
mice_long$sex <- factor(mice_long$sex)
mice_long$timepoint <- factor(mice_long$timepoint, levels=c("pre", "post"))
head(mice_long)
```

```
## # A tibble: 6 x 4
```

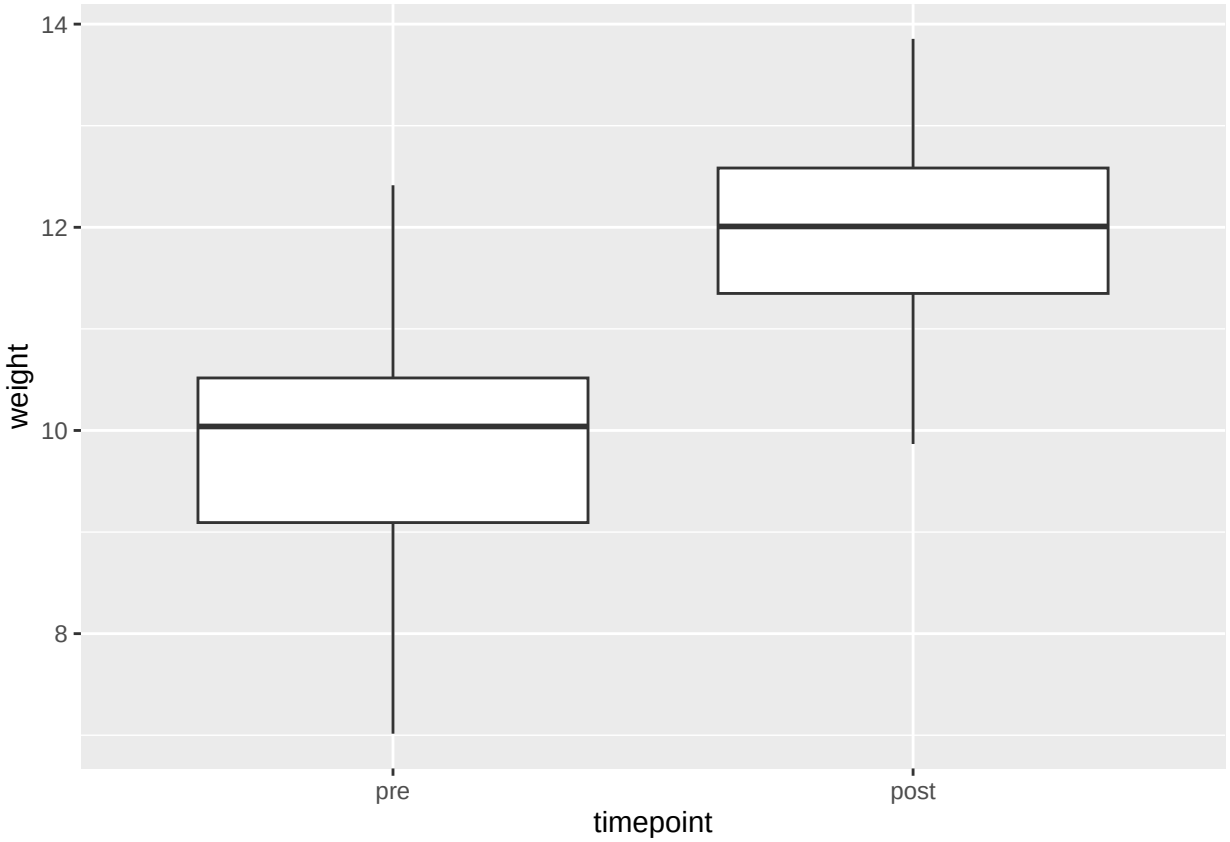
```
##   id    sex   timepoint weight
##   <fct> <fct> <fct>      <dbl>
## 1 m1    f     pre         9.71
## 2 m1    f     post       13.0
## 3 m2    f     pre        12.4
## 4 m2    f     post       11.7
## 5 m3    f     pre        10.9
## 6 m3    f     post       12.0
```

I have made the id, sex and timepoint columns. I also specified the order of the timepoints since pre can be thought of as before post.

Let's try an exploratory boxplot to see if treatment has and effect.

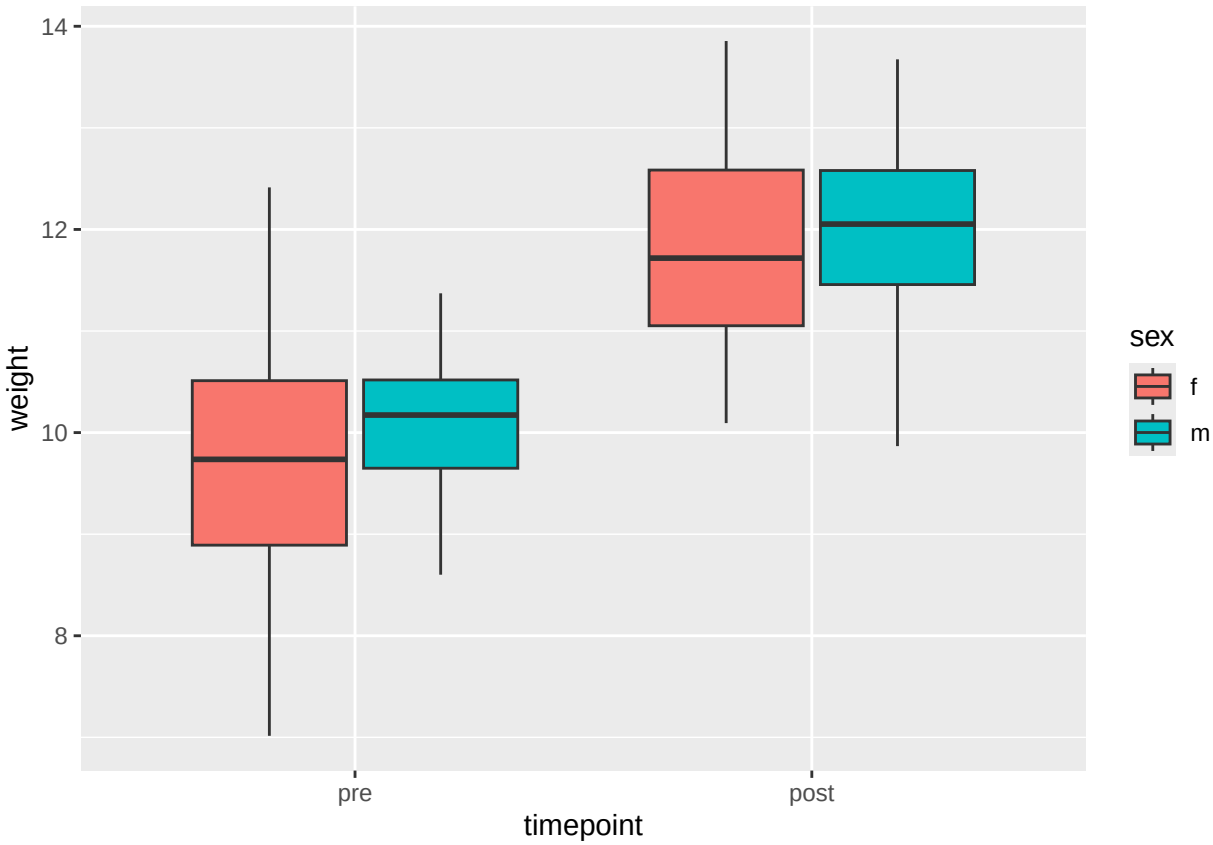
```
library(ggplot2)

ggplot(mice_long, aes(x=timepoint, y=weight)) + geom_boxplot()
```



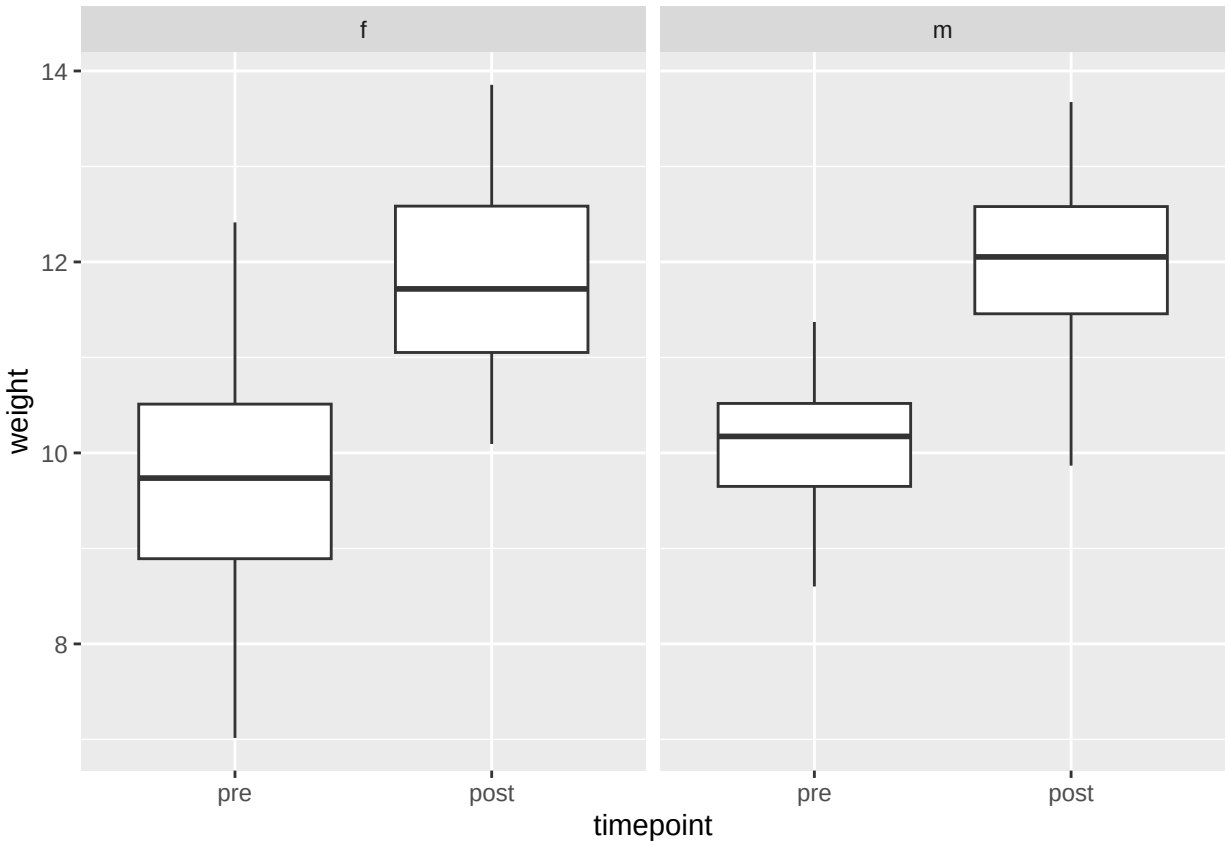
Looks like there is an effect! We can also consider the sex as part of this.

```
ggplot(mice_long, aes(x=timepoint, y=weight, fill=sex)) + geom_boxplot()
```



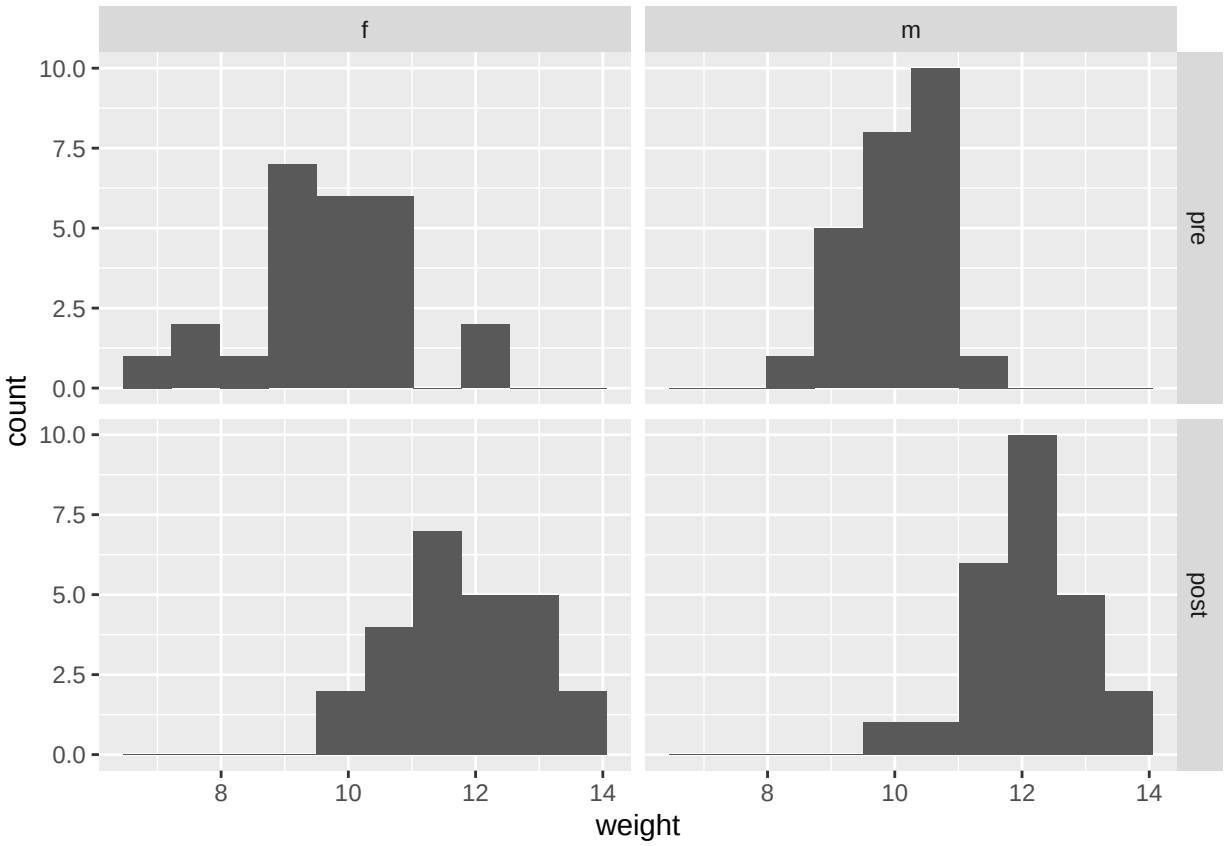
We can also use the `facet_grid` feature from ggplot to do this sort of visualization.

```
ggplot(mice_long, aes(x=timepoint, y=weight)) +
  geom_boxplot() +
  facet_grid(~sex)
```



Here we add the `facet_grid` onto our plot. The argument to `facet_grid` specifies how things should be laid out. It use the R syntax for formulas i.e. `y ~ x`. For facet grids the the variable for the rows comes before the `~` and the variable for column after the `~`. Let's see another example where we do row and column.

```
ggplot(mice_long, aes(x=weight)) +
  geom_histogram(bins=10) +
  facet_grid(timepoint~sex)
```



Going from long to wide

Tidy data is the format you will use when working with ggplot and other parts of the tidyverse. However, it is sometimes useful to have your data in wide format. You can move from tidy/long format to wide format using `pivot_wider` function.

```
mice_wide_again <- pivot_wider(mice_long, names_from=timepoint, values_from=weight)
head(mice_wide_again)
```

```
## # A tibble: 6 x 4
##   id    sex     pre  post
##   <fct> <fct> <dbl> <dbl>
## 1 m1    f      9.71  13.0
## 2 m2    f     12.4   11.7
## 3 m3    f     10.9   12.0
## 4 m4    f     12.1   12.6
## 5 m5    f      9.27  11.4
## 6 m6    f      9.06  13.1
```

For example we might want to do a t-test comparing pre/post treatment. We can use the builtin R function `t.test` to do this. The `t.test` functions takes two arguments, each vectors corresponding to the two sets of observations we want to compare. Let's try:

```
t.test(mice_wide_again$pre, mice_wide_again$post)
```

```
##
##  Welch Two Sample t-test
##
## data:  mice_wide_again$pre and mice_wide_again$post
```

```
## t = -10.282, df = 96.559, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.497245 -1.689097
## sample estimates:
## mean of x mean of y
##  9.861743 11.954914
```

We could have accomplished the same thing if we kept the data in long/tidy format. It would require we use the R indexing though. Below we select all weights from the pre-treatment timepoint.

```
mice_long[mice_long$timepoint == "pre", "weight"]
```

```
## # A tibble: 50 x 1
##    weight
##     <dbl>
##  1   9.71
##  2  12.4
##  3  10.9
##  4  12.1
##  5   9.27
##  6   9.06
##  7  10.9
##  8   8.56
##  9  11.0
## 10   8.89
## # i 40 more rows
```

We can use this idea to do our t-test.

```
t.test(
  mice_long[mice_long$timepoint == "pre", "weight"],
  mice_long[mice_long$timepoint == "post", "weight"]
  )
```

```
##
##  Welch Two Sample t-test
##
## data:  mice_long[mice_long$timepoint == "pre", "weight"] and mice_long[mice_long$timepoint == "post"
## t = -10.282, df = 96.559, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.497245 -1.689097
## sample estimates:
## mean of x mean of y
##  9.861743 11.954914
```

> Either way of doing the compuation works. Do what ever you feel most comfortable. One tip is to use a consistent style when you write code. So if you decide to go the pivot_wider route, do that consistently in your code.

Manipulating values

In the plotting tutorial we did some work to replace the sex variable which started as integer with Male/Female labels. The solution I gave there used builtin R functionality. Here we will explore a different approach to replacing values that is a bit more elegant. We will use the `dplyr` library which is another piece of the tidyverse. In general you can think of `dplyr` as an enhanced way to to manipulate your data in place of

things you could with R indexing.

Let's try an example where we replace pre/post with before/after in our timepoint column.

First we load `dplyr`.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Now we can do the replacement using the `mutate` and `recode` functions from `dplyr`.

```
mutate(
  mice_long,
  timepoint=recode(timepoint, "pre"="before", "post"="after")
  )
```

```
## # A tibble: 100 x 4
##    id    sex   timepoint weight
##    <fct> <fct> <fct>      <dbl>
##  1 m1    f     before      9.71
##  2 m1    f     after      13.0
##  3 m2    f     before     12.4
##  4 m2    f     after      11.7
##  5 m3    f     before     10.9
##  6 m3    f     after      12.0
##  7 m4    f     before     12.1
##  8 m4    f     after      12.6
##  9 m5    f     before      9.27
## 10 m5    f     after      11.4
## # i 90 more rows
```

The `mutate` function serves to alter the values in a dataset. Here we specify the column, timepoint, we want to alter and tell `mutate` that we want to alter it with `recode`.

Often when you look up examples using the tidyverse and `dplyr` in particular you will see them using the pipe operator `%>%`. The pipe operator is useful when stringing multiple transformations of the data together. To understand imagine your code looks like `a %>% some_func(arg_1=TRUE)`. What this says is pass the variable `a` into the function `some_func` as the first argument. I am also passing the additional argument `arg_1` to `some_func`. Let's take a look at a concrete example where we use `mutate` like before.

```
mice_long %>%
  mutate(
    timepoint=recode(timepoint, "pre"="before", "post"="after")
    )
```

```
## # A tibble: 100 x 4
##    id    sex   timepoint weight
##    <fct> <fct> <fct>      <dbl>
##  1 m1    f     before      9.71
##  2 m1    f     after      13.0
```

```
##  3 m2    f     before     12.4
##  4 m2    f     after      11.7
##  5 m3    f     before     10.9
##  6 m3    f     after      12.0
##  7 m4    f     before     12.1
##  8 m4    f     after      12.6
##  9 m5    f     before      9.27
## 10 m5    f     after      11.4
## # i 90 more rows
```

We get the same result as when we explicitly pass in the `mice_long` data frame to `mutate`. The utility of pipes in propobably not obvious from this simple example, but when you do multiple transformations it can be helpful. For more information checkout the R for Data Science manual.

Let's try a more complicated example. In the next block of code I will do the following: - Rename the values of the timepoint column using `mutate` - Group the rows based on the value of timepoint - Compute the mean weight of the groups This will return a new data frame with two columns timepoint and mean_weight. The name mean_weight is specified by us.

```
mice_long %>%
  mutate(timepoint=recode(timepoint, "pre"="before", "post"="after")) %>%
  group_by(timepoint) %>%
  summarise(mean_weight=mean(weight))
```

```
## # A tibble: 2 x 2
##   timepoint mean_weight
##   <fct>           <dbl>
## 1 before           9.86
## 2 after           12.0
```

Pipes and `dplyr` are a bit complex. If you do not completely understand that is fine. You do everything you need to in this course without them. I am mainly explaining them so you can understand some of the solutions you will find online.