A ggplot2 Tutorial

Maximilian Klemp

2023-11-26

A tutorial on the use of ggplot2 in R.

Data

Before we can start into building plots with ggplot2, we have to load data to do so. To this end, we utilize the `Rdshs` package that was created specifically to provide data for this course. In case you haven't used the package yet or you have to update it, uncomment and run the `devtools::install_github()` function in line 20 first. Once installed, you can omit this function from your notebook. Afterwards, load the package via the `library()` function at load the datasets you need into the namespace using `data()`. For the rest of this tutorial, we will need the `fb.europe.xg1622`, the `fb.leicester.xg1516` and the `fb.europe.players1718` dataframes.

```r
#devtools::install_github("spoho-datascience/Rdshs")
library(tidyverse)
library(ggsoccer)
library(knitr)
library(Rdshs)
data("fb.europe.xg1622")
data("fb.leicester.xg1516")
data("fb.europe.players1718")

# make the position an ordered factor
fb.europe.players1718 <- fb.europe.players1718 %>%
  mutate(
    foot = ifelse(foot == "", "both", foot),
    position_wy = factor(position_wy, levels = c("Goalkeeper", "Defender", "Midfielder", "Fo
  )

my_pitch_dimensions <- list(
  length = 105,
  width = 68,
  penalty_box_length = 16,
  penalty_box_width = 39.32,
  six_yard_box_length = 5,
  six_yard_box_width = 17.32,
  penalty_spot_distance = 11,
  goal_width = 7.32,
  origin_x = 0,
  origin_y = 0
)
```

Introduction

We wuickly review the Grammar of Graphics as it is at the heart of `ggplot2`. Afterwards, this notebook will contain mostly code, with only short explanatory texts. If you need a deeper understanding of the contents covered, please revisit the slides or the video file of this session. In order to make this tutorial accessible, even if rendered to PDF or HTML, the `echo` parameter in the YAML header is set to `true` so that the code chunks are printed to the rendered document. If you want to avoid this behavior, change the parameter to `false`.

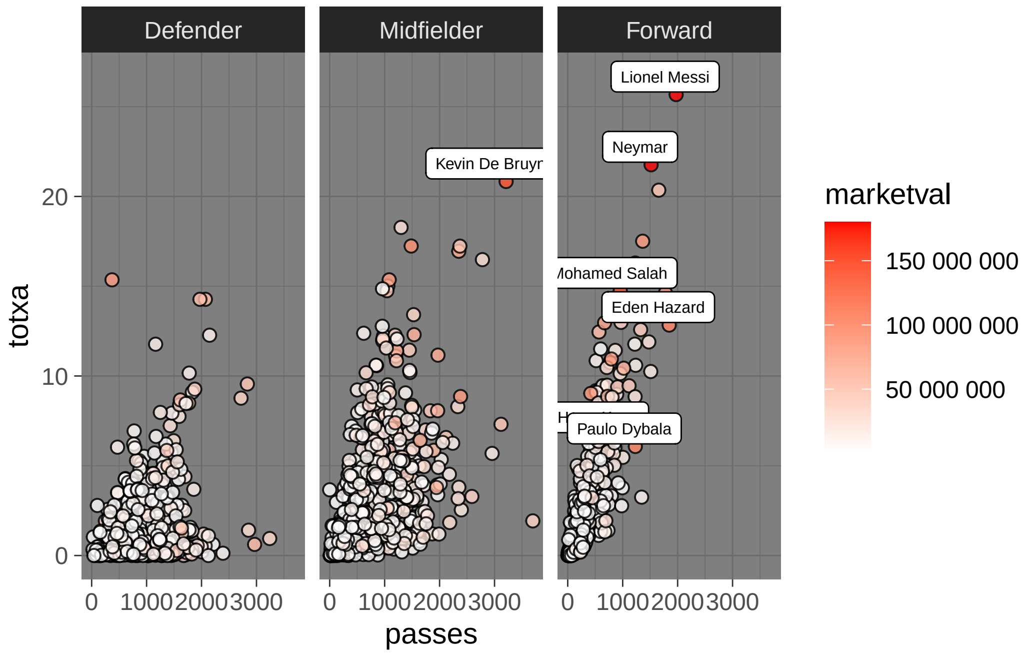
Grammar of Graphics

We quickly review the Grammar of

Grammar of Graphics assumes that all plots are composed of:

- **Data** you want to visualize and a set of aesthetic **mappings** describing how variables are mapped to aesthetic attributes (e.g. the value of a variable might be mapped to the position of a point on the coordinate system or to the color of the point)
- **Layers** made up of geometric elements and statistical transformation (a layer can be a set of points showing the raw data, another layer can be a regression line on the data, which refers to a statistical transformation)
- **Scales** map values in data space to values in aesthetics space, such as color, size, shape (e.g. a categorical variable is mapped to a set of different points shapes or a continuous variable is mapped to a color scale)
- **Coordinate System** describes how data coordinates are mapped to plane of graphic; normally Cartesian Coordinate System is used, but others are available
- **Faceting** specification describes breaking up data into subsets and displaying those subsets
- **Theme** controls appearance of the plot, like font size, background color etc.

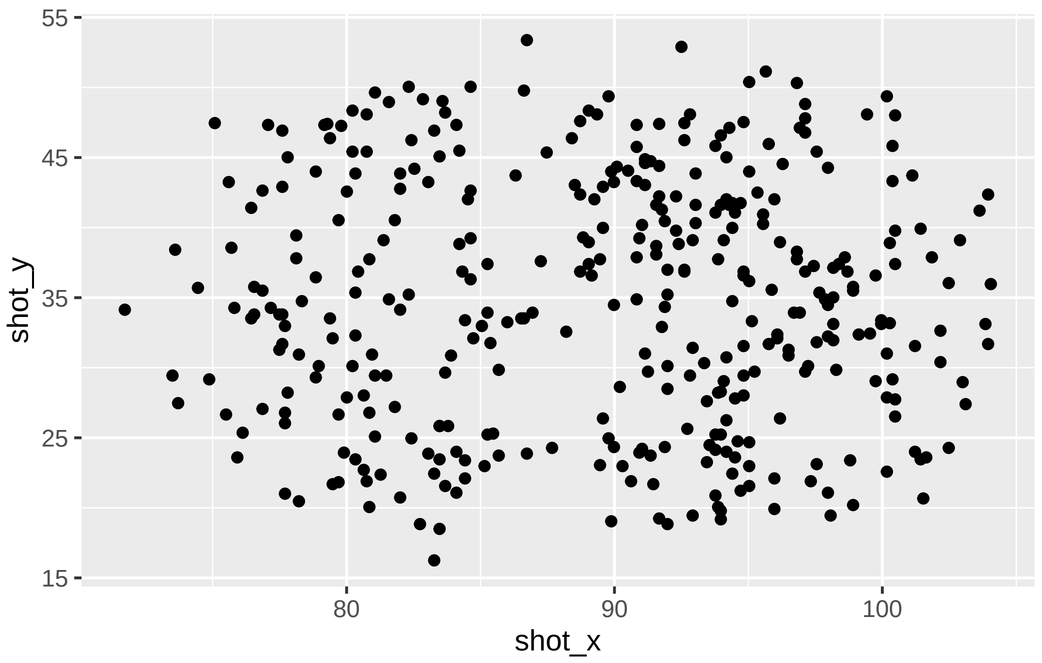
```
fb.europe.players1718 %>%
  filter(position_wy != "Goalkeeper",
         !is.na(marketval)) %>%
  ggplot(aes(x=passes, y=totxa)) +
  geom_point(aes(fill=marketval), shape = 21, size = 2, alpha = .8) +
  facet_wrap(~position_wy) +
  geom_label(data = filter(fb.europe.players1718, marketval > 100000000), aes(label = player
  scale_fill_gradient(low = "white", high = "red", labels = scales::label_number()) +
  theme_dark()
```



Basic ggplot2 structure

Example plot using `fb.leicester.xg1516`:

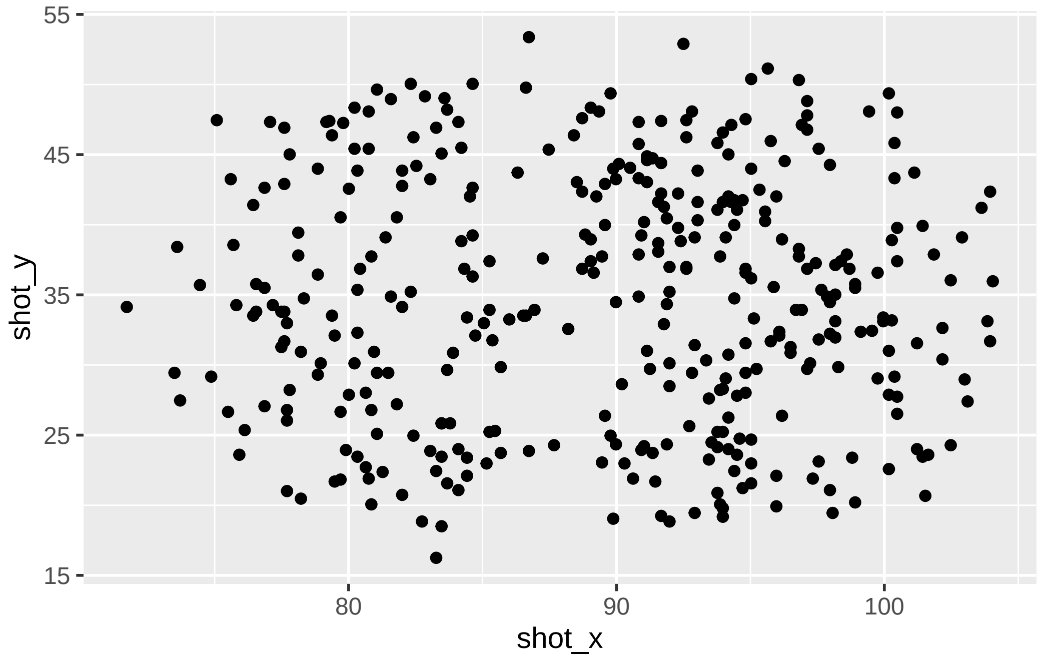
```
ggplot(fb.leicester.xg1516, aes(x=shot_x, y=shot_y)) +
geom_point()
```



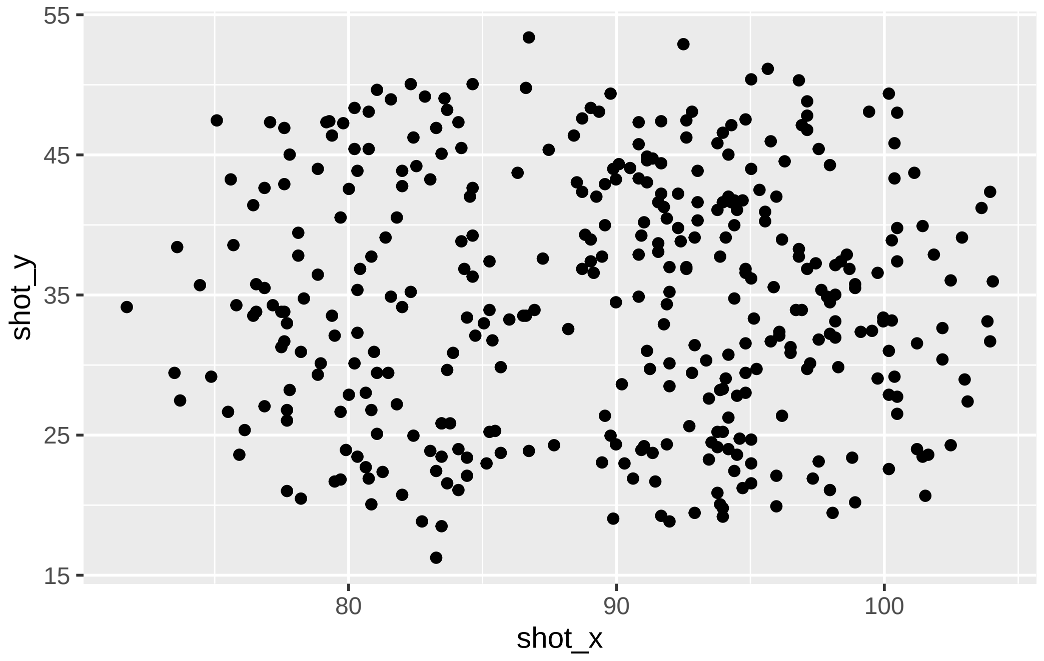
Using pipes instead of the data argument

- the `data` argument to the `ggplot` function indicates the dataset to be used
- instead of calling `ggplot` and providing the dataset as the first argument, you can also call the dataset first and then use the pipe operator `%>%` to pass the data to a `ggplot` function
- this is especially useful, if transformations shall be applied to data before creating the plot, such as `mutate` and `filter`
- both code snippets render the same plot:

```
ggplot(fb.leicester.xg1516, aes(x=shot_x, y=shot_y)) +
geom_point()
```



```
fb.leicester.xg1516 %>% ggplot(aes(x=shot_x, y=shot_y)) +
geom_point()
```



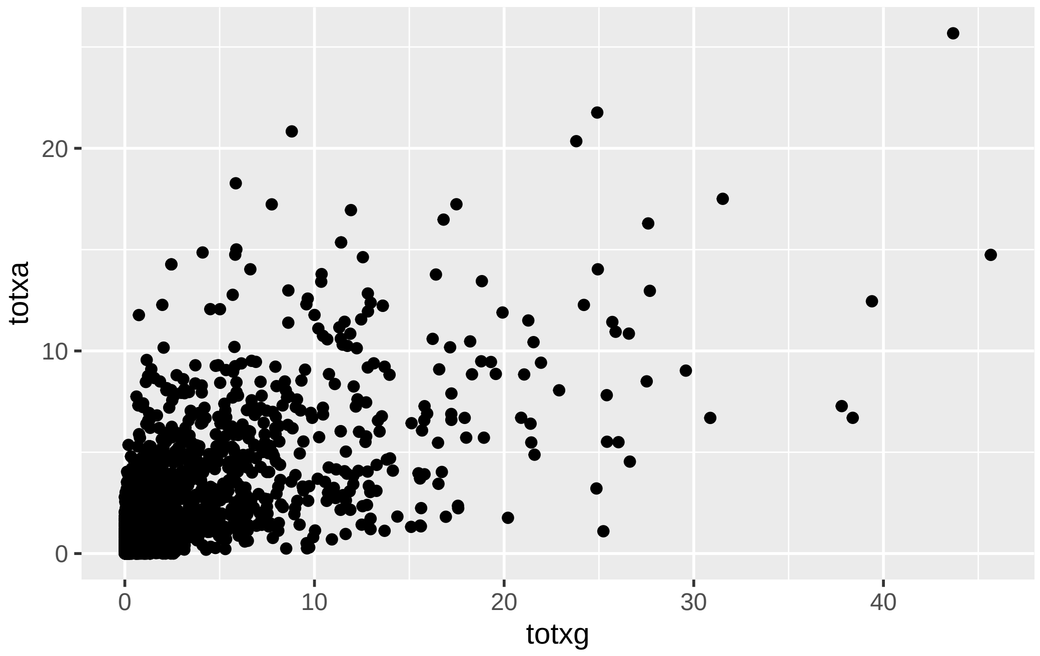
:::

ggplot2 Tutorial

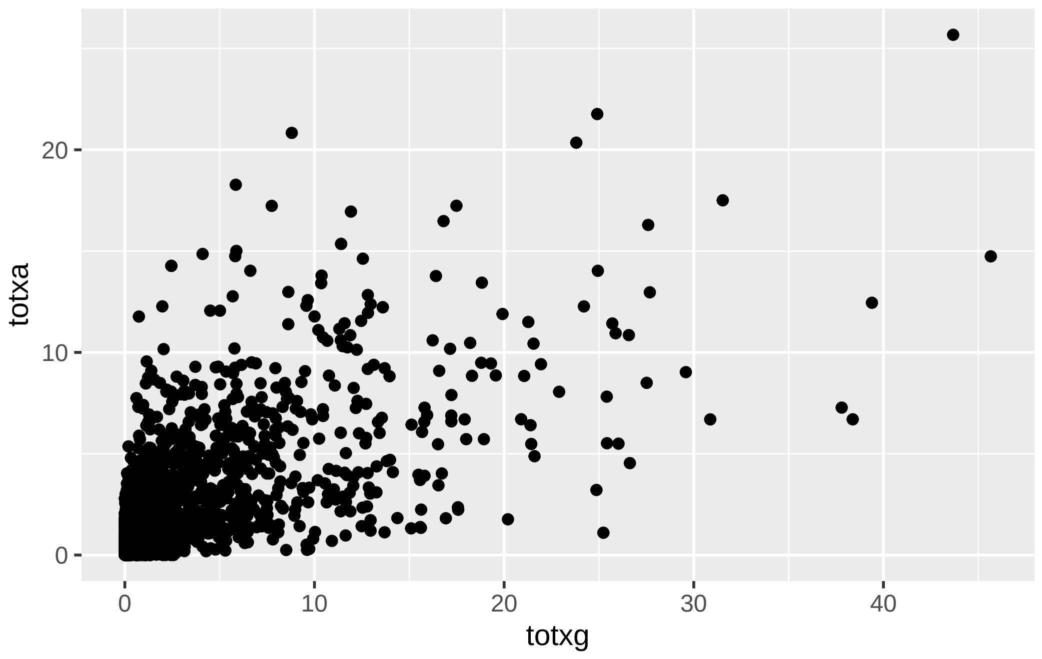
Scatterplot

- aesthetics defined within `ggplot` object will be used in every subsequent geom if not overridden
- aesthetics defined within a **geom** object will be used only for that geom
- both code snippets render the same plot

```
fb.europe.players1718 %>%
  ggplot(aes(x=totxg, y = totxa)) +
  geom_point()
```



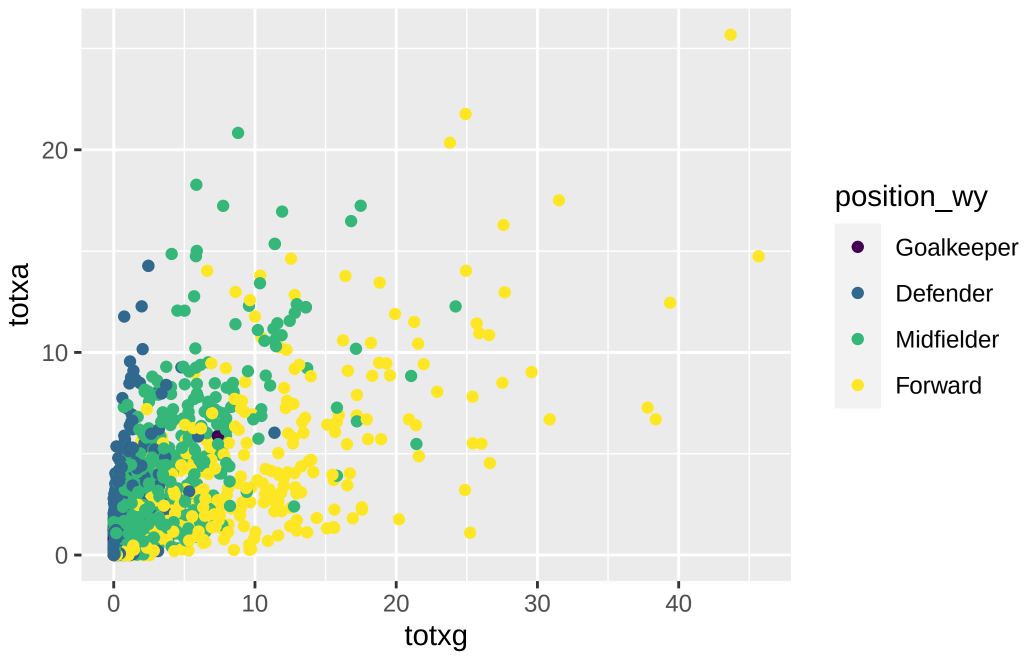
```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x=totxg, y = totxa))
```



Scatterplot - color mapping

- the `col` aesthetic maps a variable to the color of the points
- in this example, point color indicates player positions
- also possible to map a continuous variable

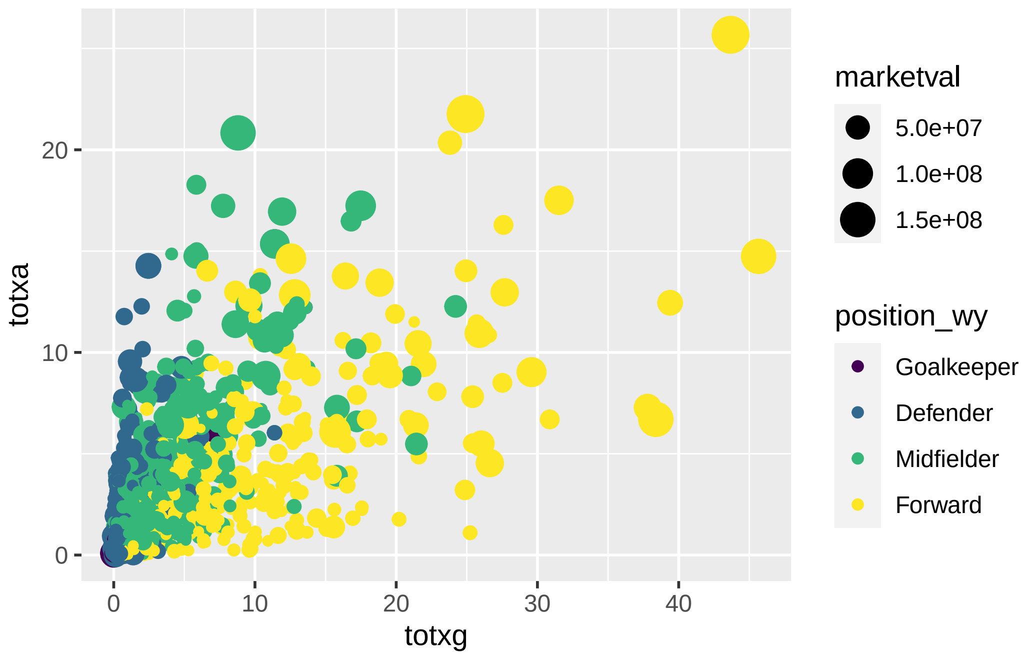
```
fb.europe.players1718 %>%
  ggplot(aes(x=totxg, y = totxa)) +
  geom_point(aes(col = position_wy))
```



Scatterplot - size mapping

- the `size` aesthetic maps a variable to the size of the points
- in this example, point size indicates player market value
- also possible to map a discrete variable

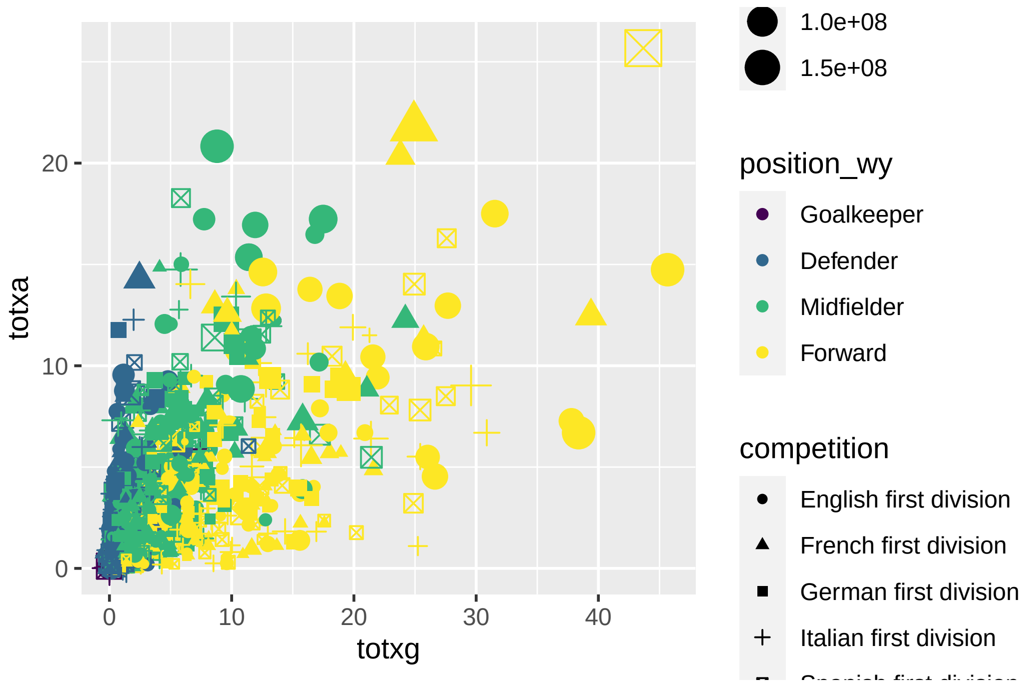
```
fb.europe.players1718 %>%
  ggplot(aes(x=totxg, y = totxa)) +
  geom_point(aes(col = position_wy, size = marketval))
```



Scatterplot - shape mapping

- the `shape` aesthetic maps a variable to the shape of the points
- in this example, point shape indicates the league the players are competing in
- mapping a continuous variable to `shape` aesthetic is not possible

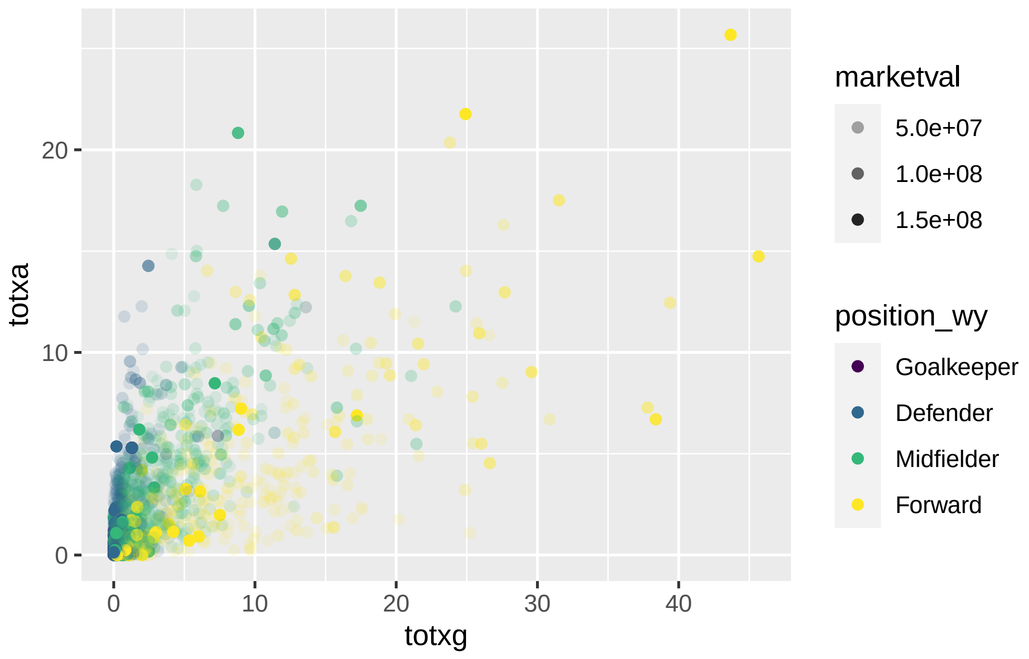
```
fb.europe.players1718 %>%
  ggplot(aes(x=totxg, y = totxa)) +
  geom_point(aes(col = position_wy, size = marketval, shape = competition))
```



Scatterplot - alpha mapping

- the `alpha` aesthetic maps a variable to the transparency of the points
- now, the market value is mapped to point transparency instead of point size
- mapping a continuous variable to `alpha` aesthetic is technically possible, but not useful

```
fb.europe.players1718 %>%
  ggplot(aes(x=totxg, y = totxa)) +
  geom_point(aes(col = position_wy, alpha = marketval))
```



Mapping vs. setting

- so far, we performed **mapping** of different aesthetics to variables
- e.g. we mapped the position of the players to the `color` aesthetic of the points or the market value to the `size` aesthetic
- also, xG and xA values were mapped to`x` and `y` aesthetics

Instead of **mapping** aesthetics to *variables*, it is also possible to **set** them to *constant* values

Aesthetics that are commonly used in ggplot are

- color
- size
- shape
- fill
- alpha
- …

Different **geoms** accept different **aesthetics**

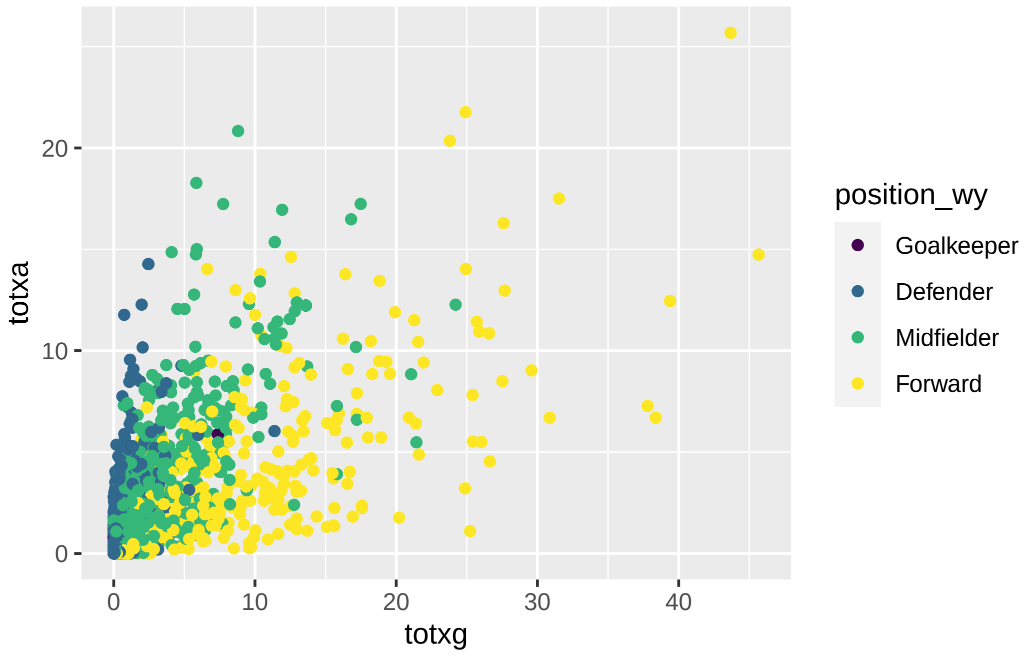
- if we just want to modify the appearance of a geom (color, size, shape) without including information in it, we can just **set** the respective aesthetic refer back to grammar, aesthetic mappings
- in ggplot2, the difference between **mapping** and **setting** is made by either defining the respective aesthetic within the `aes()` function or outside of it

- if an aesthetic is defined within `aes()`, it is mapped to a variable, if it is defined outside of `aes()`, it is set to a value

Mapping vs. setting

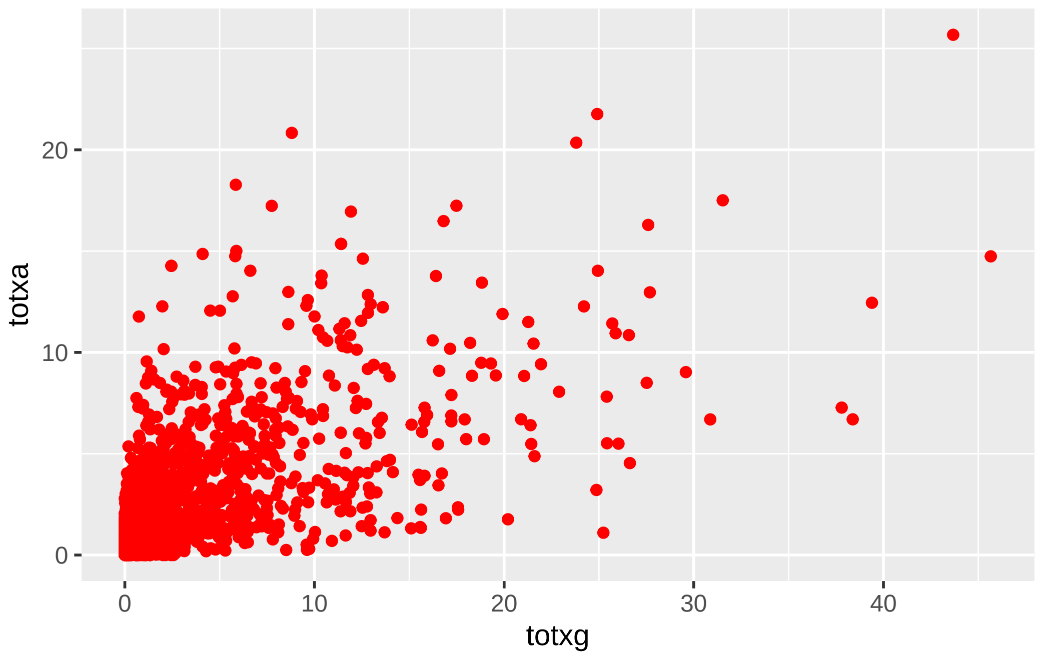
Mapping

```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x = totxg, y = totxa, col = position_wy))
```



Setting

```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x = totxg, y = totxa), col = "red")
```

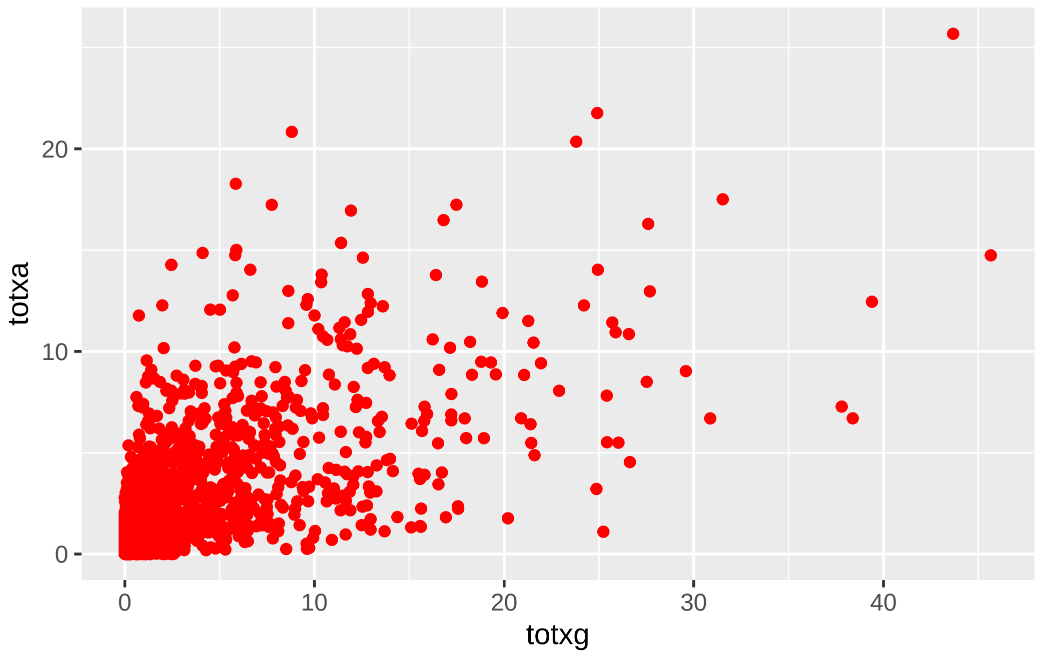


Mapping vs. setting with geom_point

Color

- if `col` is specified outside `aes()`, a fixed color can be defined for the respective geom
- to call a color, different methods are available:
  - the **name**, e.g. `col = "red"`; all 657 color names can be accessed by `colors()`
  - using `rgb()`, manually specifying intensites of *red*, *green*, and *blue*, e.g. `col = rgb(1, 0, 0)`
  - directly specifying the hexadecimal code, e.g. `col = #FF0000`

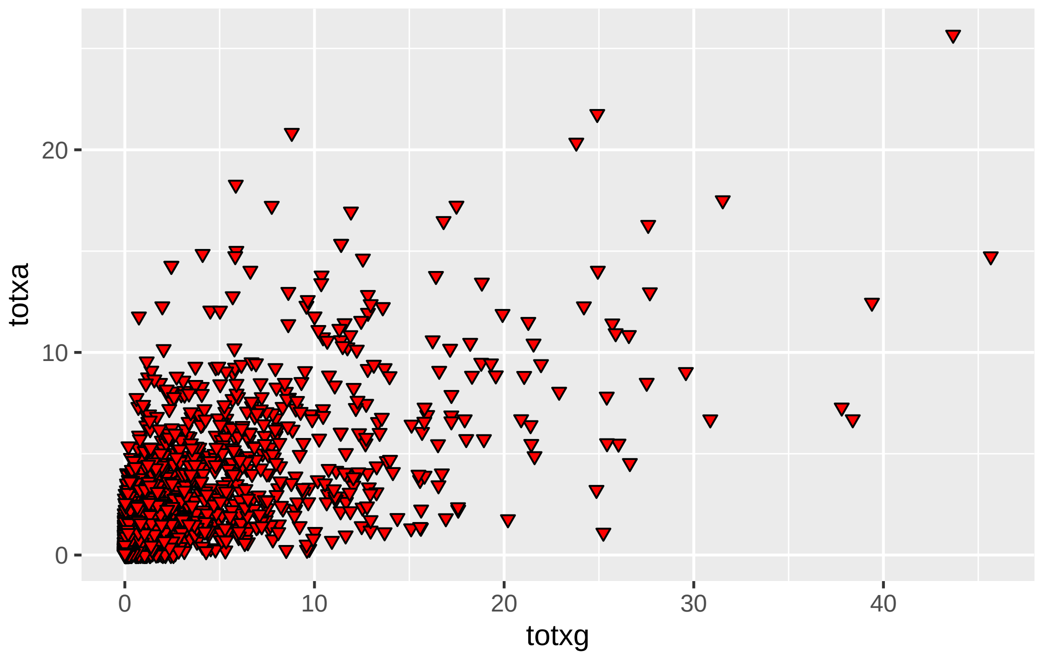
```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x=totxg, y = totxa), col = "#FF0000")
```



Shape and Fill

- by specifying `shape` outside of `aes()`, a desired shape can be chosen for points
- shapes are specified using their respective number or name:
- depending on the chosen shape, `col` either defines the color of the whole point or only the border color
- e.g. shape 21 allows to define different colors for border and area of the geom - in this case, `fill` aesthetic is used for the main area

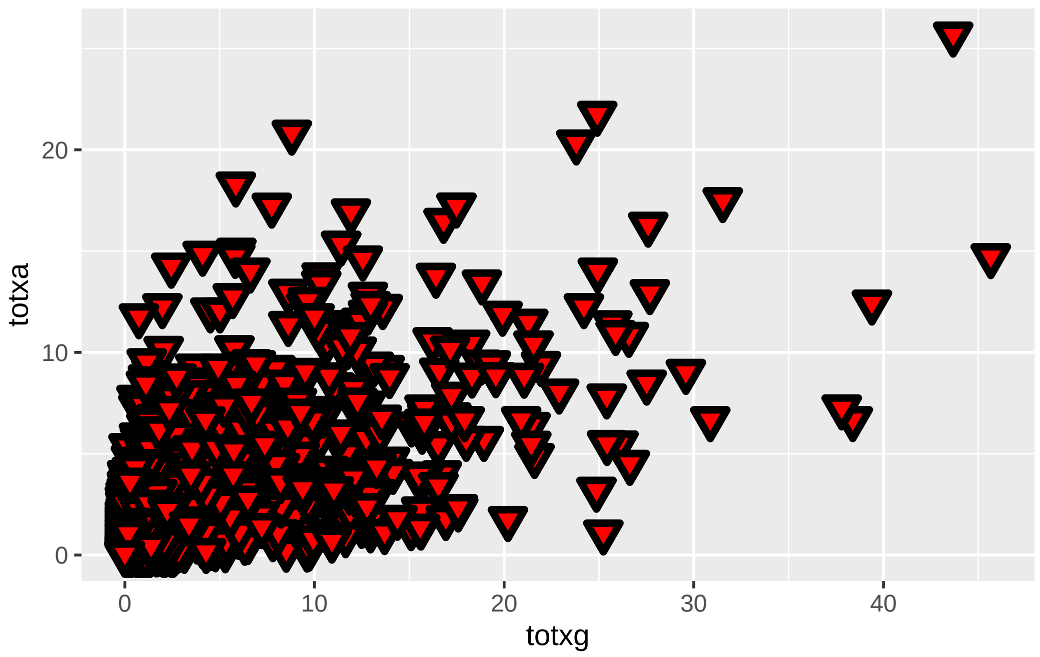
```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x=totxg, y = totxa), shape = 25, col = "black", fill = "red")
```



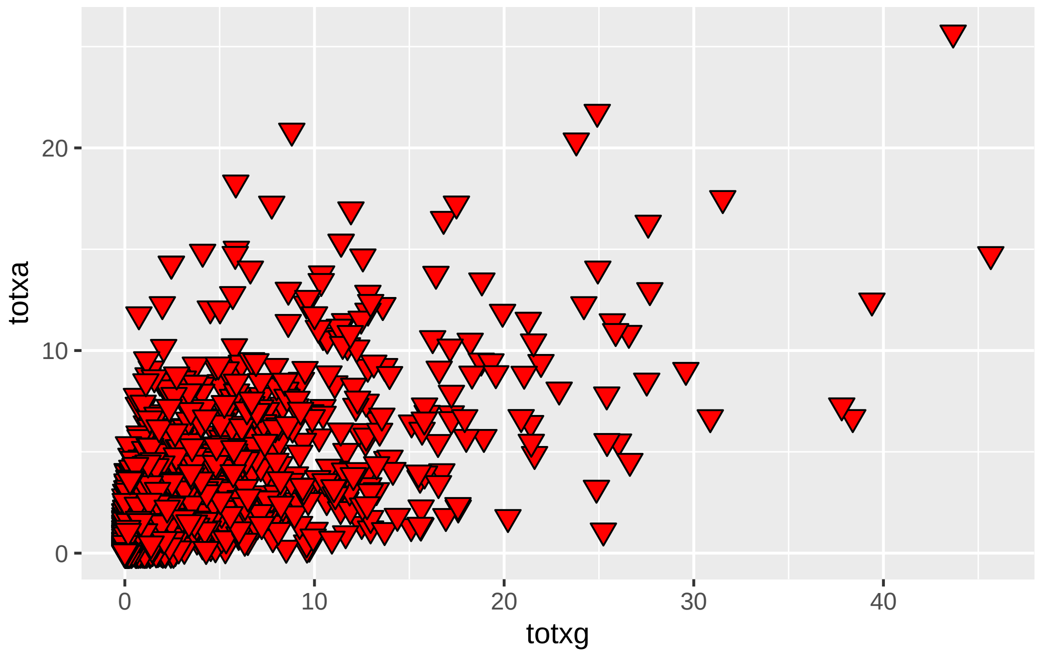
Size and stroke

- `size` accepts a numeric value, specifying the size of a point in mm
- for shapes that allow separate specification of `col` and `fill`, the size of the filled part is controlled by `size` and the border linewidth is controlled by `stroke`

```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x=totxg, y = totxa), shape = 25, col = "black", fill = "red", size = 3, str
```



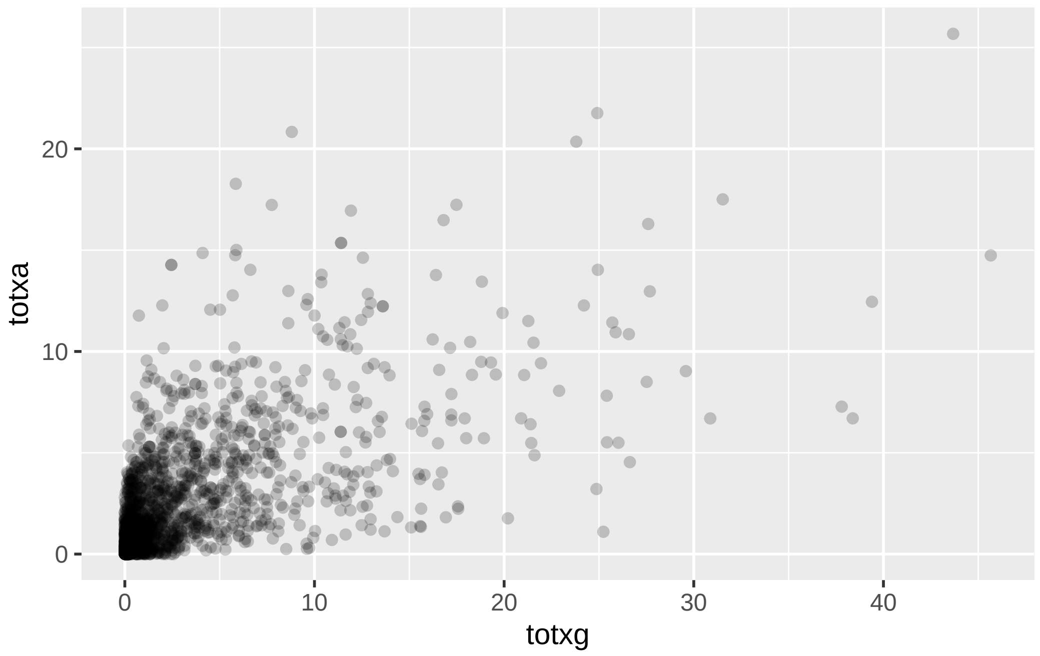
```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x=totxg, y = totxa), shape = 25, col = "black", fill = "red", size = 3, str
```



Transparency

- `alpha` accepts a numeric value, but only values in $[0, 1]$ affect the appearance
- if alpha is specified as a fraction with numerator 1, the denominator indicates how many points have to be plotted on top of each other for solid shape
- i.e., if `alpha` is set to $\frac{1}{5}$ or `alpha = 0.2`, five points overlapping result in a solid appearance

```
fb.europe.players1718 %>%
  ggplot() +
  geom_point(aes(x=totxg, y = totxa), alpha = .2)
```



Other geoms

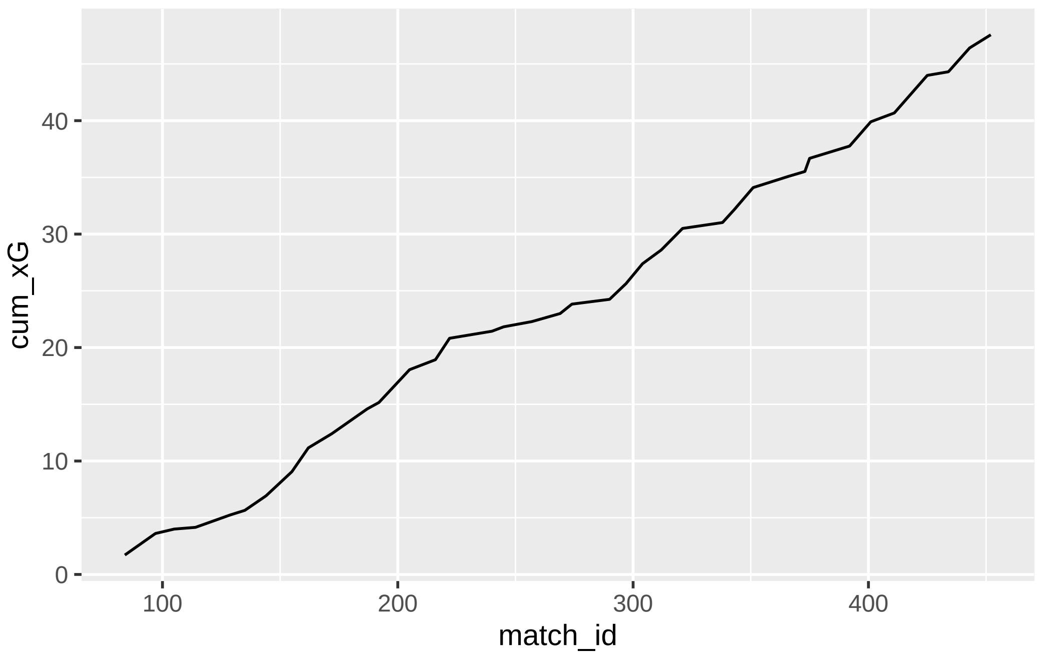
- in the previous code snippets, `geom_point()` could be replaced by any other geometric object
- however, different **geoms** accept different aesthetics (while a lot of them are common to most geoms)
- common geoms and the respective plots they are used for are:

geom	Plot
`geom_line()`	Line plot
`geom_boxplot()`	Box plot
`geom_bar()`	Bar plot
`geom_col()`	Bar plot
`geom_histogram()`	Histogram
`geom_freqpoly()`	Density plot
`geom_smooth()`	(Nonlinear) regression line
…	…

geom_line

- `geom_line()` mostly used for time series data
- e.g. evolution of cumulated xG values for one team during a season
- `geom_path()` is an alternative where lines are not restrained to go left to right

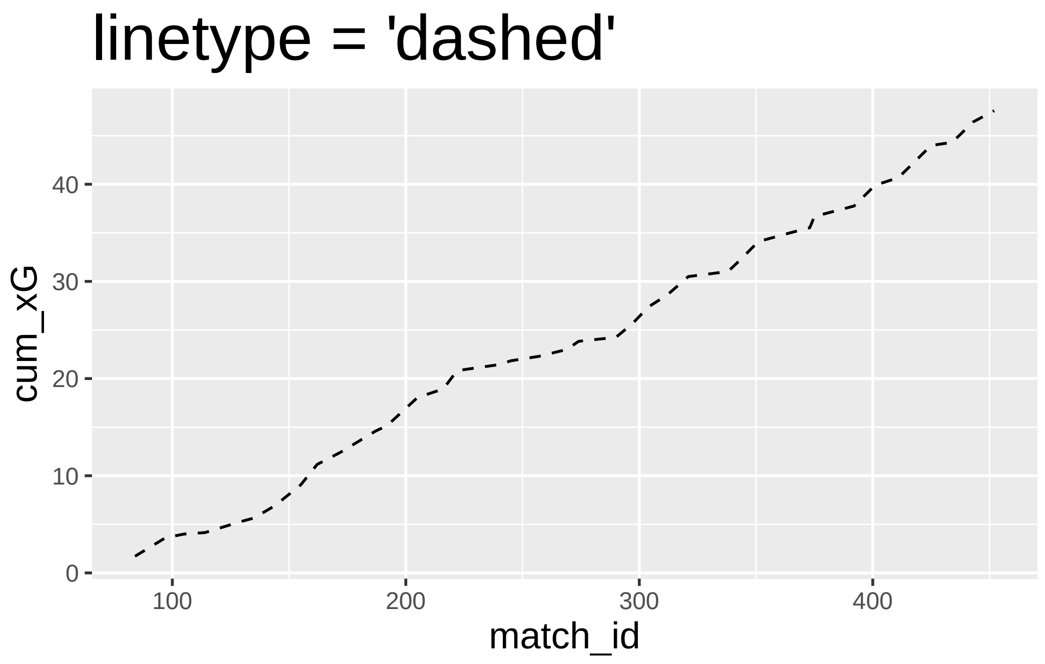
```r
fb.leicester.xg1516 %>%
  group_by(match_id) %>%
  summarise(date = first(date),
            xG = sum(xG)) %>%
  ungroup() %>%
  mutate(cum_xG = cumsum(xG),
         date = as.Date(date)) %>%
  ggplot() +
  geom_line(aes(x=match_id, y=cum_xG))
```



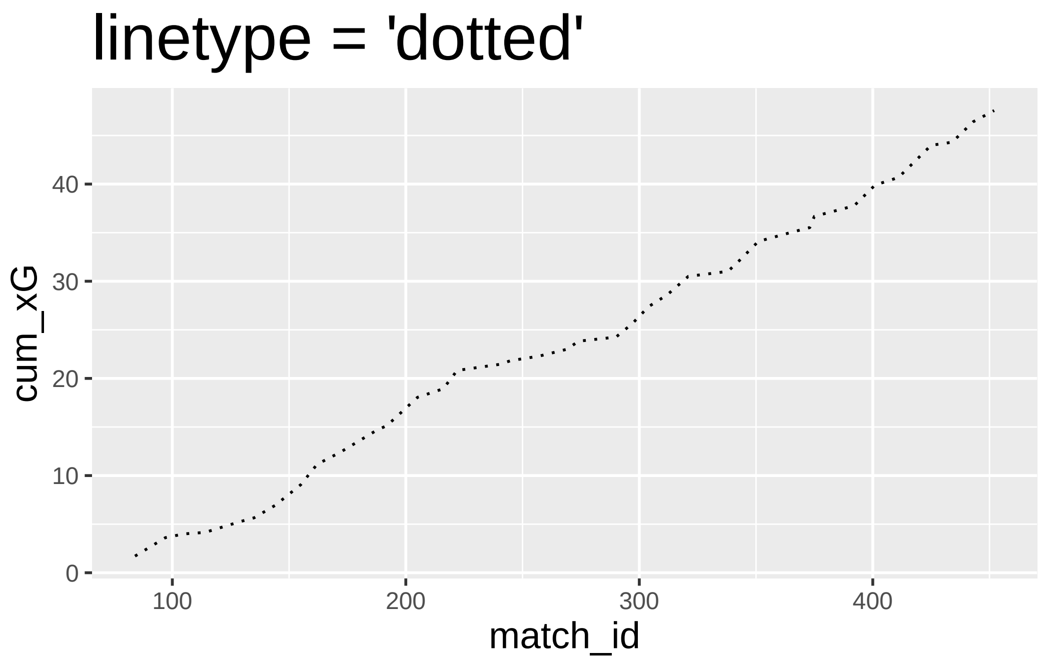
Linetype

- `linetype` can be specified with **name** or **number**: 0 = blank, 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash:
- you can also customize the length of on/off stretches of a line using 2, 4, 6 or 8 hexadecimal digits

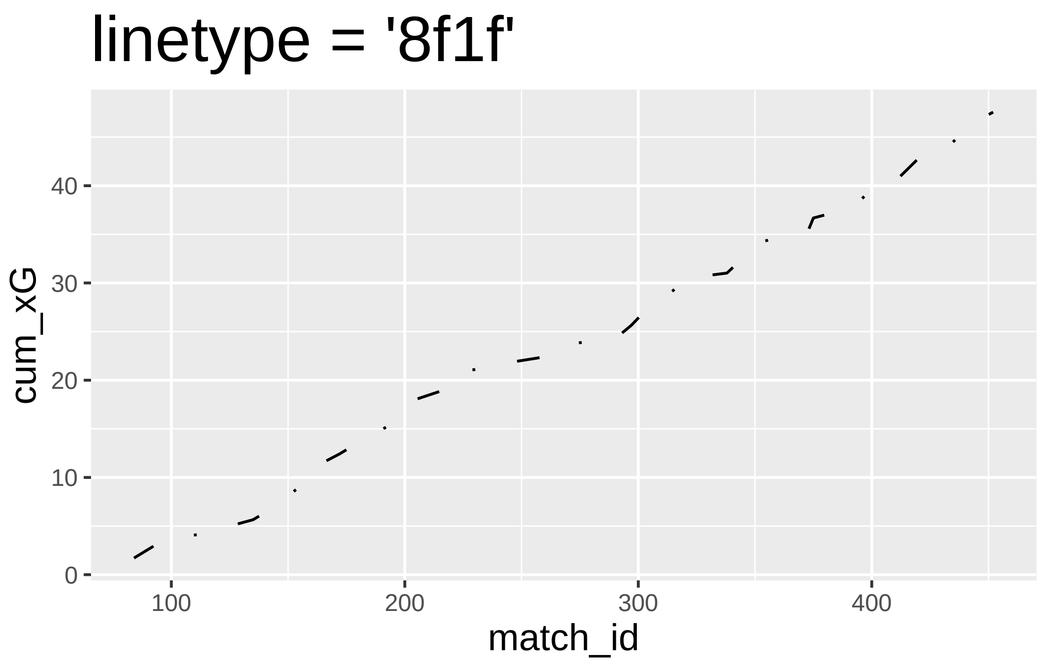
```r
p_line <- fb.leicester.xg1516 %>%
  group_by(match_id) %>%
  summarise(date = first(date),
            xG = sum(xG)) %>%
  ungroup() %>%
  mutate(cum_xG = cumsum(xG),
         date = as.Date(date)) %>%
  ggplot() +
  theme(title = element_text(size = 20), axis.title.x = element_text(size = 14), axis.title.

p_line + geom_line(aes(x=match_id, y=cum_xG), linetype = 'dashed') +
  ggtitle("linetype = 'dashed'")
```



```
p_line + geom_line(aes(x=match_id, y=cum_xG), linetype = 'dotted')+
  ggtitle("linetype = 'dotted'")
```



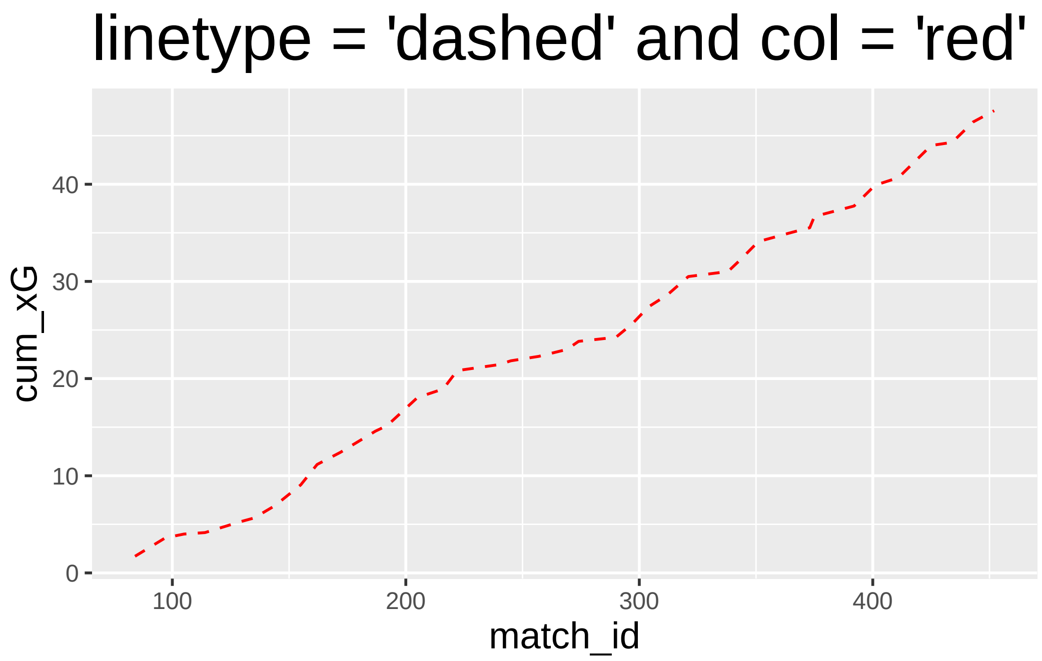
```
p_line + geom_line(aes(x=match_id, y=cum_xG), linetype = '8f1f')+
  ggtitle("linetype = '8f1f'")
```



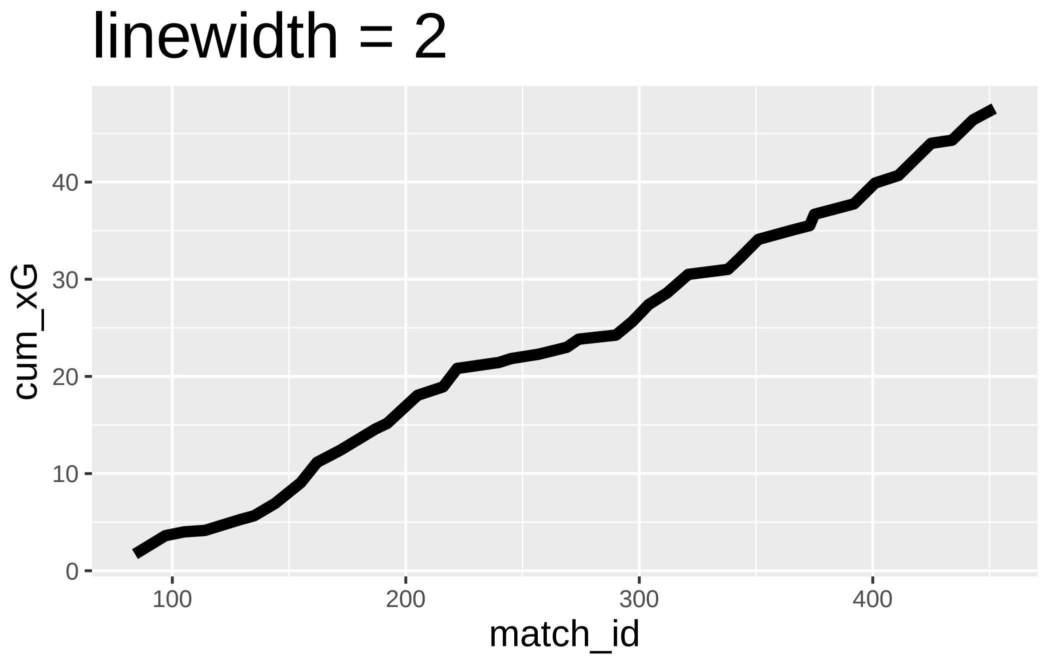
Color and linewidth

- `col` is specified in the same way as for `geom_point()`
- `linewidth` determines the width of lines
- due to historical error, unit of `linewidth` is roughly 0.75 mm
- using `size` for linewidth currently still works but is deprecated and is not recommended
- `lineend` and `linejoin` are further aesthetics

```
p_line + geom_line(aes(x=match_id, y=cum_xG), linetype = 'dashed', col = "red") +
  ggtitle("linetype = 'dashed' and col = 'red'")
```



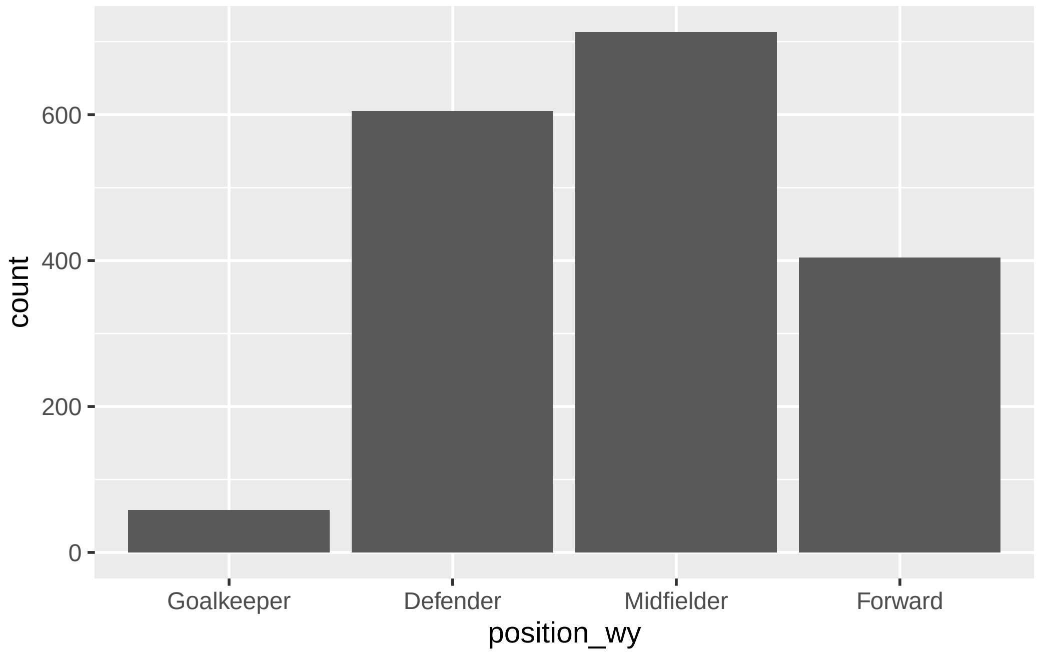
```
p_line + geom_line(aes(x=match_id, y=cum_xG), linewidth = 2)+
  ggtitle("linewidth = 2")
```



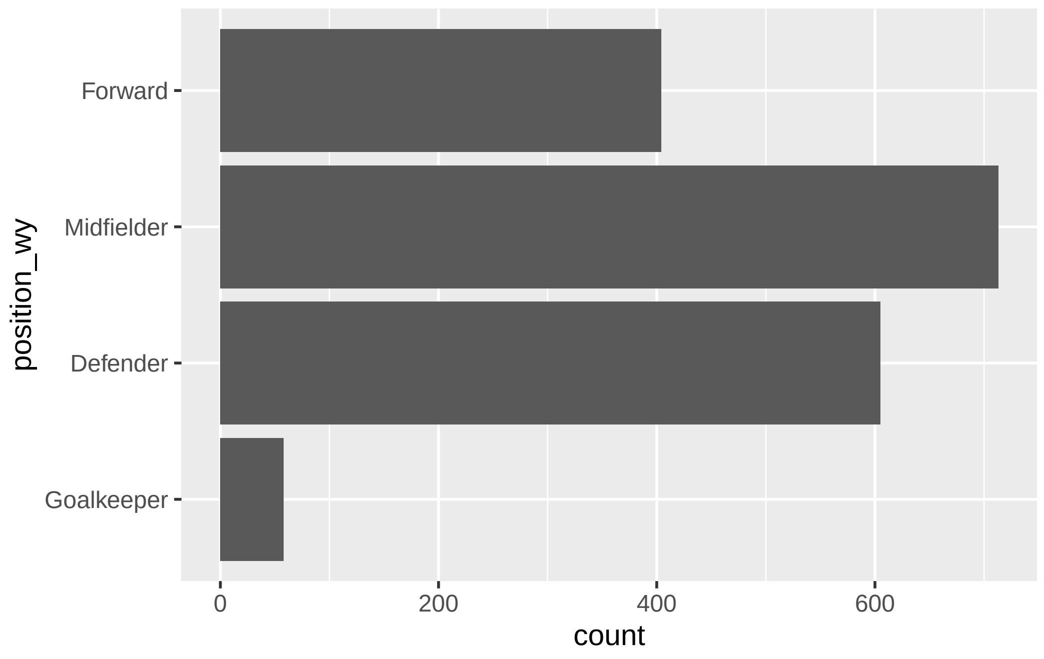
geom_bar

- `geom_bar()` creates a bar plot
- per default, it accepts only an x **or** y aesthetic and calculates the number of observations per group for the other one
- depending on which aesthetic is specified, the bars appear horizontal or vertical

```
fb.europe.players1718 %>%
  ggplot() +
  geom_bar(aes(x=position_wy))
```

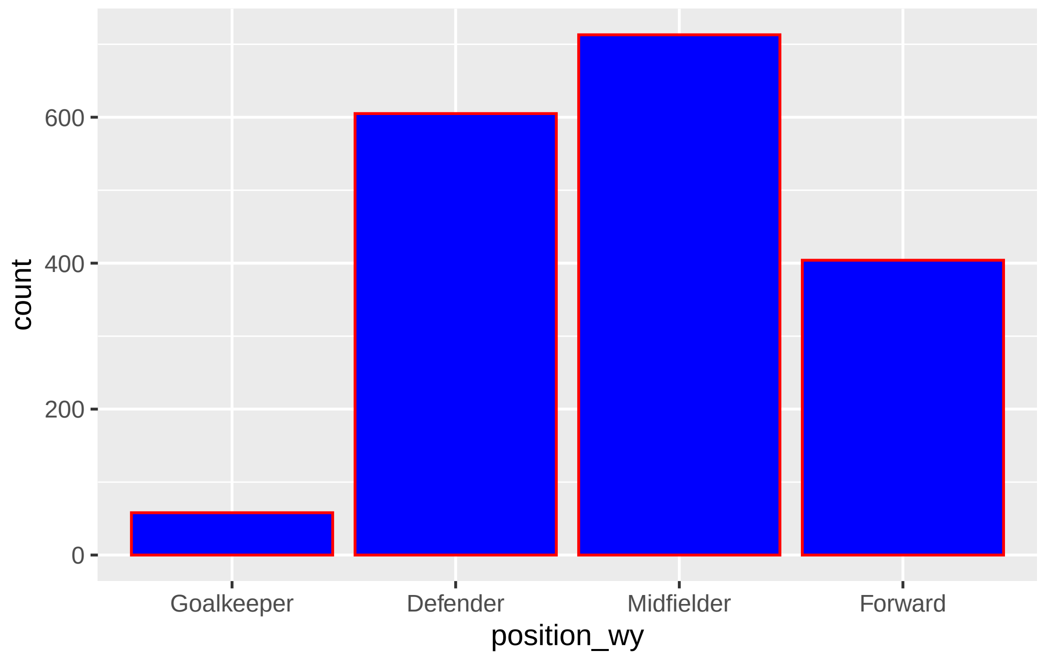


```
fb.europe.players1718 %>%
  ggplot() +
  geom_bar(aes(y=position_wy))
```

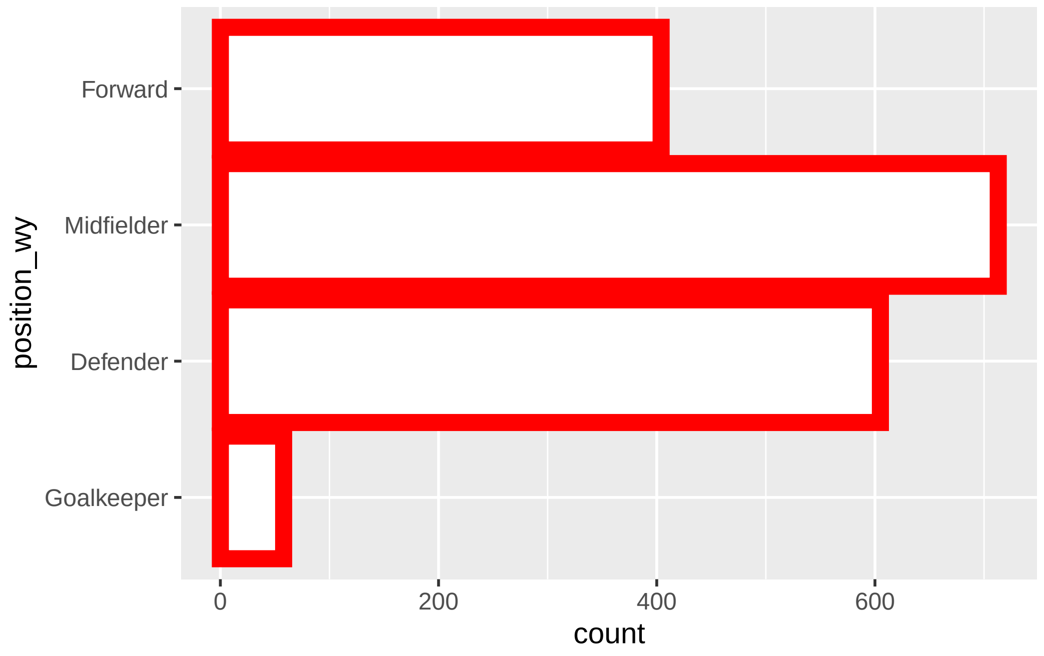


- for changing the color of the bars, `geom_bar()` accepts `col` for its border and `fill` for the bar area
- specifying `linetype` changes the appearance of the lines around the bars and `size` changes their thickness
- `width` modifies bar width

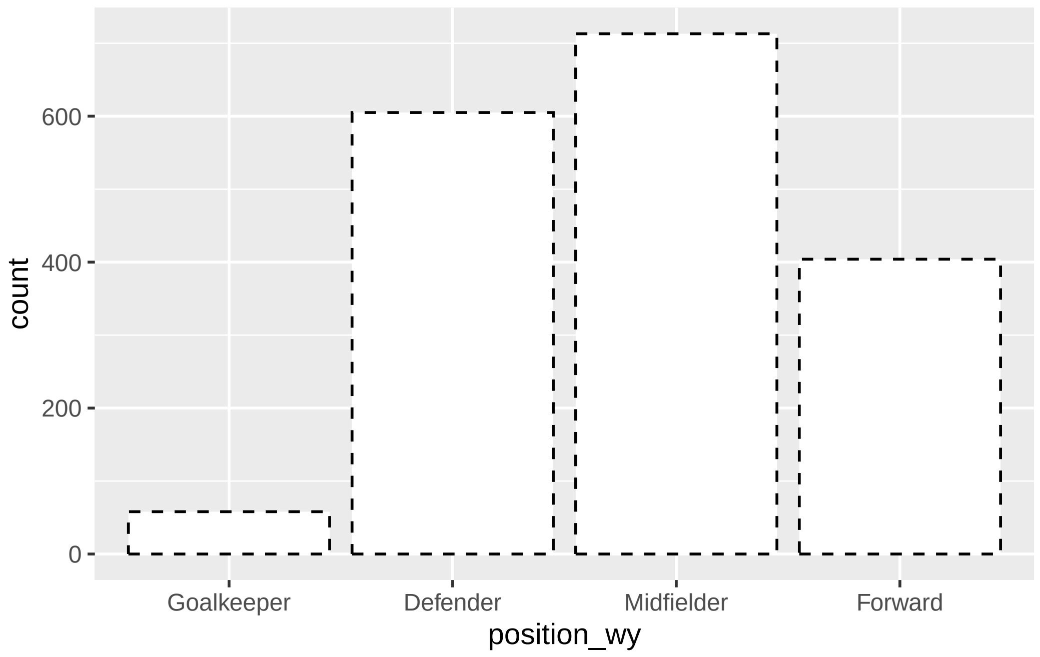
```
fb.europe.players1718 %>%
  ggplot() +
  geom_bar(aes(x=position_wy), fill = "blue", col = "red")
```



```
fb.europe.players1718 %>%
  ggplot() +
  geom_bar(aes(y=position_wy), fill = "white", col = "red", size = 3)
```



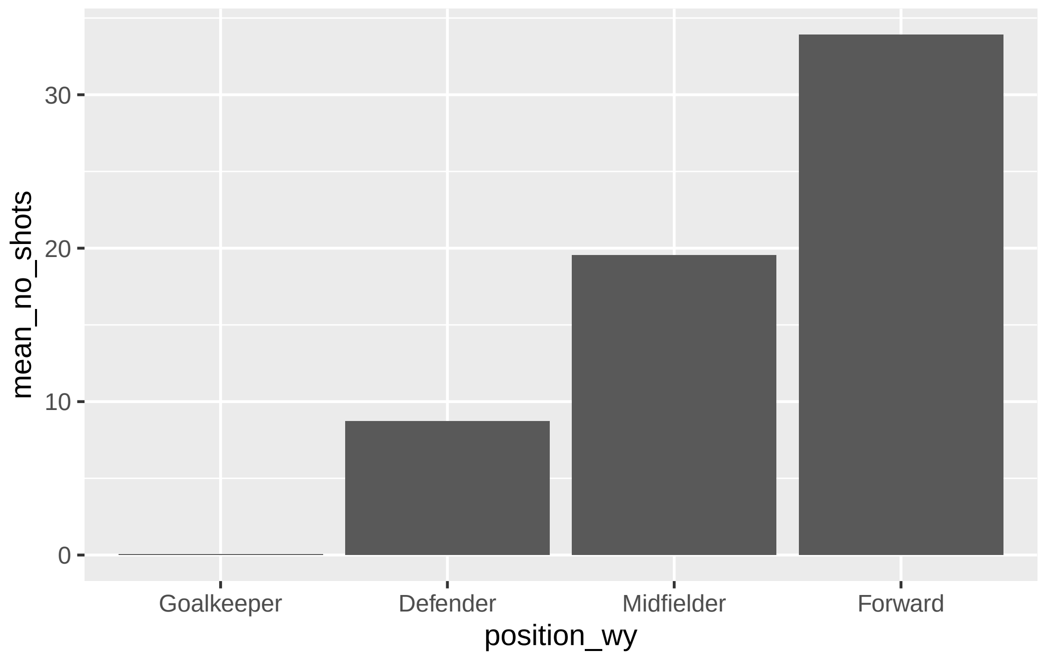
```
fb.europe.players1718 %>%
  ggplot() +
  geom_bar(aes(x=position_wy), fill = "white", col = "black", linetype = "dashed")
```



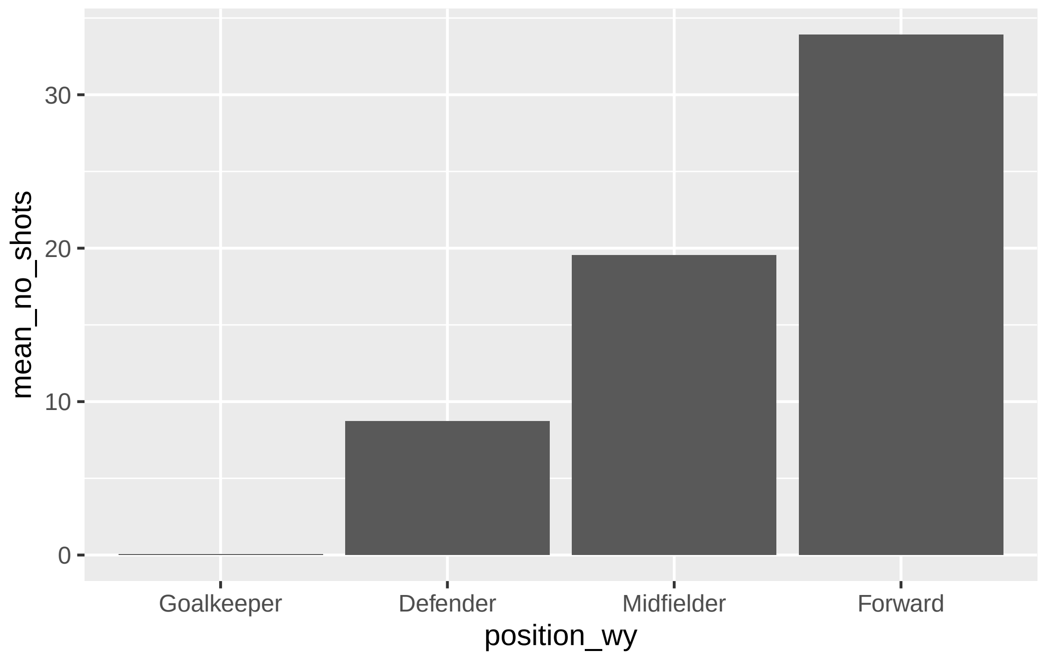
geom_bar vs. geom_col

- the reason `geom_bar()` accepts only one position aesthetic and displays number of occurrence on the other is that it includes `stat = "count"` per default
- if the height of the bar shall indicate a numeric value, one can add a second position aesthetic and specify `stat = "identity"`
- using `geom_col()` instead of `geom_bar()` also leads to this behavior
- however, bar plots should only be used to indicate counts and proportions, not continuous variables

```
fb.europe.players1718 %>%
  group_by(position_wy) %>%
  summarise(
    mean_no_shots = mean(n_shots)
  ) %>%
  ggplot() +
  geom_bar(aes(x=position_wy, y=mean_no_shots), stat = "identity")
```



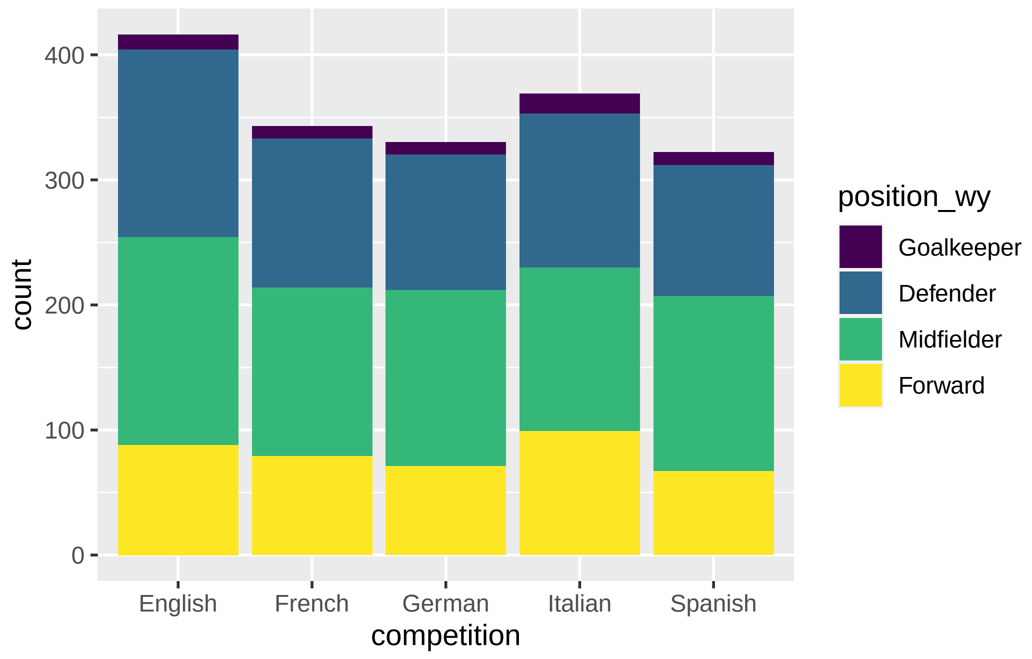
```r
fb.europe.players1718 %>%
  group_by(position_wy) %>%
  summarise(
    mean_no_shots = mean(n_shots)
  ) %>%
  ggplot() +
  geom_col(aes(x=position_wy, y=mean_no_shots))
```



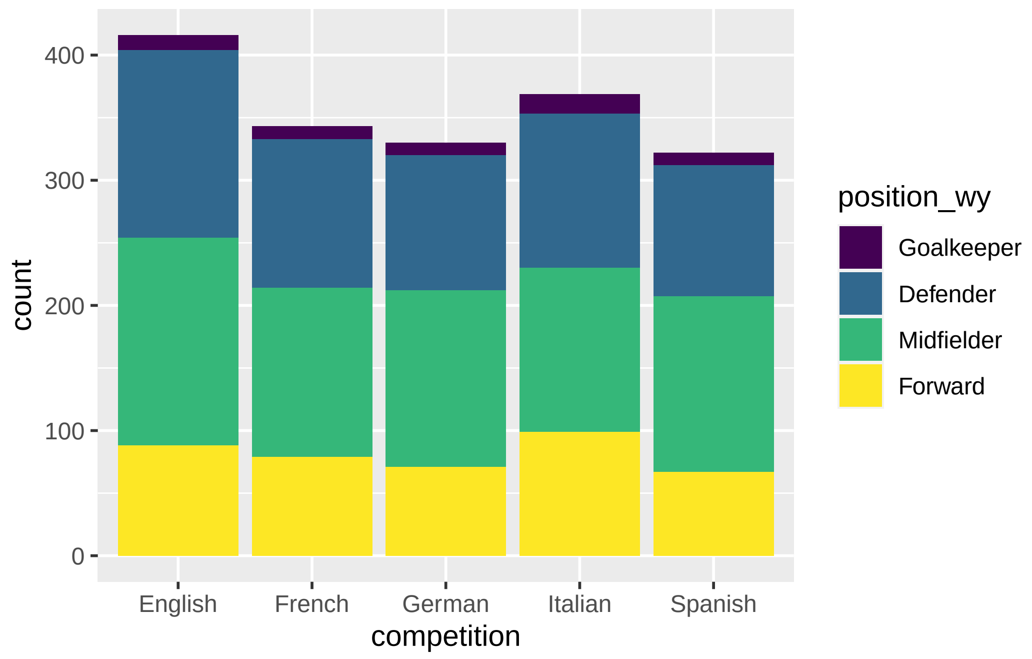
Groups in geom_bar

- sometimes, the relationship between two discrete variables shall be expressed by comparing distributions of one variable between groups of the other variable
- e.g., we could be interested in the distribution of playing positions per competition
- this can be achieved by using **grouped bar plots**
- to create a grouped bar plot we have to specify the `col` or `fill` argument to `geom_bar()`
- the relative positioning of the bars within one factor level can be varied; it defaults to `position = "stack"` which stacks the absolute counts on top of each other
- if we are interested in discrete distributions across factor levels, we might want to look at relative proportions
- this can be achieved by specifying `position = "fill"`
- as an alternative, the bars can be placed as a group next to each other with `position = "dodge"`

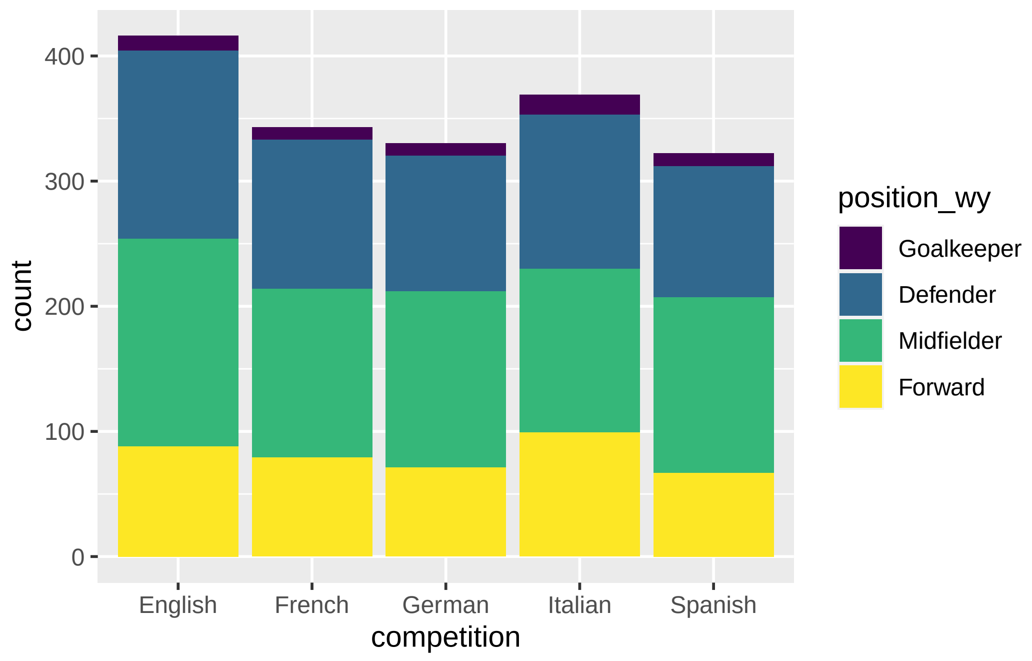
```r
fb.europe.players1718 %>%
  mutate(competition = gsub(" first division", "", competition)) %>%
  ggplot() +
  geom_bar(aes(x=competition, fill = position_wy))
```



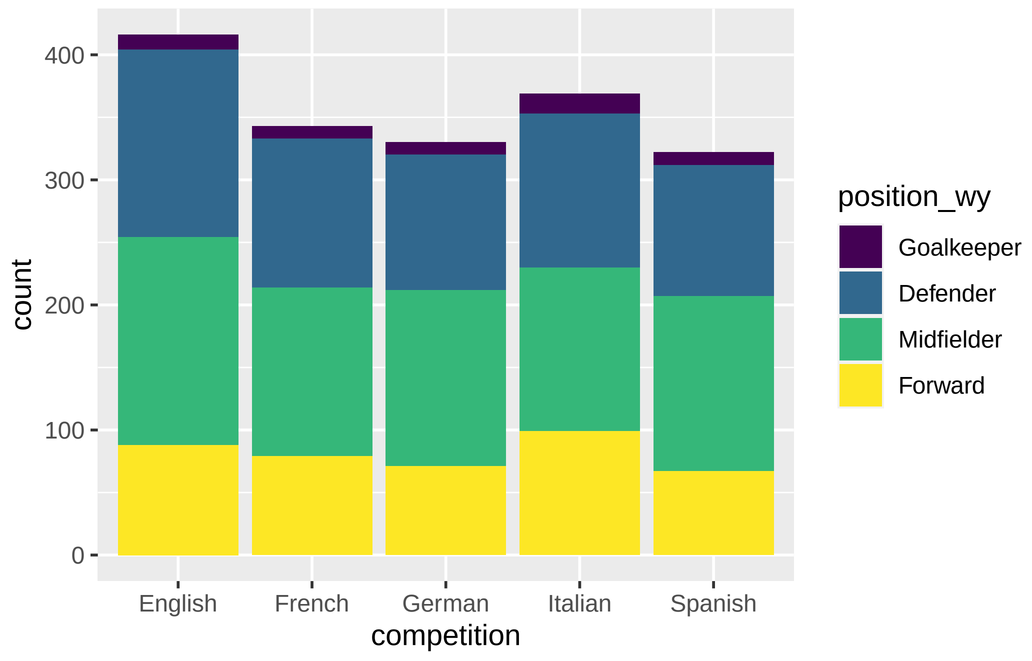
```
fb.europe.players1718 %>%
  mutate(competition = gsub(" first division", "", competition)) %>%
  ggplot() +
  geom_bar(aes(x=competition, fill = position_wy), position = "stack")
```



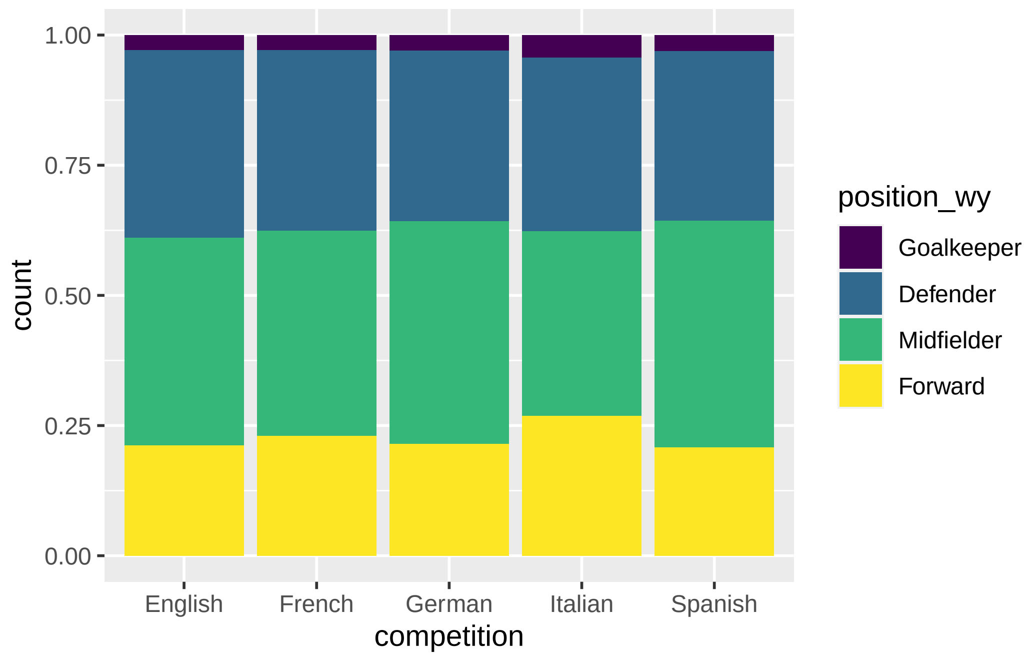
```r
fb.europe.players1718 %>%
  mutate(competition = gsub(" first division", "", competition)) %>%
  ggplot() +
  geom_bar(aes(x=competition, fill = position_wy))
```



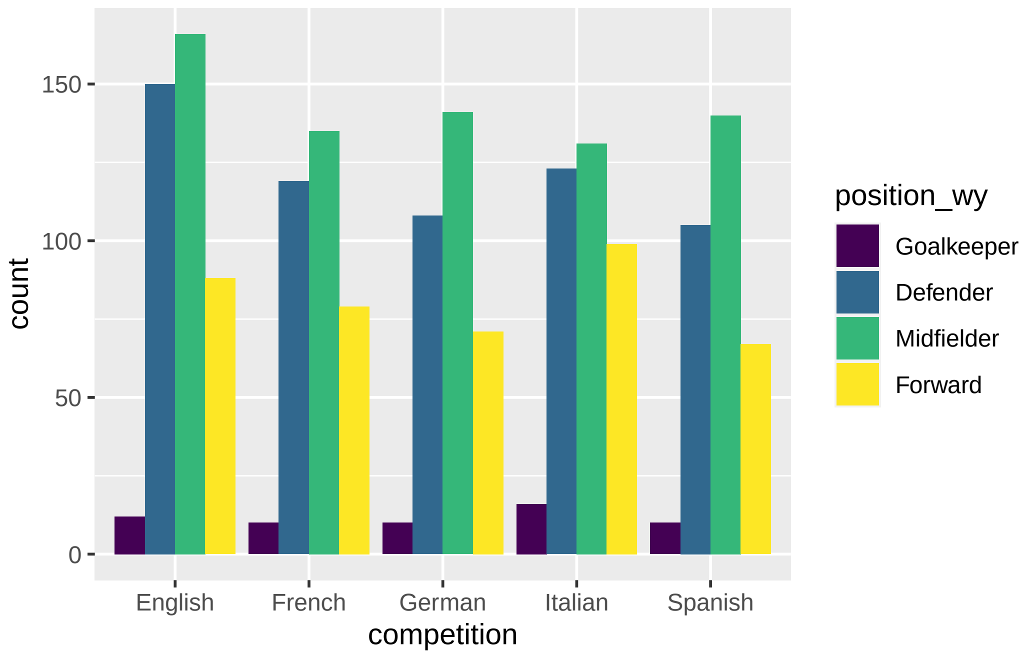
```r
fb.europe.players1718 %>%
  mutate(competition = gsub(" first division", "", competition)) %>%
  ggplot() +
  geom_bar(aes(x=competition, fill = position_wy), position = "stack")
```



```
fb.europe.players1718 %>%
  mutate(competition = gsub(" first division", "", competition)) %>%
  ggplot() +
  geom_bar(aes(x=competition, fill = position_wy), position = "fill")
```



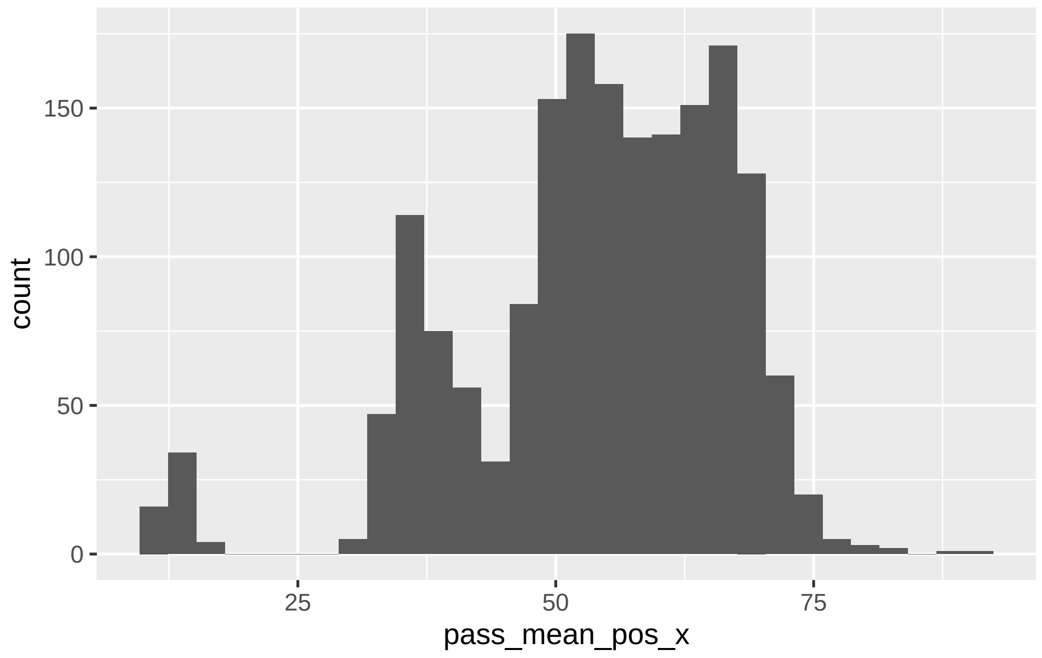
```r
fb.europe.players1718 %>%
  mutate(competition = gsub(" first division", "", competition)) %>%
  ggplot() +
  geom_bar(aes(x=competition, fill = position_wy), position = "dodge")
```



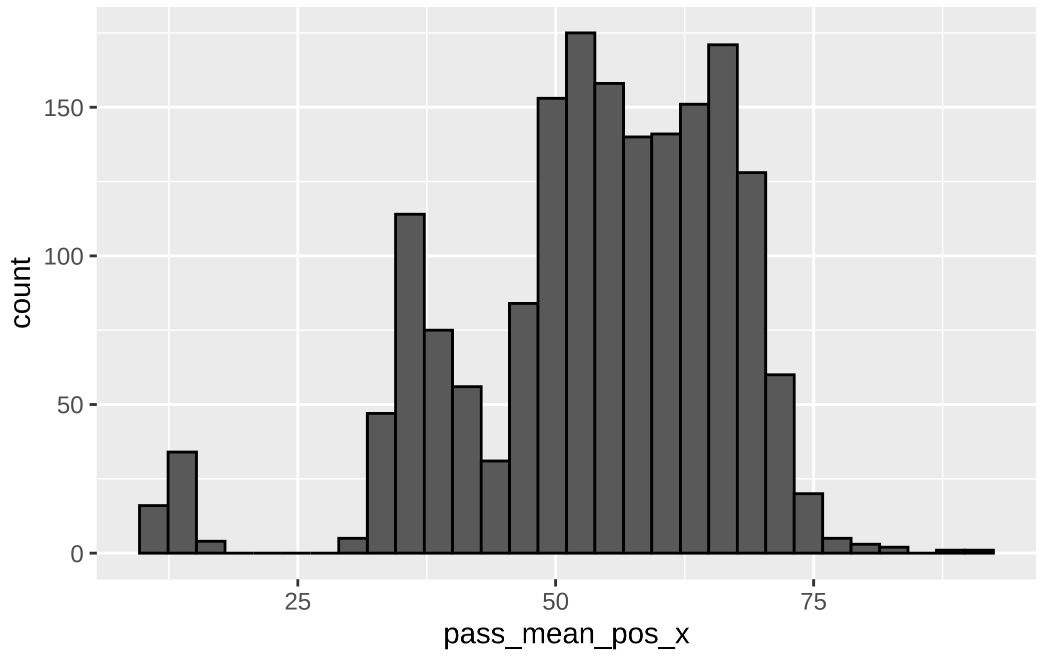
geom_histogram

- if we want to visualize the distribution of a continuous variable, a histogram is the plot of choice
- we can create histograms by passing the continuous variable to the `x` (or `y`) aesthetic and calling `geom_histogram()`
- very important to specify `bins` or `binwidth` parameter according to the use case (default `bins = 30`)
- choosing a different `col` than default might help in disentangling bins

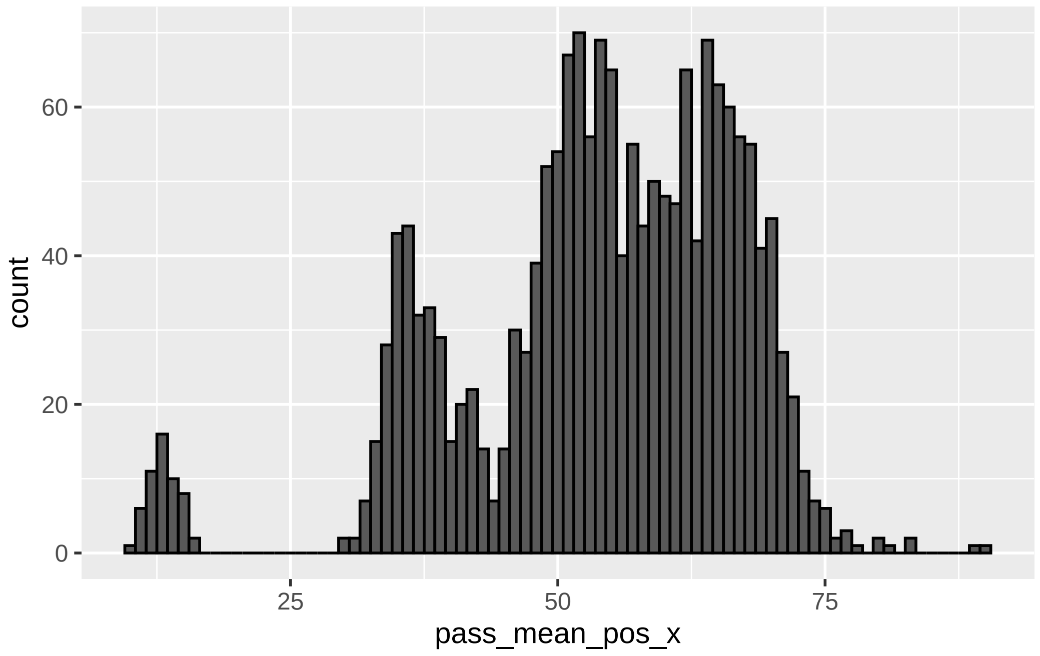
```r
fb.europe.players1718 %>%
  ggplot(aes(x=pass_mean_pos_x)) +
  geom_histogram()
```



```r
fb.europe.players1718 %>%
  ggplot(aes(x=pass_mean_pos_x)) +
  geom_histogram(col = "black")
```



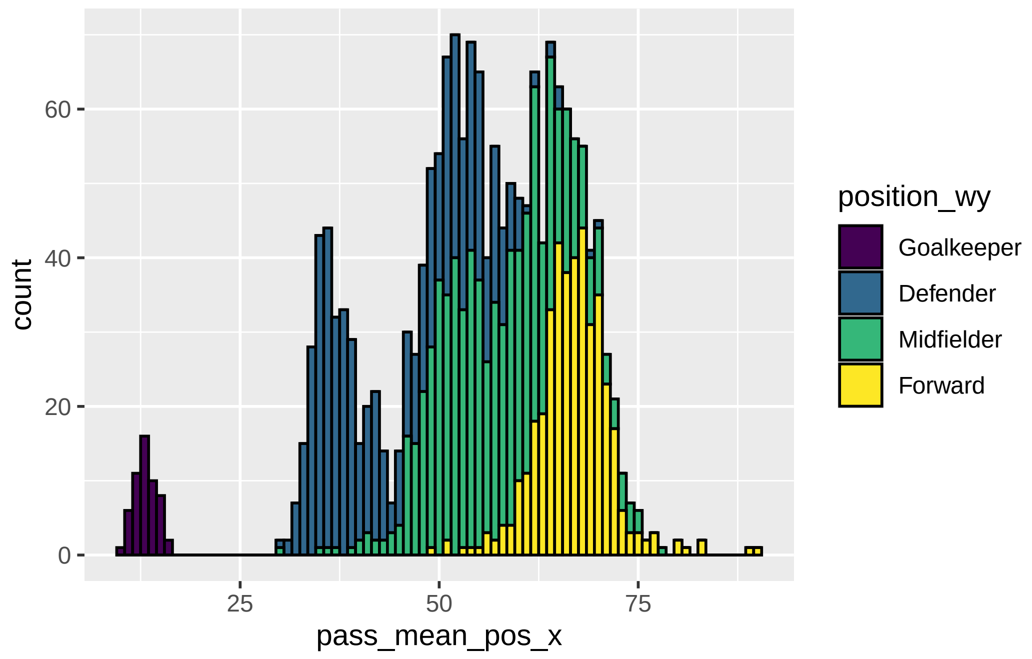
```
fb.europe.players1718 %>%
  ggplot(aes(x=pass_mean_pos_x)) +
  geom_histogram(col = "black", binwidth = 1)
```



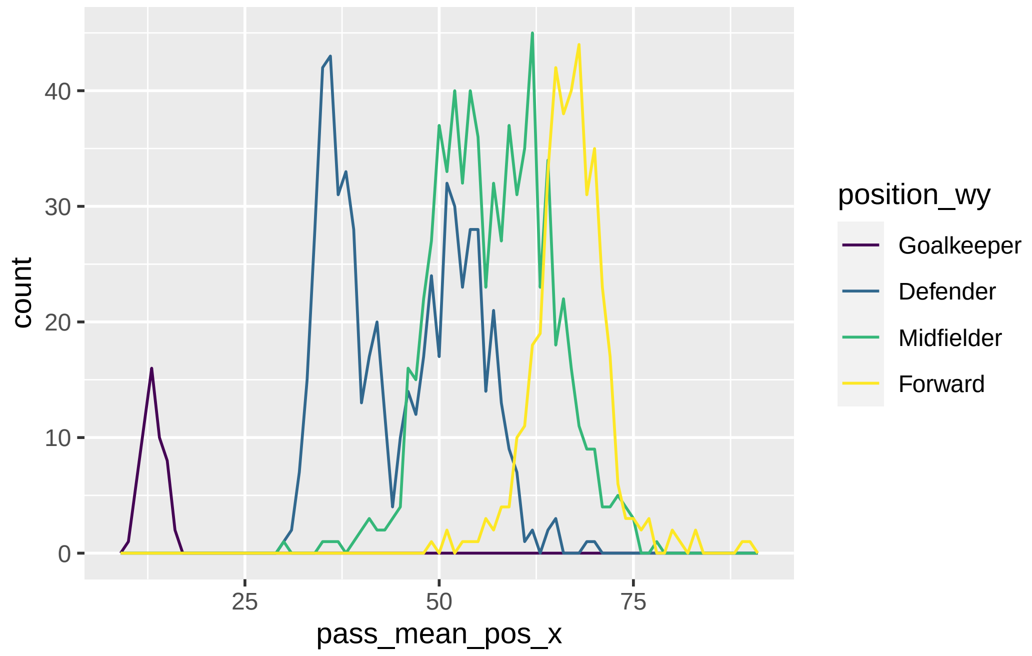
Comparing distributions with geom_histogram and geom_freqpoly

- just like with other geoms, `geom_histogram()` accepts variables for the `fill` parameter to visualize groups
- if distributions overlap too strongly, `geom_freqpoly()` might be a better choice (however, here `col` specifies groups)

```
fb.europe.players1718 %>%
  ggplot() +
  geom_histogram(aes(x=pass_mean_pos_x, fill = position_wy), col = 'black', binwidth = 1)
```



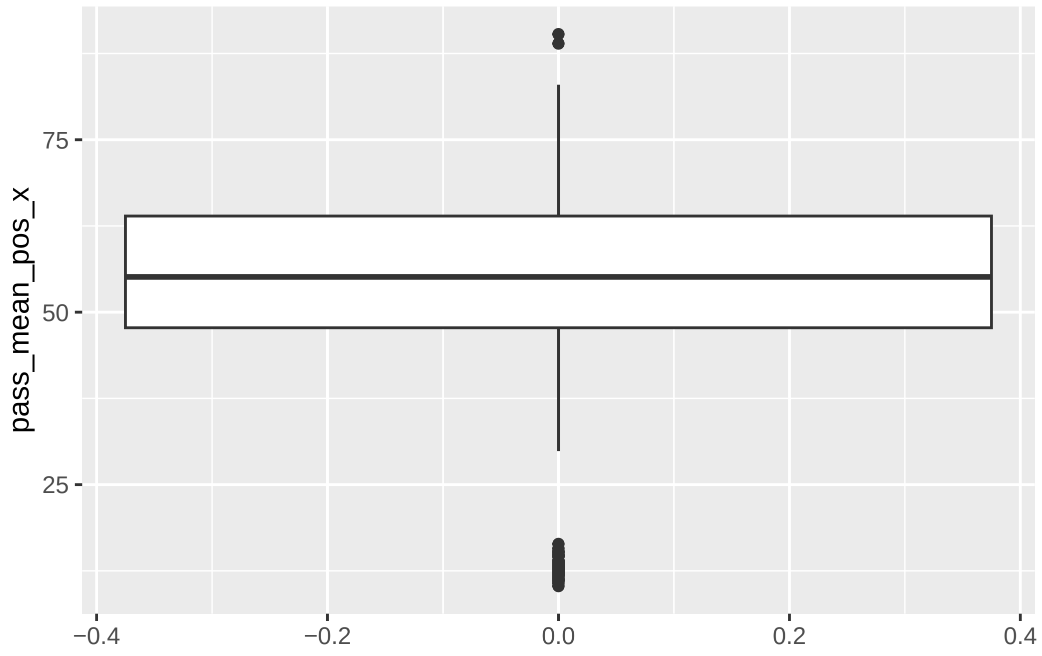
```
fb.europe.players1718 %>%
  ggplot() +
  geom_freqpoly(aes(x=pass_mean_pos_x, col = position_wy), binwidth = 1)
```



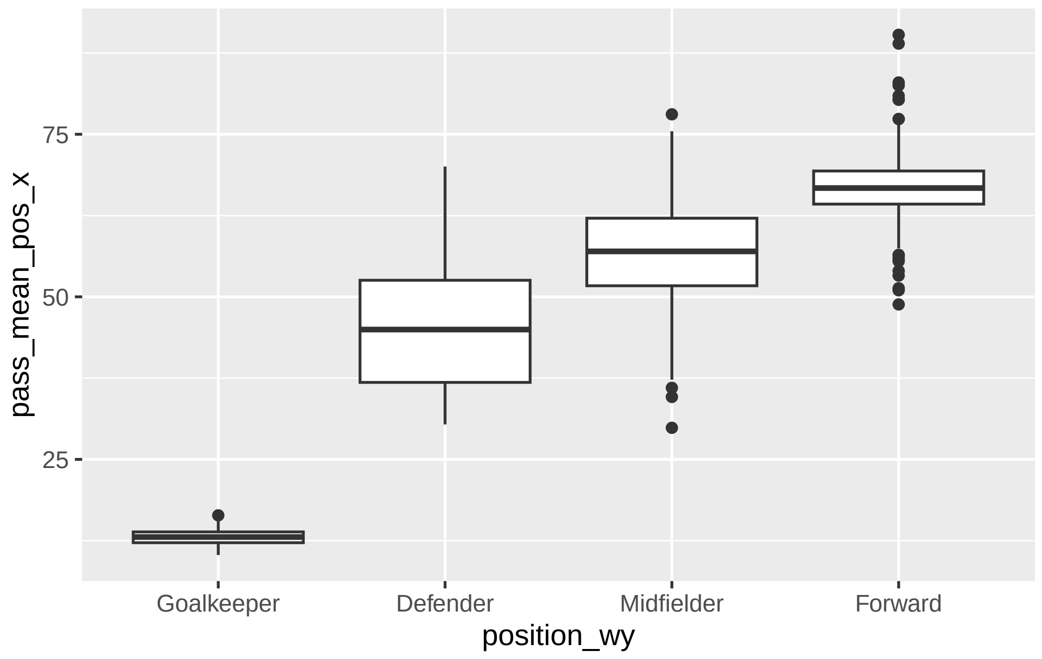
geom_boxplot

- another great option to visualize distributions and especially compare them in a space-efficient manner are box plots
- a univariate box plot is called using `geom_boxplot()` and specifying either `x` or `y` aesthetic
- if a discrete variable is given for the other position aesthetic, box plots are created per group
- if we want to add another variable, we can do so by specifying `col` or `fill` aesthetics as well

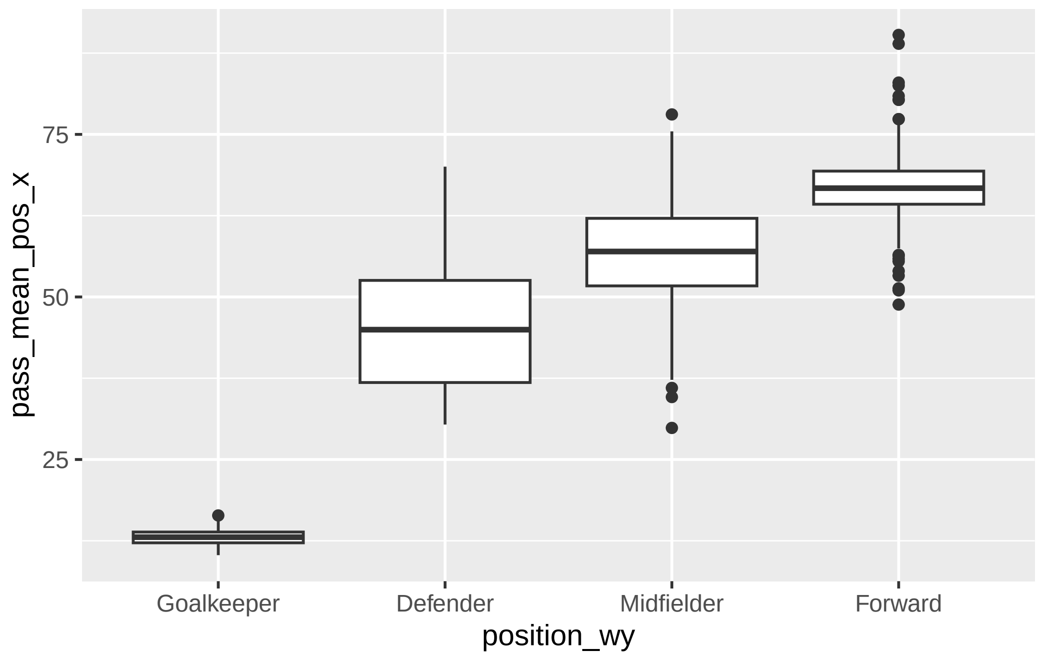
```
fb.europe.players1718 %>%
  ggplot() +
  geom_boxplot(aes(y = pass_mean_pos_x))
```



```
fb.europe.players1718 %>%
  ggplot() +
  geom_boxplot(aes(x = position_wy, y = pass_mean_pos_x))
```



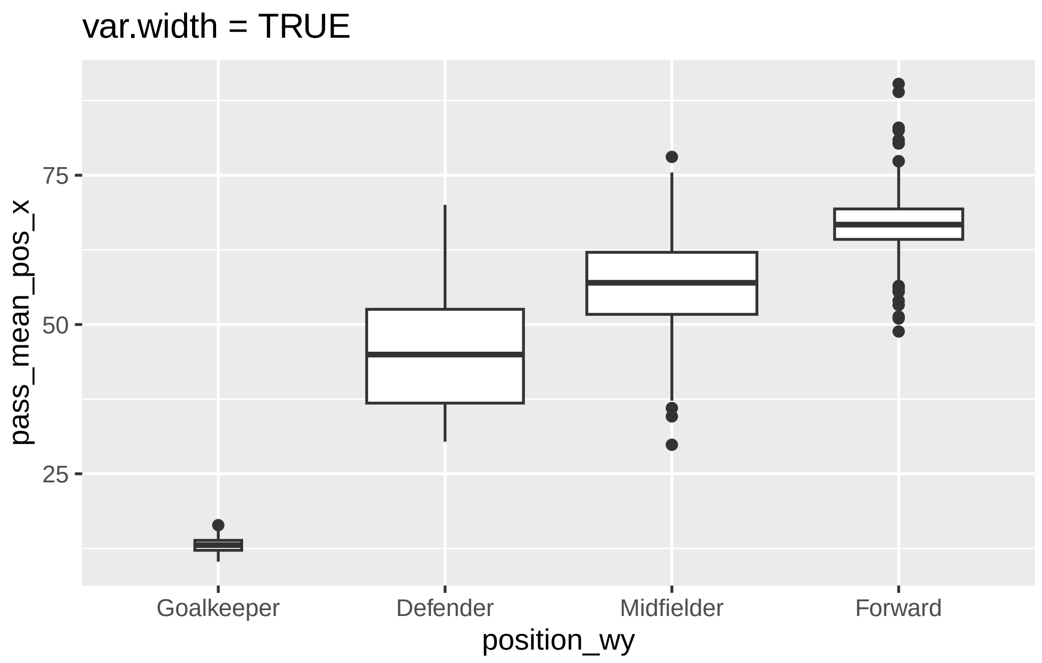
```
fb.europe.players1718 %>%
  ggplot() +
  geom_boxplot(aes(x = position_wy, y = pass_mean_pos_x, fill = foot))
```



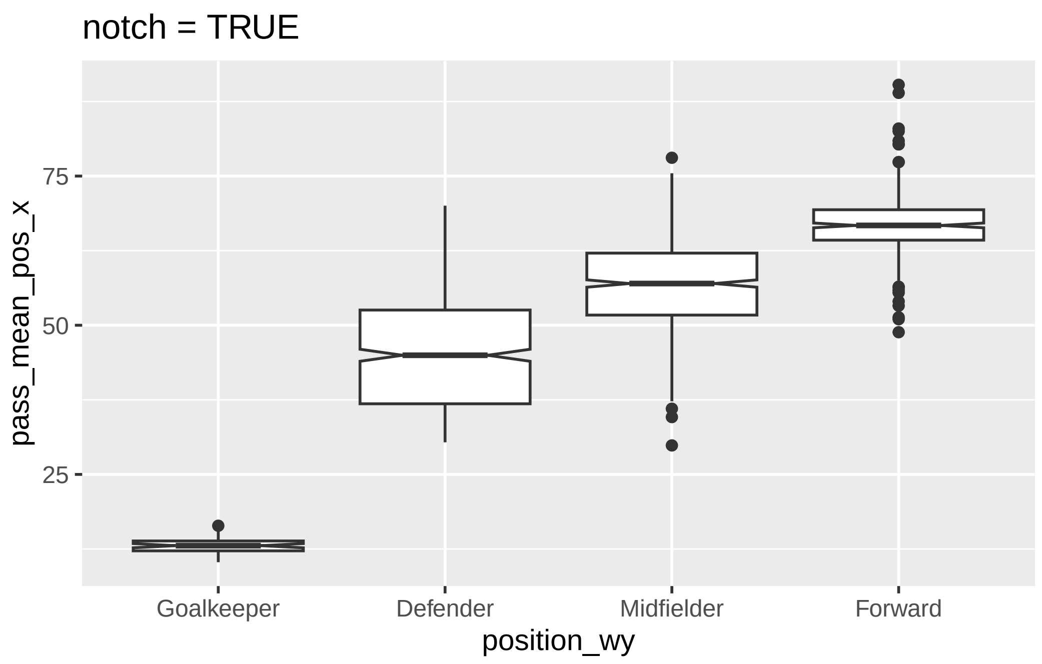
Variable width and notches

- box plots per default display five statistics of a distribution
- McGill and colleagues introduced two powerful extension to the box plot:
  - indicating sample size per group by means of variable box widths
  - indicating robustness of the median estimate by means of notches
- specifying `var_width = TRUE` and `notch = TRUE` activates these behaviors respectively

```
fb.europe.players1718 %>%
  ggplot() +
  geom_boxplot(aes(x = position_wy, y = pass_mean_pos_x, fill = foot), varwidth = TRUE) +
  ggtitle("var.width = TRUE")
```



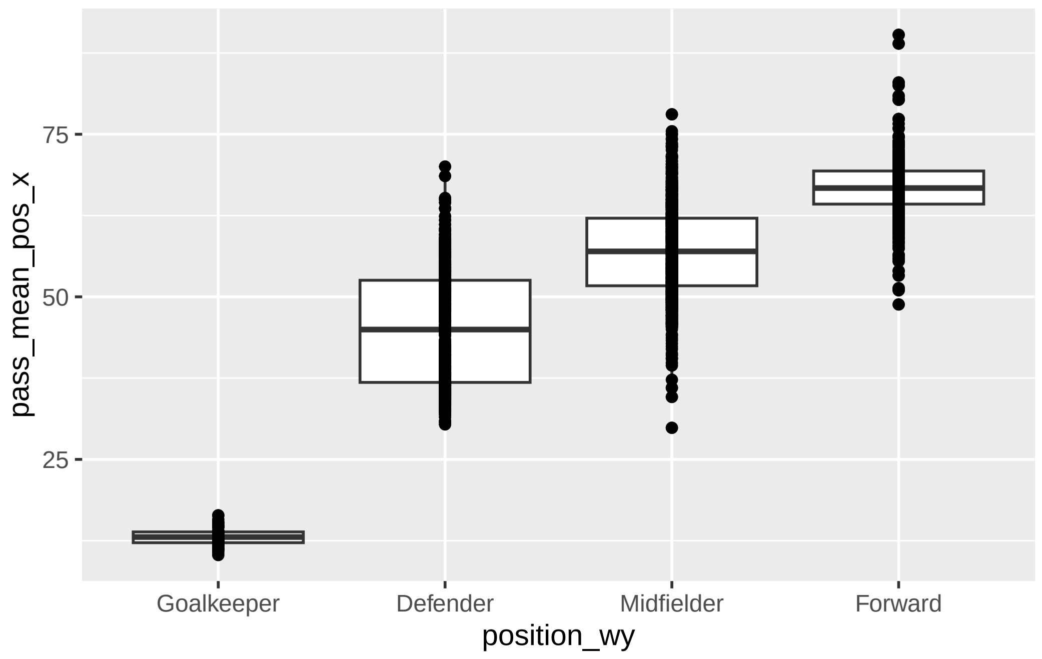
```
fb.europe.players1718 %>%
  ggplot() +
  geom_boxplot(aes(x = position_wy, y = pass_mean_pos_x, fill = foot), notch = TRUE) +
  ggtitle("notch = TRUE")
```



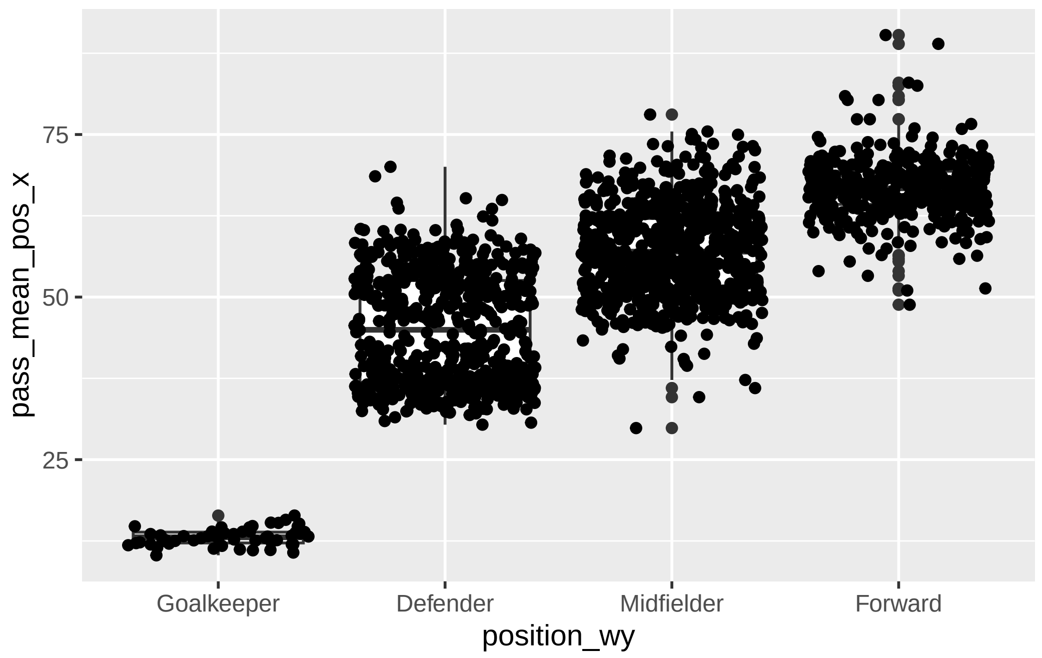
Combination with scatter plots

- box plots generally allow to add the raw data to the plot, which might give even more transparency to the storytelling
- a layer of `geom_point()` might be added to the plot
- however, introducing some horizontal jitter will help to disentangle observations
- this works well only for smaller datasets

```r
fb.europe.players1718 %>%
  ggplot(aes(x=position_wy, y=pass_mean_pos_x)) +
  geom_boxplot() +
  geom_point()
```



```
fb.europe.players1718 %>%
  ggplot(aes(x=position_wy, y=pass_mean_pos_x)) +
  geom_boxplot() +
  geom_jitter()
```



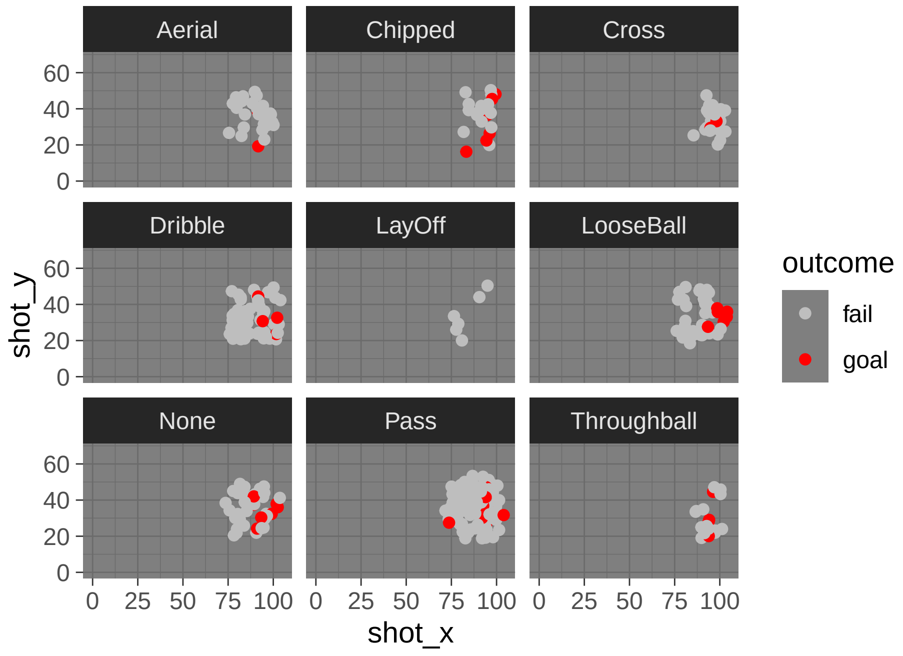
Grammar of Graphics - Recap

Component	ggplot2
Data	`data` argument
Aesthetic Mappings	`mappings` argument (specified in `aes()`)
Layers	mostly **geoms** (other layers possible)
Scales	`scale_*()` layers
Coordinate System	`coord_*()` layer
Faceting	`facet_wrap()` or `facet_grid()` layer
Theme	`theme()` layer

Building a plot layer by layer

Target plot:

```
fb.leicester.xg1516 %>%
  ggplot(aes(x=shot_x, y=shot_y)) +
  geom_point(aes(col=outcome)) +
  coord_fixed(ratio = 1, xlim = c(0,105), ylim = c(0,68)) +
  scale_color_discrete(type = c("grey", "red")) +
  facet_wrap(~lastAction) +
  theme_dark()
```

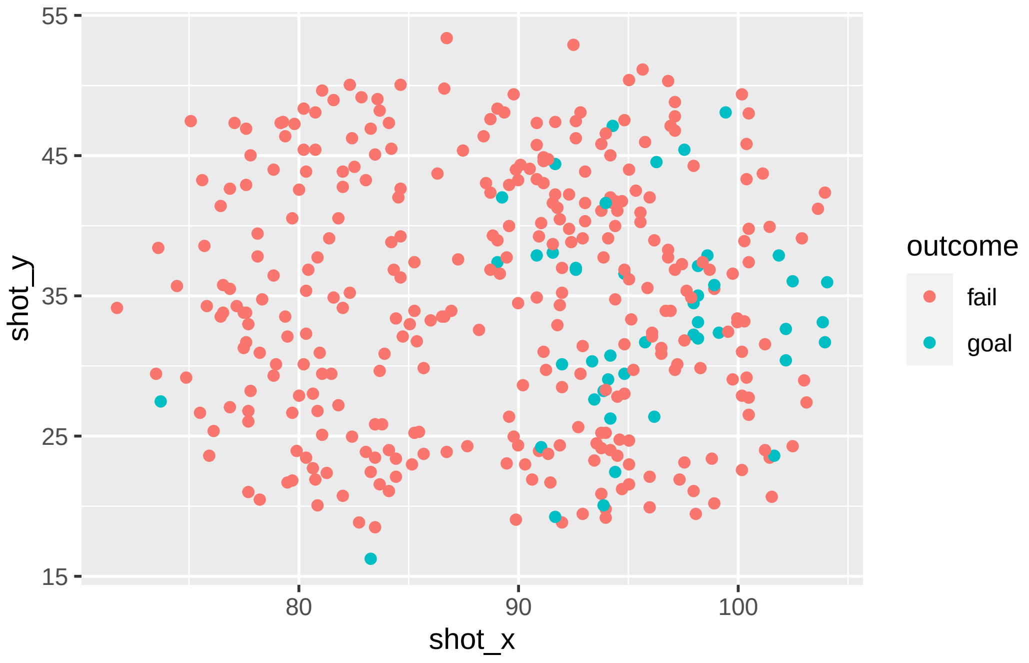


Scales

- scales are an important concept in ggplot2 and the grammar of graphics
- they control the mapping of data to aesthetics
- e.g., in the following code, we specified an `x`, `y`, and `col` aesthetic
- the x position scale tells ggplot how to translate the variable `shot_x` to a position on the x axis
- the y position scale tells ggplot how to translate the variable `shot_y` to a position on the y axis
- the colour scale tells ggplot how to color points according to the levels of `outcome`
- scales also define the **guides**, i.e. axes and legends

We map the x- and y-coordinates of shots to the x- and y-aesthetic of the plot. Then, we map a variable (`outcome`) to the color aesthetic. If we are not specifying any scales, we are using the default scales. For x- and y-aesthetics, this is a cartesian coordinate system. For color, it is the default color scale used in ggplot2:

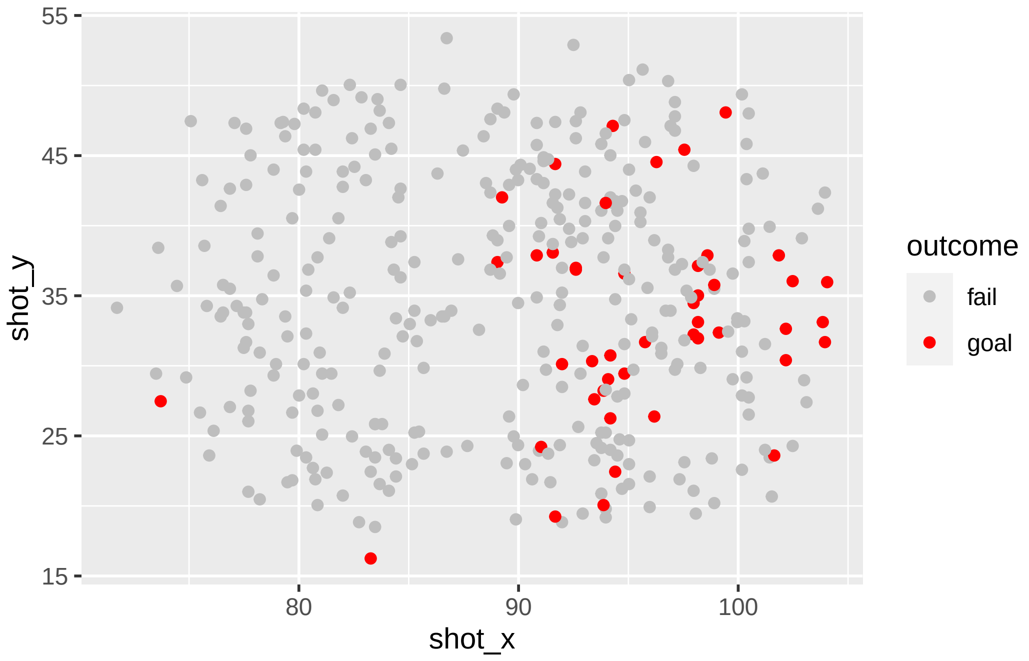
```
fb.leicester.xg1516 %>%
  ggplot(aes(x = shot_x, y = shot_y, col = outcome)) +
  geom_point()
```



- the mapping of `x` and `y` aesthetic is straightforward in a cartesian coordinate system (however, transformations like log-transform might be applied)
- scales also apply to all other aesthetics, such as size, shape, alpha

- often, the default options for scales are sufficient and do not have to be tweaked
- scales can be added as layers and ggplot and always take the form `scale_A_B`, where `A` refers to the aesthetic the scale is mapping and `B` refers to some kind of specification like `discrete` or `continuous`
- within the scale layer, the mapping can be controlled (by for example choosing custom colors for the color mapping)
- in order to change the **discrete** coloring of the points, we can use the `scale_color_discrete()` function:

```
fb.leicester.xg1516 %>%
  ggplot(aes(x = shot_x, y = shot_y, col = outcome)) +
  geom_point() +
  scale_color_discrete(type = c("grey", "red"))
```



Coordinate System

Coordinate systems have two main jobs in ggplot2:

- Combine the two position aesthetics to produce a 2d position on the plot
  - position aesthetics are called `x` and `y` but calling them position 1 and 2 is more general

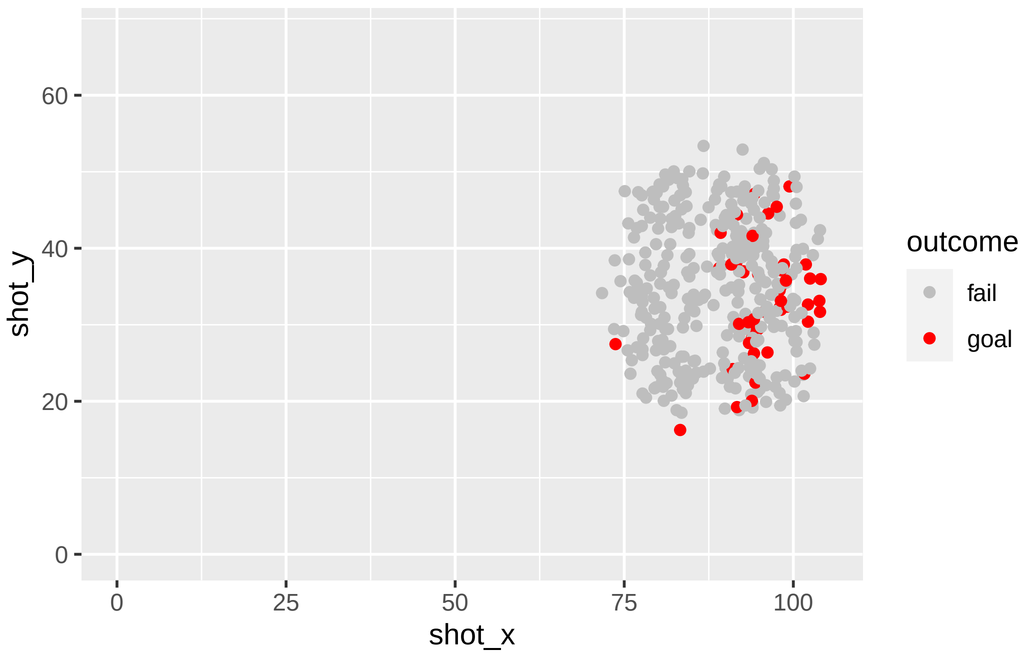
    - for example, with the polar coordinate system they become angle and radius and with maps they become latitude and longitude

- In coordination with the **faceter**, coordinate systems draw axes and panel backgrounds
    - while scales control the values that appear on the axes and mapping from data to position, the coordinate system actually draws them
    - this is because their appearance depends on the coordinate system

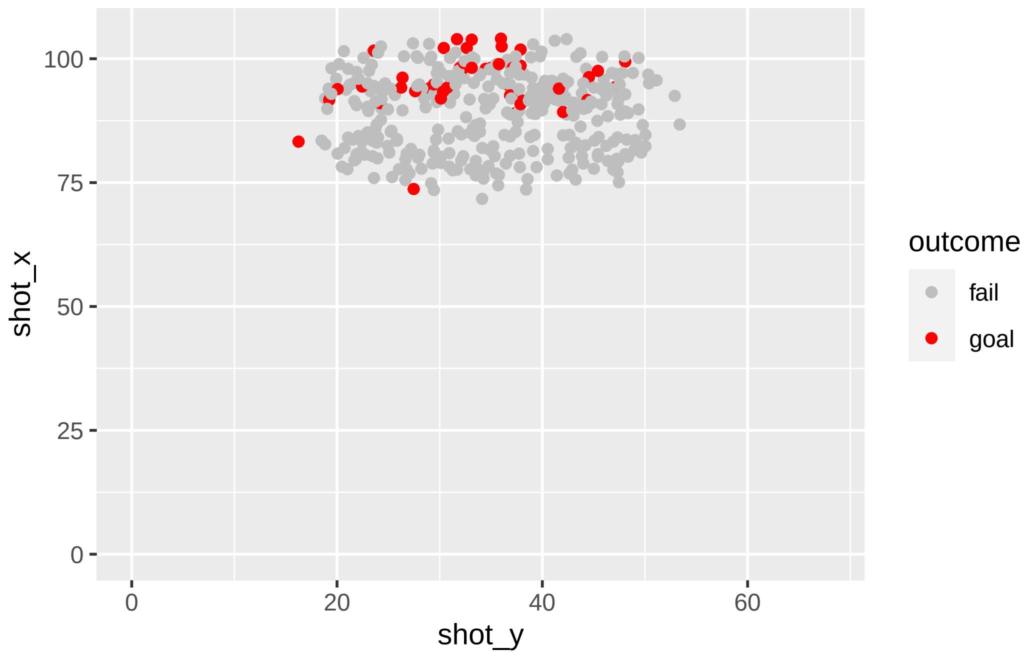
Three types of linear coordinate systems:

- `coord_cartesian()` allows zooming in or out by controlling `xlim` and `ylim`
- `coord_flip()` exchanges x and y axes
- `coord_fixed()` fixes the ratio of the length on x and y axis with the `ratio` parameter (default 1)

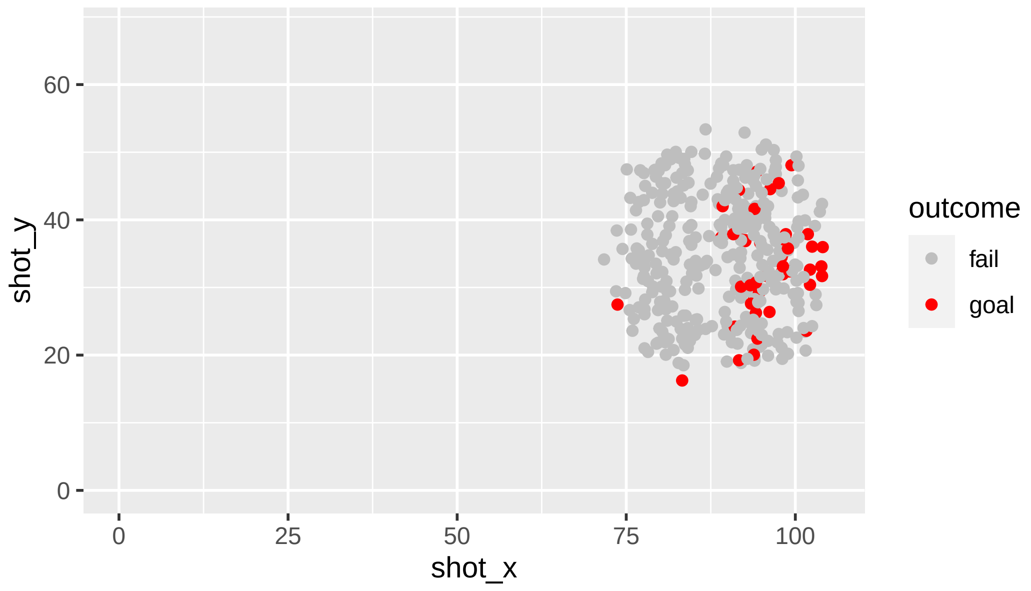
```
p <- fb.leicester.xg1516 %>%
  ggplot(aes(x = shot_x, y = shot_y, col = outcome)) +
  geom_point() +
  scale_color_discrete(type = c("grey", "red"))

p + coord_cartesian(xlim = c(0, 105), ylim = c(0,68))
```



```r
p + coord_flip(xlim = c(0, 105), ylim = c(0,68))
```



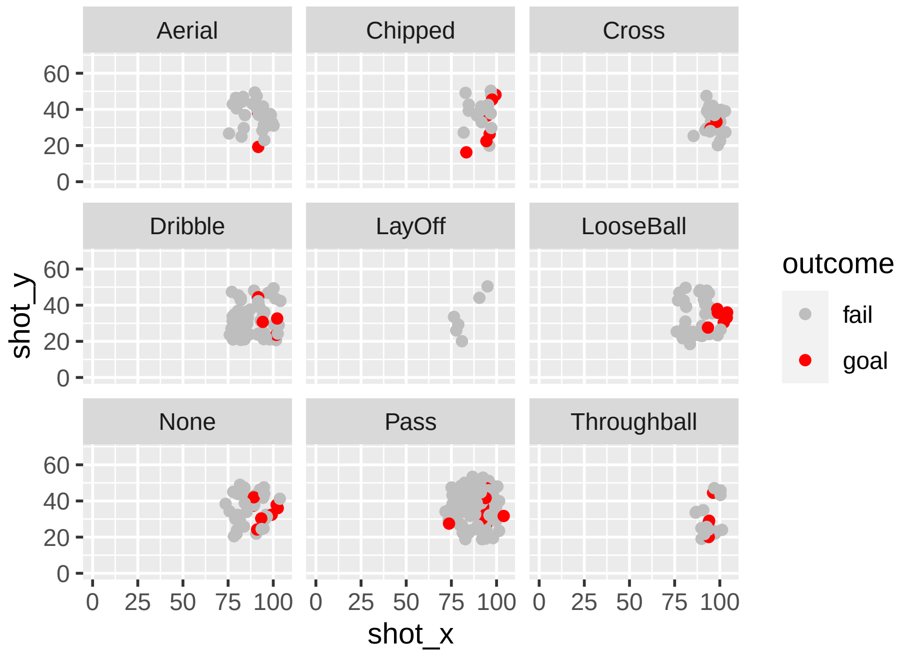
```r
p + coord_fixed(xlim = c(0, 105), ylim = c(0,68), ratio = 1)
```



Faceting

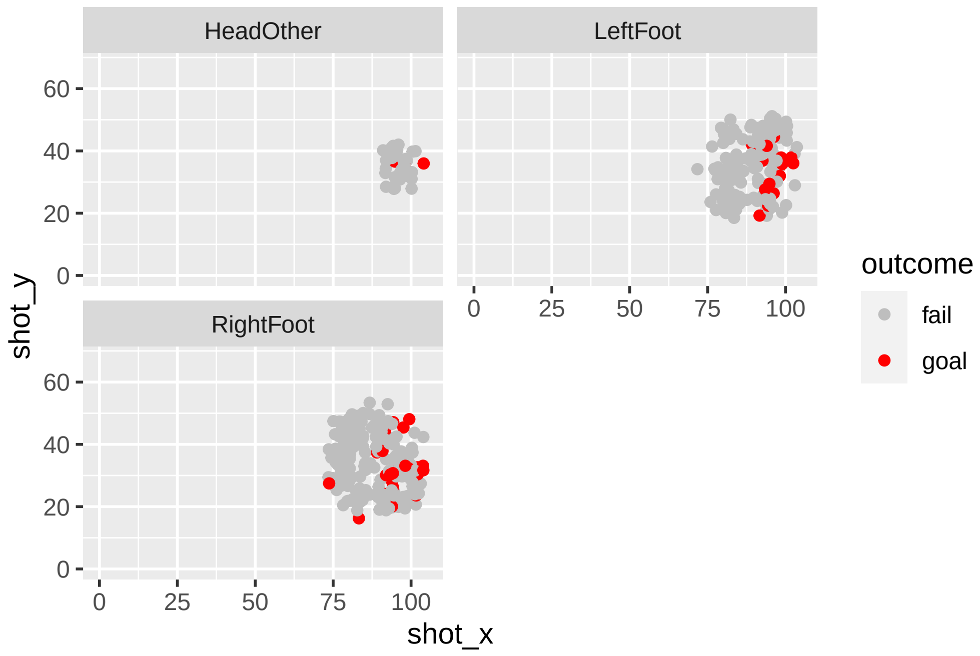
- using faceting, we can subset a plot based on discrete variables and display the subsets in distinct plots
- three types of faceting:
    - `facet_null()`: a single plot, the default
    - `facet_wrap()`: "wraps" a 1d ribbon of panels into 2d
    - `facet_grid()`: produces a 2d grid of panels defined by variables which form the rows and columns

```
fb.leicester.xg1516 %>%
  ggplot(aes(x = shot_x, y = shot_y, col = outcome)) +
  geom_point() +
  scale_color_discrete(type = c("grey", "red")) +
  coord_fixed(xlim = c(0,105), ylim= c(0,68)) +
  facet_wrap(~lastAction)
```

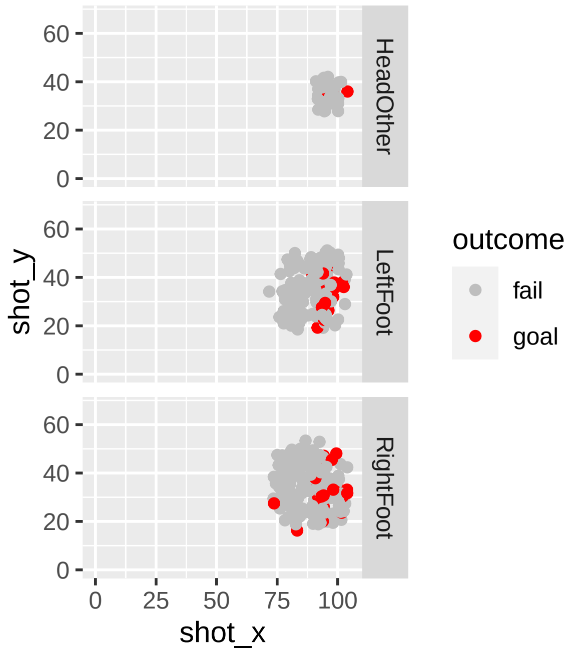


- using `facet_wrap()`, specifying `ncol` controls the number of columns to wrap the panels into
- using `facet_grid()`, if only one variable shall be used, "." replaces the other one

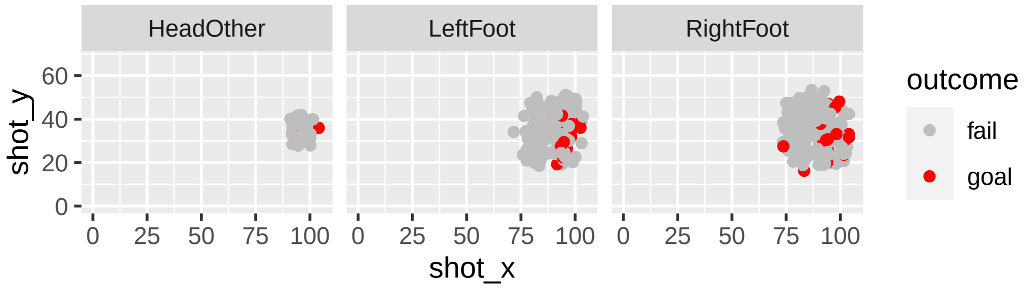
```r
p <- fb.leicester.xg1516 %>%
  ggplot(aes(x = shot_x, y = shot_y, col = outcome)) +
  geom_point() +
  scale_color_discrete(type = c("grey", "red")) +
  coord_fixed(xlim = c(0,105), ylim= c(0,68))

p + facet_wrap(~shotType, ncol = 2)
```



```r
p + facet_grid(shotType~.)
```



```
p + facet_grid(.~shotType)
```



Theme

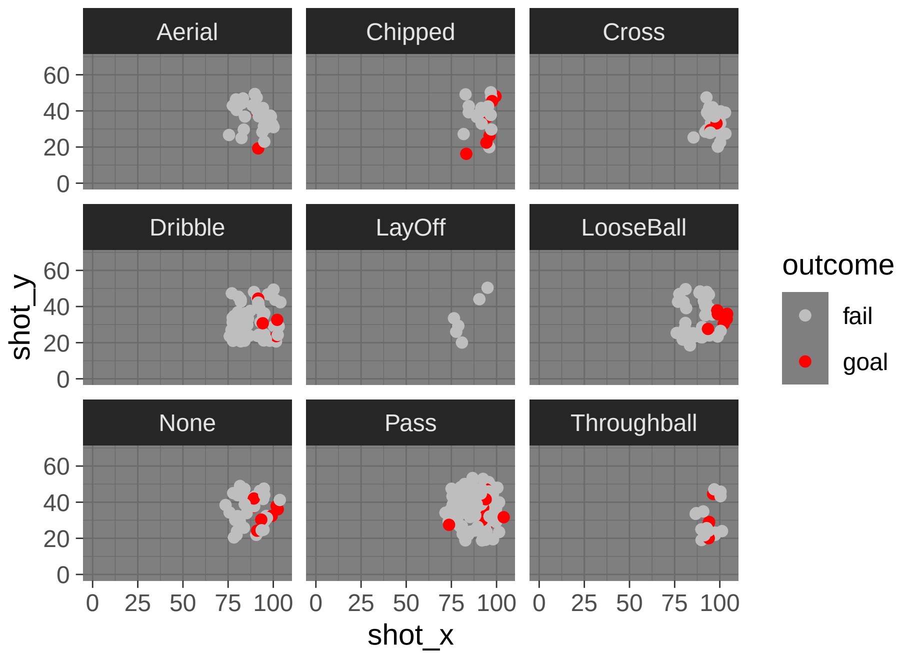
- theming system allows specification of all non-data elements
- allows specification of all theme elements, like fonts, backgrounds, colors
- customizing the theme yourself can get complex
- there are some complete **themes**, like `theme_grey()` (the default), which set all of the theme elements to values designed to work together harmoniously

built-in themes in ggplot2:

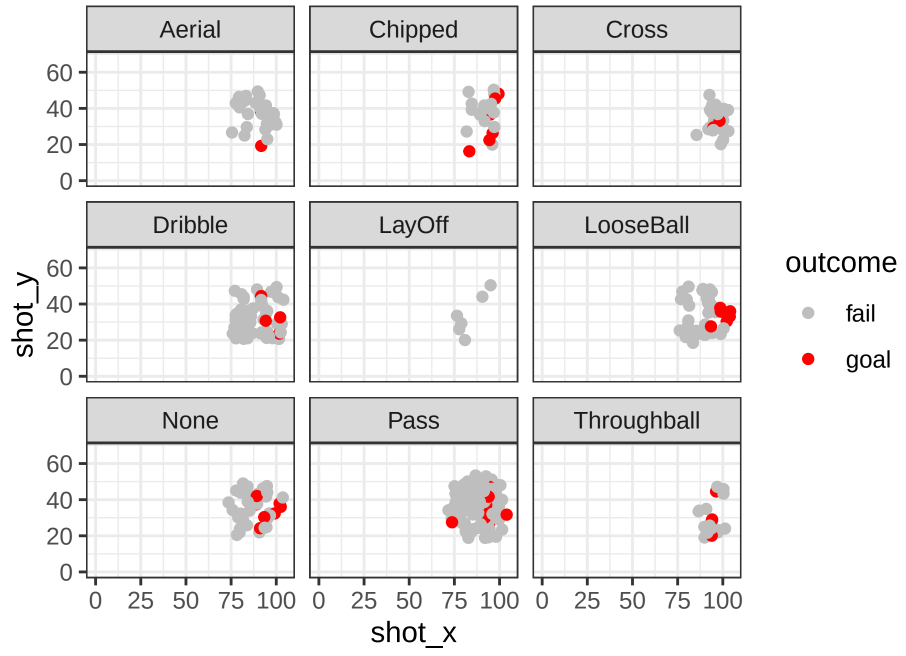
- `theme_bw()`: a variation on theme_grey() that uses a white background and thin grey grid lines.

- `theme_linedraw()`: a theme with only black lines of various widths on white backgrounds, reminiscent of a line drawing.
- `theme_light()` : similar to theme_linedraw() but with light grey lines and axes, to direct more attention towards the data.
- `theme_dark()`: the dark cousin of theme_light(), with similar line sizes but a dark background. Useful to make thin coloured lines pop out.
- `theme_minimal()`: a minimalistic theme with no background annotations.
- `theme_classic()`: a classic-looking theme, with x and y axis lines and no gridlines.
- `theme_void()`: a completely empty theme.

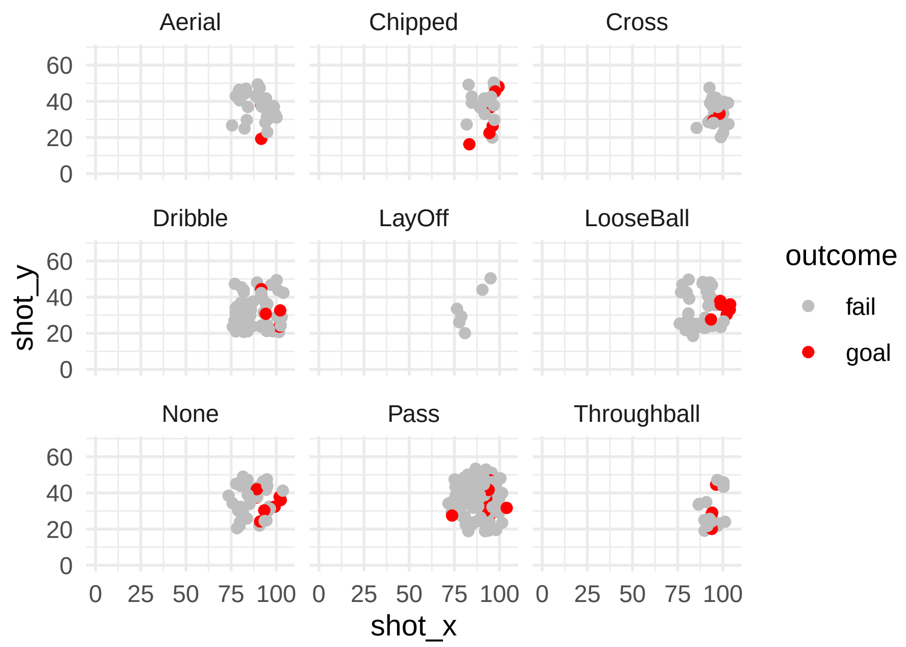
```
p <- fb.leicester.xg1516 %>%
  ggplot(aes(x=shot_x, y=shot_y)) +
  geom_point(aes(col=outcome)) +
  coord_fixed(xlim = c(0,105), ylim = c(0,68)) +
  scale_color_discrete(type = c("grey", "red")) +
  facet_wrap(~lastAction)

p + theme_dark()
```



```
p + theme_bw()
```



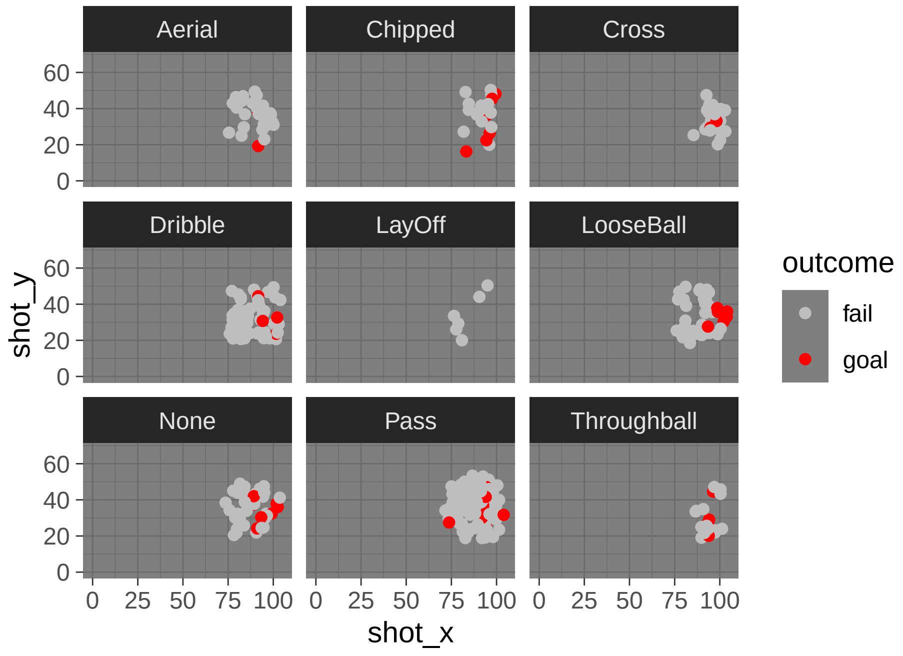
```r
p + theme_minimal()
```



Building the final plot:

- we first map the x- and y-coordinates of each shot to the position aesthetics
- we add a `geom_point()`-layer, where the outcome of shots is mapped to the color of the points
- we add a `scale_color_discrete()`-layer to override the default color scale and define the two colors for our two factor levels
- we add `coord_fixed()` for a cartesian coordinate system that has a fixed aspect ratio (so that 1m in pitch length corresponds to 1m in pitch width); we also define limits for the coordinate systems that are wider then the range of the data, in order to create a realistic display of a football pitch
- we add a `facet_wrap()` in order to separate plots by the last action performed before the shot
- lastly, we add a `theme_dark()` for a dark theming

```
fb.leicester.xg1516 %>%
  ggplot(aes(x=shot_x, y=shot_y)) +
  geom_point(aes(col=outcome)) +
  scale_color_discrete(type = c("grey", "red")) +
  coord_fixed(ratio = 1, xlim = c(0,105), ylim = c(0,68)) +
  facet_wrap(~lastAction) +
  theme_dark()
```



The beauty and flexibility in ggplot now comes from the fact that we could alter each layer in a way that will change the appearance of the plot, without having to rebuild the whole visualization. Try out to change the theme or the coloring of the plots, or create a different faceting for a different factor.

Exporting plots

Ways to export ggplot plots:

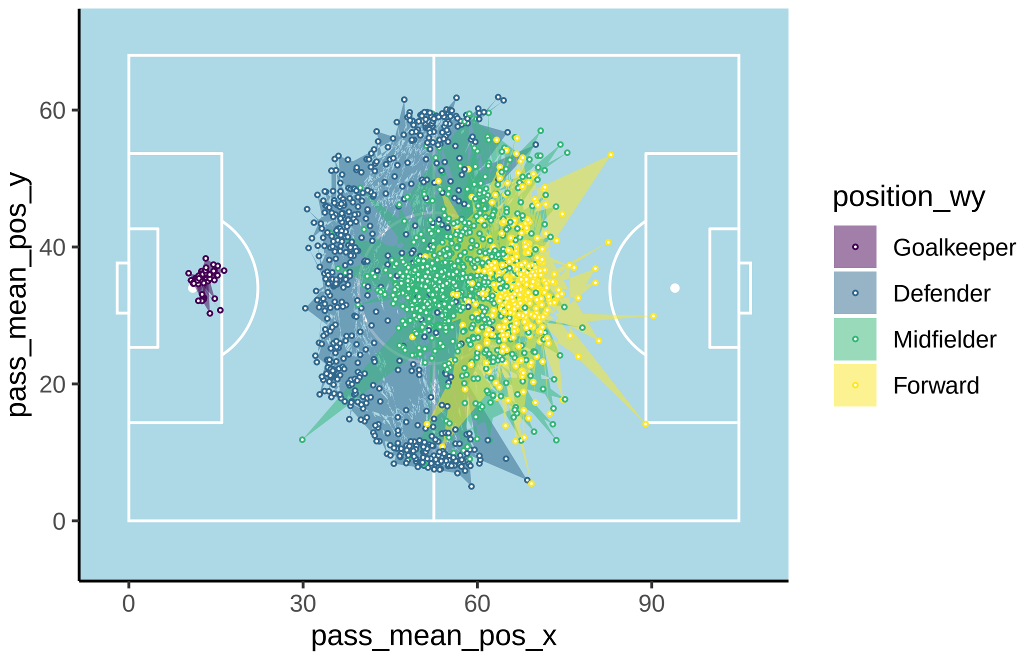
- either just run the code to display the plot, or save it by assigning it to a variable: `p <- ggplot(dat, aes(x,y)) + geom_point()`
- render it on screen with `p` or, inside loop/function, with `print(p)`
- render it to documents within notebooks using **Markdown/quarto**
- save to disk with `ggsave(p, "plot.png", width = 5, height = 5)`
- save a cached copy of the plot to disk: `saveRDS(p, "plot.rds")` and read it back in with `p <- readRDS("plot.rds")`

Backlog

36 Individual and collective geoms

- plot mean passing positions
- plot transparent polygon per player / per player position

```r
fb.europe.players1718 %>%
ggplot(aes(x=pass_mean_pos_x, y=pass_mean_pos_y)) +
annotate_pitch(dimensions = my_pitch_dimensions, colour = "white", fill = "lightblue") +
theme_classic() +
theme(panel.background = element_rect(fill = "lightblue")) +
geom_polygon(aes(group = position_wy, fill = position_wy), alpha = 0.5) +
geom_point(aes(col=position_wy), fill = "white", stroke = .5, shape = 21, size=.5) #+
```



```
facet_wrap(~position_wy)
```

```
<ggproto object: Class FacetWrap, Facet, gg>
    compute_layout: function
    draw_back: function
    draw_front: function
    draw_labels: function
    draw_panels: function
    finish_data: function
    init_scales: function
    map_data: function
    params: list
    setup_data: function
    setup_params: function
    shrink: TRUE
    train_scales: function
    vars: function
    super:  <ggproto object: Class FacetWrap, Facet, gg>
```

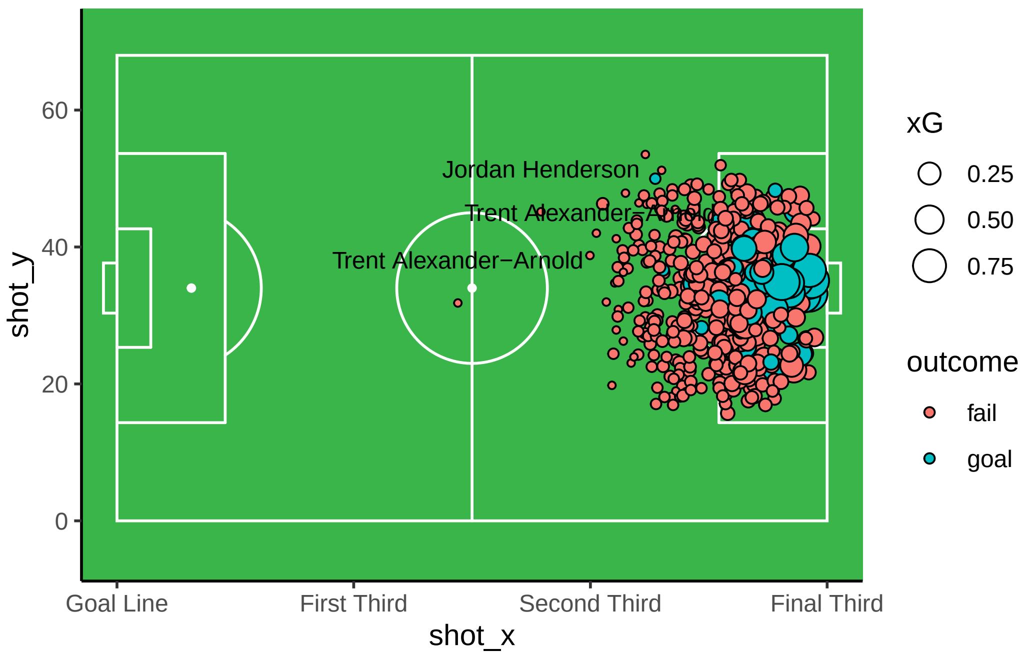
---

39 Building a shot map layer by layer

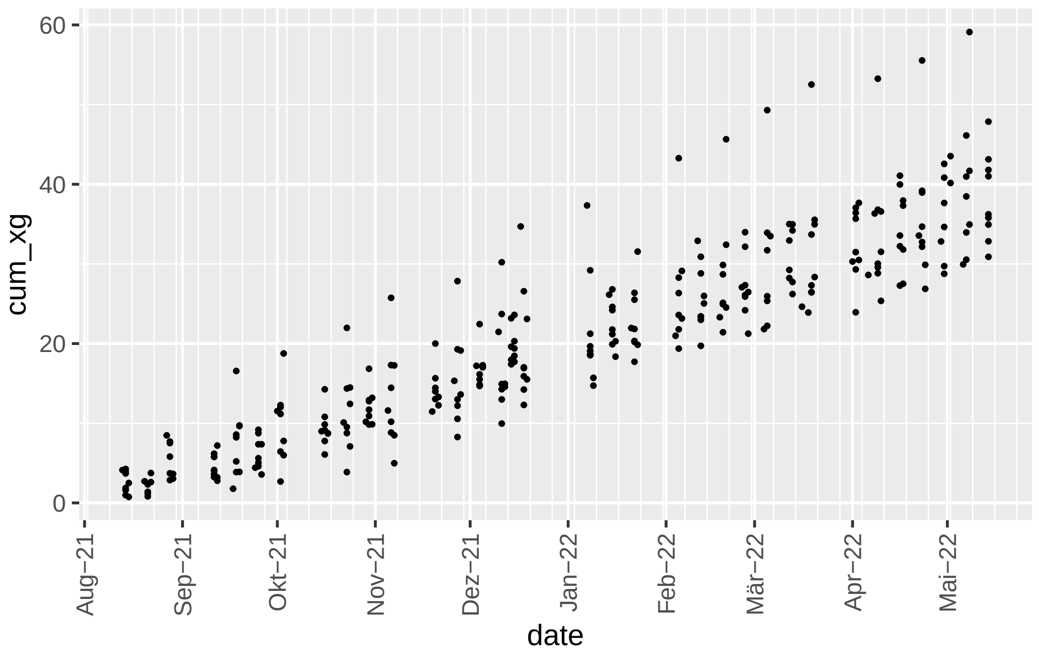
1) Coordinate system
2) pitch annotation
3) adding raw shot locations
    1) overplotting: reduce size, transparency, density estimation
4) adding outcome as color coding
5) adding xG value as size
6) maybe add shotType as polygon
7) annotate goals from high distance

40 Layer by layer: shot map

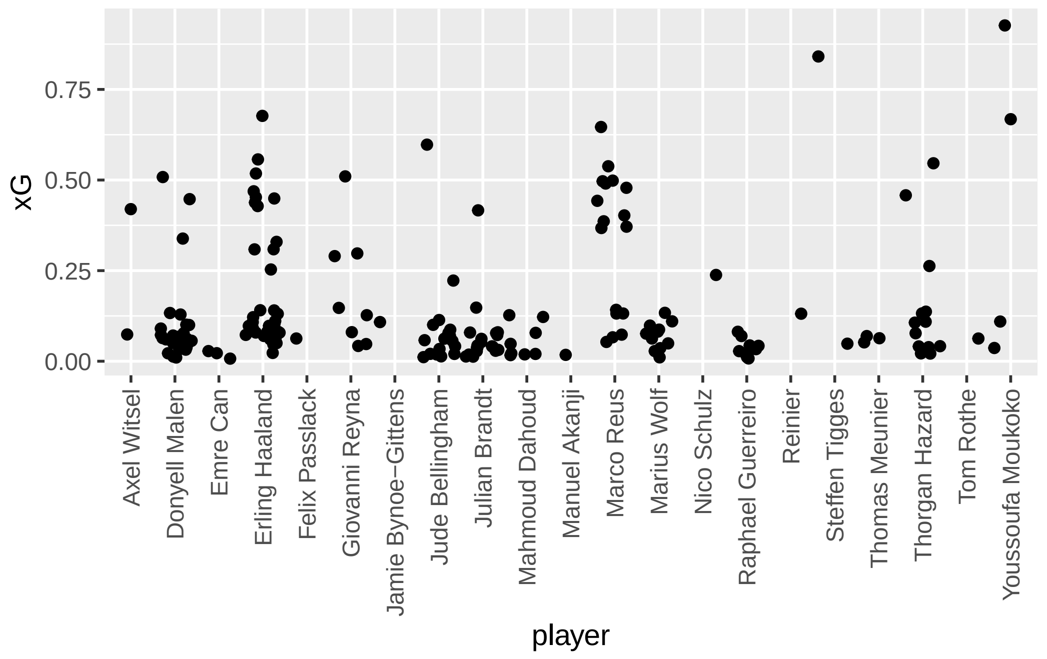
```
df_high <- fb.europe.xg1622 %>%
  filter(season == 2021, team == "Liverpool", shot_x < 70)

fb.europe.xg1622 %>%
  filter(season == 2021, team == "Liverpool") %>%
  ggplot(aes(x = shot_x, y = shot_y)) +
  annotate_pitch(dimensions = my_pitch_dimensions, colour = "white",fill = "#3ab54a") +
  coord_cartesian(xlim = c(0, 105)) +
  theme_classic() +
  theme(panel.background = element_rect(fill = "#3ab54a")) +
  geom_point(aes(size = xG, fill = outcome), col = 'black', shape = 21) +
  geom_text(data = df_high, aes(label = player), col = "black", size = 3, vjust = -2) +
  #geom_vline(xintercept = c(0, 35, 70, 105), col = "red", linetype = "dashed") +
  scale_x_continuous(breaks = seq(0,105, 35), labels = c("Goal Line", "First Third", "Second
```



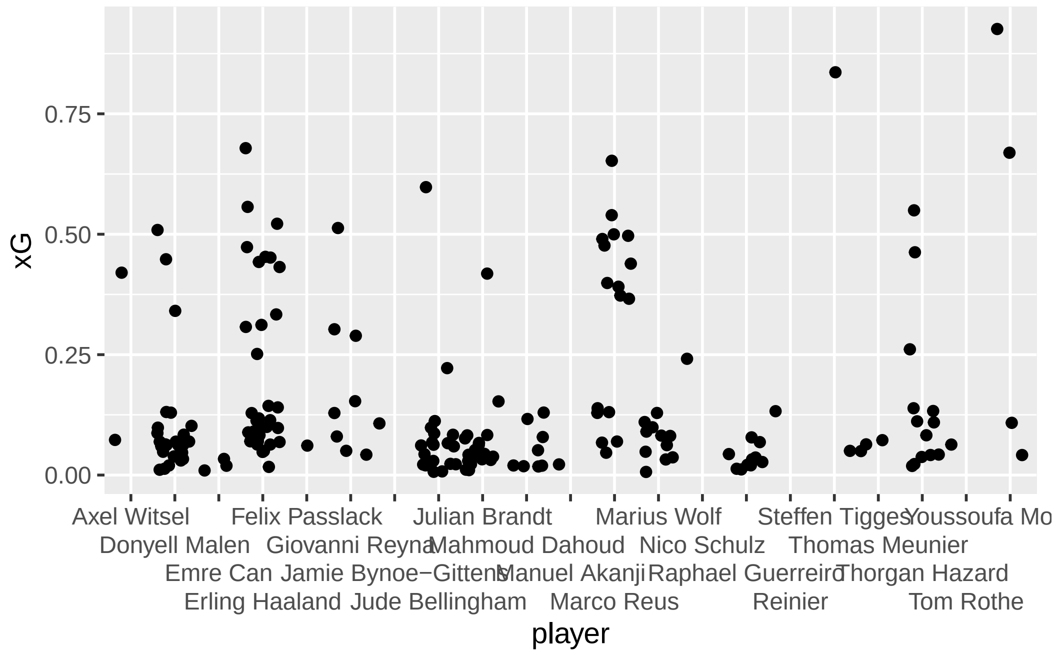
```
fb.europe.xg1622_evolution <- fb.europe.xg1622 %>%
  mutate( team = ifelse(h_a == "h", as.character(home_team), as.character(away_team)) ) %>%
  select(match_id, season, league, team, home_team, away_team, h_a, xG, date) %>%
  group_by(match_id) %>%
  summarise( league = first(league), team = first(team), date = first(date), season = first(
  ungroup() %>%
  group_by(season, team) %>%
  mutate( time = date, date = as.Date(date), cum_xg = cumsum(xG) )

fb.europe.xg1622_evolution %>%
  filter(season == 2021, league == "Bundesliga") %>%
  ggplot(aes(x=date, y = cum_xg)) +
  #geom_point() geom_line(aes(col = team)) +
  geom_point(size = .51) +
  scale_x_date(date_breaks = "1 month", date_minor_breaks = "1 week", date_labels = "%b-%y")
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
```



```
p_bvb <- fb.europe.xg1622 %>%
  filter(home_team == "Borussia Dortmund", h_a == "h", season == 2021) %>%
  ggplot(aes(x= player, y = xG)) +
  geom_jitter()

p_bvb +
  guides(x = guide_axis(angle = 90))
```



```
p_bvb +
  guides(x = guide_axis(n.dodge = 4))
```


xG
Axel Witsel
Donyell Malen
Emre Can
Erling Haaland
Felix Passlack
Giovanni Reyna
Jamie Bynoe−Gittens
Jude Bellingham
Julian Brandt
Mahmoud Dahoud
Manuel Akanji
Marco Reus
Marius Wolf
Nico Schulz
Raphael Guerreiro
Reinier
Steffen Tigges
Thomas Meunier
Thorgan Hazard
Tom Rothe
Youssoufa Mo
player