lab5

Irlanda Ayon-Moreno

4/7/2018

Lab 5: First contact with dplyr and ggplot2

Gaston Sanchez

> ## Learning Objectives:
>
> - Get started with `"dplyr"`
> - Get to know the basic dplyr verbs:
> - `slice()` , `filter()` , `select()`
> - `mutate()`
> - `arrange()`
> - `summarise()`
> - `group_by()`
> - Get started with `"ggplot2"`
> - Produce basic plots with `ggplot()`

Manipulating and Visualizing Data Frames

Last week you started to manipulate data tables (under the class of `"data.frame"` objects) using bracket notation, `dat[ , ]` , and the dollar operator, `dat$name` , in order to select specific rows, columns, or cells. In addition, you have been creating charts with functions like `plot()` , `boxplot()` , and `barplot()` , which are part of the `"graphics"` package.

In this lab, you will start learning about other approaches to manipulate tables and create statistical charts. We are going to use the functionality of the package `"dplyr"` to work with tabular data in a more consistent way. This is a fairly recent package introduced a couple of years ago, but it is based on more than a decade of research and work lead by Hadley Wickham.

Likewise, to create graphics in a more consistent and visually pleasing way, we are going to use the package `"ggplot2"` , also originally authored by Hadley Wickham, and developed as part of his PhD more than a decade ago.

Use the first hour of the lab to get as far as possible with the material associated to `"dplyr"` . Then use the second hour of the lab to work on graphics with `"ggplot2"` .

While you follow this lab, you may want to open these cheat sheets:

- dplyr cheatsheet (../cheatsheets/data-transformation-cheatsheet.pdf)
- ggplot2 cheatsheet (../cheatsheets/ggplot2-cheatsheet-2.1.pdf)

Filestructure and Shell Commands

We want you to keep practicing with the command line (e.g. Mac Terminal, Gitbash). Follow the steps listed below to create the necessary subdirectories like those depicted in this scheme:

```
    lab05/
      README.md
      data/
        nba2017-players.csv
      report/
        lab05.Rmd
        lab05.html
      images/
        ... # all the plot files
```

- Open a command line interface (e.g. Terminal or GitBash)
- Change your working directory to a location where you will store all the materials for this lab
- Use `mkdir` to create a directory `lab05` for the lab materials
- Use `cd` to change directory to (i.e. move inside) `lab05`
- Create other subdirectories: `data` , `report` , `images`
- Use `ls` to list the contents of `lab05` and confirm that you have all the subdirectories.
- Use `touch` to create an empty `README.md` text file
- Use a text editor (e.g. the one in RStudio) to open the `README.md` file, and then add a brief description of today's lab, using markdown syntax.
- Change directory to the `data/` folder.
- Download the data file with the command `curl` , and the `-O` option (letter O)

  ```
  curl -O https://raw.githubusercontent.com/ucb-stat133/stat133-spring-2018/master/data/nba2017-players.csv
  ```

- Use `ls` to confirm that the csv file is in `data/`
- Use *word count* `wc` to count the lines of the csv file
- Take a peek at the first rows of the csv file with `head`
- Take a peek at the last 5 rows of the csv file with `tail`

Installing packages

I'm assuming that you already installed the packages `"dplyr"` and `"ggplot2"` . If that's not the case then run on the console the command below (do NOT include this command in your `Rmd`):

```
# don't include this command in your Rmd file
# don't worry too much if you get a warning message
install.packages(c("dplyr", "ggplot2"))
```

Remember that you only need to install a package once! After a package has been installed in your machine, there is no need to call `install.packages()` again on the same package. What you should always invoke in order to use the functions in a package is the `library()` function:

```
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 3.4.2
```

```
##
## 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
```

```
library(ggplot2)
```

About loading packages: Another rule to keep in mind is to always load any required packages at the very top of your script files (`.R` or `.Rmd` or `.Rnw` files). Avoid calling the `library()` function in the middle of a script. Instead, load all the packages before anything else.

Path for Images

The other important specification to include in your Rmd file is a global chunk option to specify the location of plots and graphics. This is done by setting the `fig.path` argument inside the `knitr::opts_chunk$set()` function.

If you don't specify `fig.path` , `"knitr"` will create a default directory to store all the plots produced when knitting an Rmd file. This time, however, we want to have more control over where things are placed. Because you already have a folder `images/` as part of the filestructure, this is where we want `"knitr"` to save all the generated graphics.

NBA Players Data

The data file for this lab is the same you used last week: `nba2017-players.csv` .

To import the data in R you can use the base function `read.csv()` , or you can also use `read_csv()` from the package `"readr"` :

```
# with "base" read.csv()
dat <- read.csv('../data/nba2017-players.csv', stringsAsFactors = FALSE)

# with "readr" read_csv()
# dat <- read_csv('nba2017-players.csv')
```

Basic `"dplyr"` verbs

To make the learning process of `"dplyr"` gentler, Hadley Wickham proposes beginning with a set of five *basic verbs* or operations for data frames (each verb corresponds to a function in `"dplyr"`):

- **filter**: keep rows matching criteria
- **select**: pick columns by name
- **mutate**: add new variables
- **arrange**: reorder rows
- **summarise**: reduce variables to values

I've slightly modified Hadley's list of verbs:

- `filter()`, `slice()`, and `select()`: subsetting and selecting rows and columns
- `mutate()`: add new variables
- `arrange()`: reorder rows
- `summarise()`: reduce variables to values
- `group_by()`: grouped (aggregate) operations

Filtering, slicing, and selecting

`slice()` allows you to select rows by position:

```
# first three rows
three_rows <- slice(dat, 1:3)
three_rows
```

```
## # A tibble: 3 x 15
##   player   team  position height weight   age experience college    salary
##   <chr>    <chr> <chr>     <int>  <int> <int>      <int> <chr>       <dbl>
## 1 Al Horf… BOS   C            82    245    30          9 Universit… 2.65e7
## 2 Amir Jo… BOS   PF           81    240    29         11 ""         1.20e7
## 3 Avery B… BOS   SG           74    180    26          6 Universit… 8.27e6
## # ... with 6 more variables: games <int>, minutes <int>, points <int>,
## #   points3 <int>, points2 <int>, points1 <int>
```

`filter()` allows you to select rows by condition:

```
# subset rows given a condition
# (height greater than 85 inches)
gt_85 <- filter(dat, height > 85)
gt_85
```

```
##                  player team position height weight age experience
## 1        Edy Tavares  CLE        C     87    260  24          1
## 2   Boban Marjanovic  DET        C     87    290  28          1
## 3 Kristaps Porzingis  NYK       PF     87    240  21          1
## 4        Roy Hibbert  DEN        C     86    270  30          8
## 5      Alexis Ajinca  NOP        C     86    248  28          6
##                   college  salary games minutes points points3 points2
## 1                          5145     1      24      6       0       3
## 2                       7000000    35     293    191       0      72
## 3                       4317720    66    2164   1196     112     331
## 4 Georgetown University 5000000     6      11      4       0       2
## 5                       4600000    39     584    207       0      89
##   points1
## 1       0
## 2      47
## 3     198
## 4       0
## 5      29
```

`select()` allows you to select columns by name:

```
# columns by name
player_height <- select(dat, player, height)
```

Your turn:

- use `slice()` to subset the data by selecting the first 5 rows.

```
slice(dat, 1:5)
```

- use `slice()` to subset the data by selecting rows 10, 15, 20, …, 50.

```
slice(dat, seq(10, 50, 5))
```

- use `slice()` to subset the data by selecting the last 5 rows.

```
slice(dat, (nrow(dat)-4):nrow(dat))
```

- use `filter()` to subset those players with height less than 70 inches tall.

```
filter(dat, height < 70)
```

- use `filter()` to subset rows of Golden State Warriors (‘GSW’).

```
filter(dat, team == "GSW")
```

- use `filter()` to subset rows of GSW centers (‘C’).

```
filter(dat, team == "GSW" & position == "C")
```

- use `filter()` and then `select()` , to subset rows of lakers ('LAL'), and then display their names.

```
lakers <- filter(dat, team == 'LAL')
select(lakers, player)
```

- use `filter()` and then `select()` , to display the name and salary, of GSW point guards

```
gsw_pg <- filter(dat, team == 'GSW' & position == 'PG')
select(gsw_pg, player, salary)
```

- find how to select the name, age, and team, of players with more than 10 years of experience, making 10 million dollars or less.

```
select(filter(dat, experience > 10 & salary <= 10000000 ), player, age, team)
```

- find how to select the name, team, height, and weight, of rookie players, 20 years old, displaying only the first five occurrences (i.e. rows)

```
slice(select( filter(dat, experience == 0, age == 20), player, team, height, weight), 1:
5)
```

Adding new variables: `mutate()`

Another basic verb is `mutate()` which allows you to add new variables. Let's create a small data frame for the warriors with three columns: `player` , `height` , and `weight` :

```
# creating a small data frame step by step
gsw <- filter(dat, team == 'GSW')
gsw <- select(gsw, player, height, weight)
gsw <- slice(gsw, c(4, 8, 10, 14, 15))
gsw
```

Now, let's use `mutate()` to (temporarily) add a column with the ratio `height / weight` :

```
mutate(gsw, height / weight)
```

You can also give a new name, like: `ht_wt = height / weight` :

```
mutate(gsw, ht_wt = height / weight)
```

```
## # A tibble: 5 x 4
##   player          height weight ht_wt
##   <chr>            <int>  <int> <dbl>
## 1 Draymond Green      79    230 0.343
## 2 Kevin Durant        81    240 0.338
## 3 Klay Thompson       79    215 0.367
## 4 Stephen Curry       75    190 0.395
## 5 Zaza Pachulia       83    270 0.307
```

In order to permanently change the data, you need to assign the changes to an object:

```
gsw2 <- mutate(gsw, ht_m = height * 0.0254, wt_kg = weight * 0.4536)
gsw2
```

```
## # A tibble: 5 x 5
##   player         height weight  ht_m wt_kg
##   <chr>           <int>  <int> <dbl> <dbl>
## 1 Draymond Green     79    230  2.01 104
## 2 Kevin Durant       81    240  2.06 109
## 3 Klay Thompson      79    215  2.01  97.5
## 4 Stephen Curry      75    190  1.90  86.2
## 5 Zaza Pachulia      83    270  2.11 122
```

Reordering rows: `arrange()`

The next basic verb of `"dplyr"` is `arrange()` which allows you to reorder rows. For example, here's how to arrange the rows of `gsw` by `height`

```
# order rows by height (increasingly)
arrange(gsw, height)
```

```
## # A tibble: 5 x 3
##   player         height weight
##   <chr>           <int>  <int>
## 1 Stephen Curry      75    190
## 2 Draymond Green     79    230
## 3 Klay Thompson      79    215
## 4 Kevin Durant       81    240
## 5 Zaza Pachulia      83    270
```

By default `arrange()` sorts rows in increasing order. To arrange rows in descending order you need to use the auxiliary function `desc()` .

```
# order rows by height (decreasingly)
arrange(gsw, desc(height))
```

```
## # A tibble: 5 x 3
##   player         height weight
##   <chr>           <int>  <int>
## 1 Zaza Pachulia      83    270
## 2 Kevin Durant       81    240
## 3 Draymond Green     79    230
## 4 Klay Thompson      79    215
## 5 Stephen Curry      75    190
```

```
# order rows by height, and then weight
arrange(gsw, height, weight)
```

```
## # A tibble: 5 x 3
##   player         height weight
##   <chr>           <int>  <int>
## 1 Stephen Curry      75    190
## 2 Klay Thompson      79    215
## 3 Draymond Green     79    230
## 4 Kevin Durant       81    240
## 5 Zaza Pachulia      83    270
```

Your Turn

- using the data frame `gsw` , add a new variable `product` with the product of `height` and `weight` .

```
mutate(gsw, product = height * weight)
```

- create a new data frame `gsw3` , by adding columns `log_height` and `log_weight` with the log transformations of `height` and `weight` .

```
gsw3 <- mutate(gsw, log_height = log(height), log_weight = log(weight))
gsw3
```

- use the original data frame to `filter()` and `arrange()` those players with height less than 71 inches tall, in increasing order.

```
arrange(filter(dat, height < 71), height)
```

- display the name, team, and salary, of the top-5 highest paid players

```
slice( select( arrange(dat, desc(salary)), player, team, salary), 1:5)
```

- display the name, team, and points3, of the top 10 three-point players

```
slice( arrange( select(dat, player, team, points3), desc(points3)), 1:10)
```

- create a data frame `gsw_mpg` of GSW players, that contains variables for player name, experience, and `min_per_game` (minutes per game), sorted by `min_per_game` (in descending order)

```
arrange( select( mutate( filter(dat, team == "GSW"), min_per_game = minutes / games), pl
ayer, experience, min_per_game), desc(min_per_game))
```

Summarizing values with `summarise()`

The next verb is `summarise()` . Conceptually, this involves applying a function on one or more columns, in order to summarize values. This is probably easier to understand with one example.

Say you are interested in calculating the average salary of all NBA players. To do this “a la dplyr” you use `summarise()` , or its synonym function `summarize()` :

```
# average salary of NBA players
summarise(dat, avg_salary = mean(salary))
```

```
##   avg_salary
## 1    6187014
```

Calculating an average like this seems a bit *verbose*, especially when you can directly use `mean()` like this:

```
mean(dat$salary)
```

```
## [1] 6187014
```

So let's make things a bit more interessting. What if you want to calculate some summary statistics for `salary` : min, median, mean, and max?

```
# some stats for salary (dplyr)
summarise(
  dat,
  min = min(salary),
  median = median(salary),
  avg = mean(salary),
  max = max(salary)
)
```

```
##    min  median     avg      max
## 1 5145 3500000 6187014 30963450
```

Well, this may still look like not much. You can do the same in base R (there are actually better ways to do this):

```
# some stats for salary (base R)
c(min = min(dat$salary),
  median = median(dat$salary),
  median = mean(dat$salary),
  max = max(dat$salary))
```

```
##      min   median   median      max
##     5145  3500000  6187014 30963450
```

Grouped operations

To actually appreciate the power of `summarise()` , we need to introduce the other major basic verb in `"dplyr"` : `group_by()` . This is the function that allows you to perform data aggregations, or *grouped operations*.

Let's see the combination of `summarise()` and `group_by()` to calculate the average salary by team:

```
group_by(dat, team)
```

```
## # A tibble: 441 x 15
## # Groups:   team [30]
##               player  team position height weight   age experience
##                <chr> <chr>    <chr>  <int>  <int> <int>      <int>
##  1        Al Horford   BOS        C     82    245    30          9
##  2      Amir Johnson   BOS       PF     81    240    29         11
##  3     Avery Bradley   BOS       SG     74    180    26          6
##  4 Demetrius Jackson   BOS       PG     73    201    22          0
##  5      Gerald Green   BOS       SF     79    205    31          9
##  6     Isaiah Thomas   BOS       PG     69    185    27          5
##  7       Jae Crowder   BOS       SF     78    235    26          4
##  8       James Young   BOS       SG     78    215    21          2
##  9      Jaylen Brown   BOS       SF     79    225    20          0
## 10     Jonas Jerebko   BOS       PF     82    231    29          6
## # ... with 431 more rows, and 8 more variables: college <chr>,
## #   salary <dbl>, games <int>, minutes <int>, points <int>, points3 <int>,
## #   points2 <int>, points1 <int>
```

```
# average salary, grouped by team
summarise(
  group_by(dat, team),
  avg_salary = mean(salary)
)
```

```
## # A tibble: 30 x 2
##    team  avg_salary
##    <chr>      <dbl>
##  1 ATL      6491892
##  2 BOS      6127673
##  3 BRK      4363414
##  4 CHI      6138459
##  5 CHO      6683086
##  6 CLE      8386014
##  7 DAL      6139880
##  8 DEN      5225533
##  9 DET      6871594
## 10 GSW      6579394
## # ... with 20 more rows
```

Here’s a similar example with the average salary by position:

```
# average salary, grouped by position
summarise(
  group_by(dat, position),
  avg_salary = mean(salary)
)
```

```
## # A tibble: 5 x 2
##   position avg_salary
##   <chr>         <dbl>
## 1 C           6987682
## 2 PF          5890363
## 3 PG          6069029
## 4 SF          6513374
## 5 SG          5535260
```

Here’s a more fancy example: average weight and height, by position, displayed in desceding order by average height:

```
arrange(
  summarise(
    group_by(dat, position),
    avg_height = mean(height),
    avg_weight = mean(weight)),
  desc(avg_height)
)
```

```
## # A tibble: 5 x 3
##   position avg_height avg_weight
##   <chr>         <dbl>      <dbl>
## 1 C              83.3        251
## 2 PF             81.5        236
## 3 SF             79.6        220
## 4 SG             77.0        205
## 5 PG             74.3        189
```

Your turn:

- use `summarise()` to get the largest height value.

```
summarise(dat, max_height = max(height))
```

- use `summarise()` to get the standard deviation of `points3`.

```
summarise(dat, sd(points3))
```

- use `summarise()` and `group_by()` to display the median of three-points, by team.

```
summarise( group_by(dat, team), median3 = median(points3))
```

- display the average triple points by team, in ascending order, of the bottom-5 teams (worst 3pointer teams)

```
avg3t <- arrange( summarise( group_by(dat, team), avg3 = mean(points3)), avg3)
slice(avg3t, 1:5)
```

- obtain the mean and standard deviation of `age`, for Power Forwards, with 5 and 10 years (including) years of experience.

```
summarise( filter(dat, position == "PF" & experience >=5 & experience <= 10), avg_age =
mean(age), sd_age = sd(age))
```

First contact with `ggplot()`

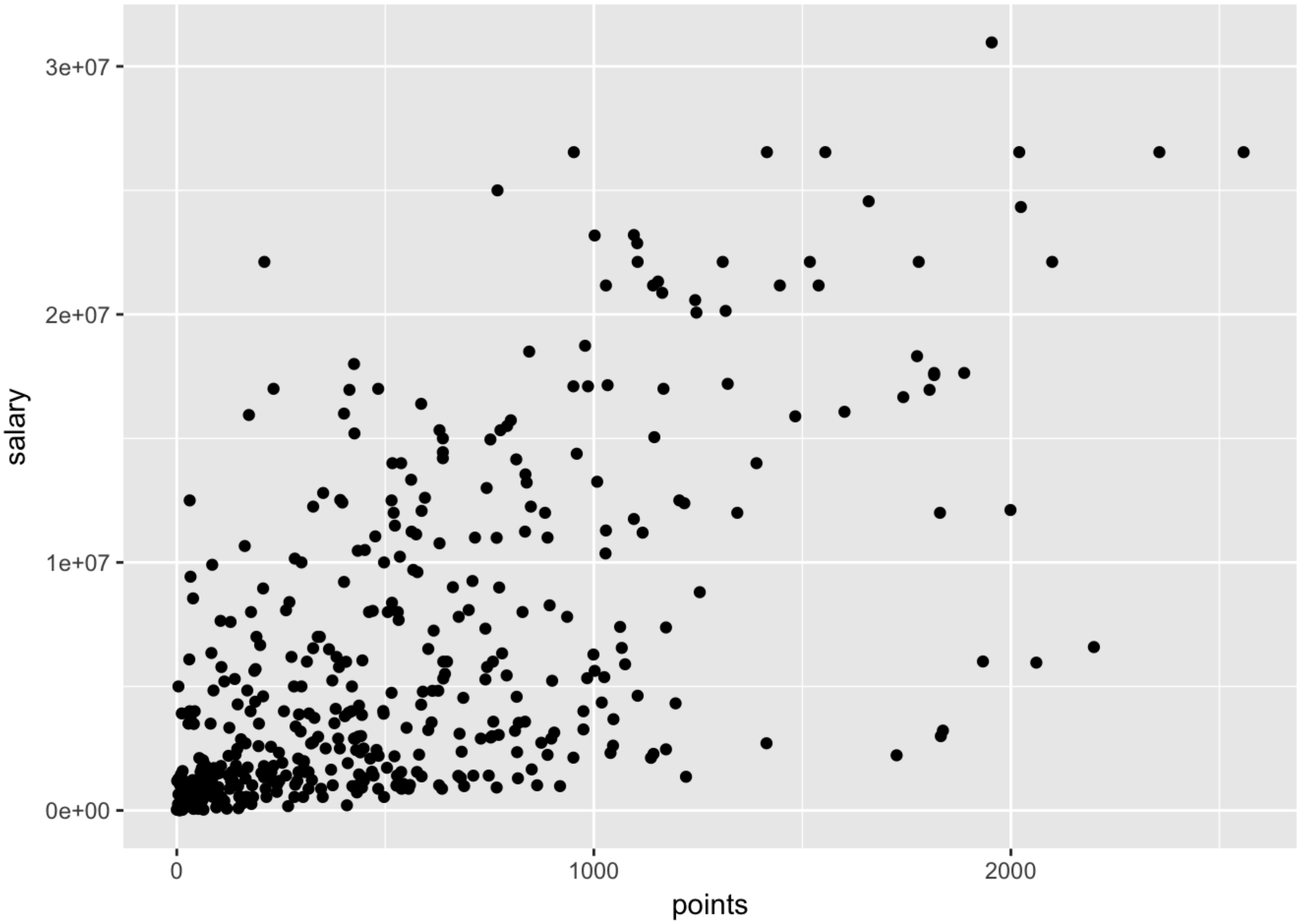
The package `"ggplot2"` is probably the most popular package in R to create *beautiful* static graphics. Comapred to the functions in the base package `"graphcics"`, the package `"ggplot2` " follows a somewhat different philosophy, and it tries to be more consistent and modular as possible.

- The main function in `"ggplot2"` is `ggplot()`
- The main input to `ggplot()` is a data frame object.
- You can use the internal function `aes()` to specify what columns of the data frame will be used for the graphical elements of the plot.
- You must specify what kind of *geometric objects* or **geoms** will be displayed: e.g. `geom_point()`, `geom_bar()`, `geom_boxpot()`.
- Pretty much anything else that you want to add to your plot is controlled by auxiliary functions, especially those things that have to do with the format, rather than the underlying data.
- The construction of a ggplot is done by *adding layers* with the `+` operator.

Scatterplots

Let's start with a scatterplot of `salary` and `points`

```
# scatterplot (option 1)
ggplot(data = dat) +
  geom_point(aes(x = points, y = salary))
```



- `ggplot()` creates an object of class `"ggplot"`
- the main input for `ggplot()` is `data` which must be a data frame
- then we use the `"+"` operator to add a layer
- the geometric object (geom) are points: `geom_points()`
- `aes()` is used to specify the `x` and `y` coordinates, by taking columns `points` and `salary` from the data frame

The same scatterplot can also be created with this alternative, and more common use of `ggplot()`

```
# scatterplot (option 2)
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point()
```

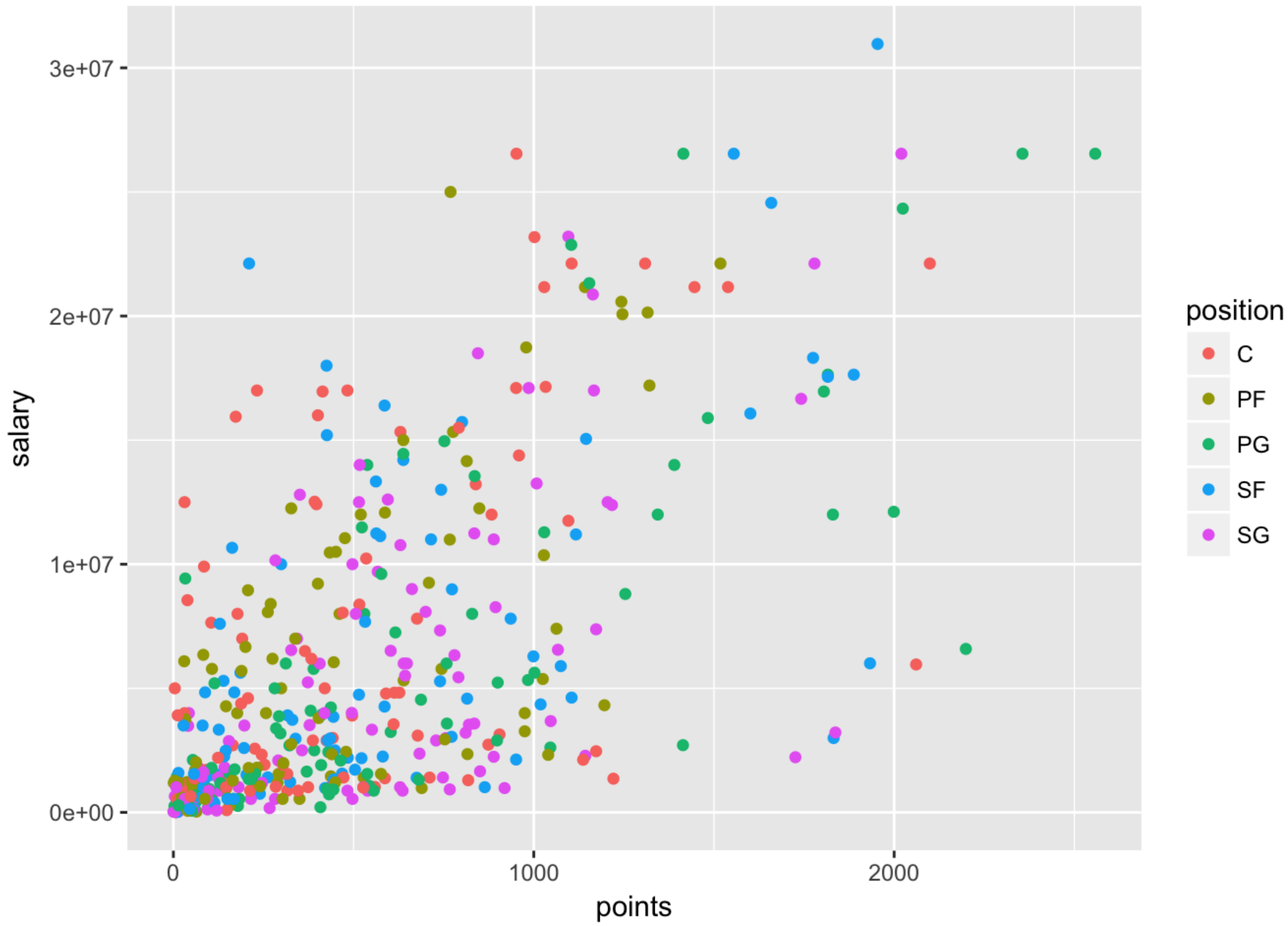
Label your chunks!

When including code for plots and graphics, we strongly recommend that you create an individual code chunk for each plot, and that you **give a label** to that chunk. When `"knitr"` creates the file of the plot, it will use the chunk label for the graph. So it’s better to give meaningful names to those chunks containing graphics.

Adding color

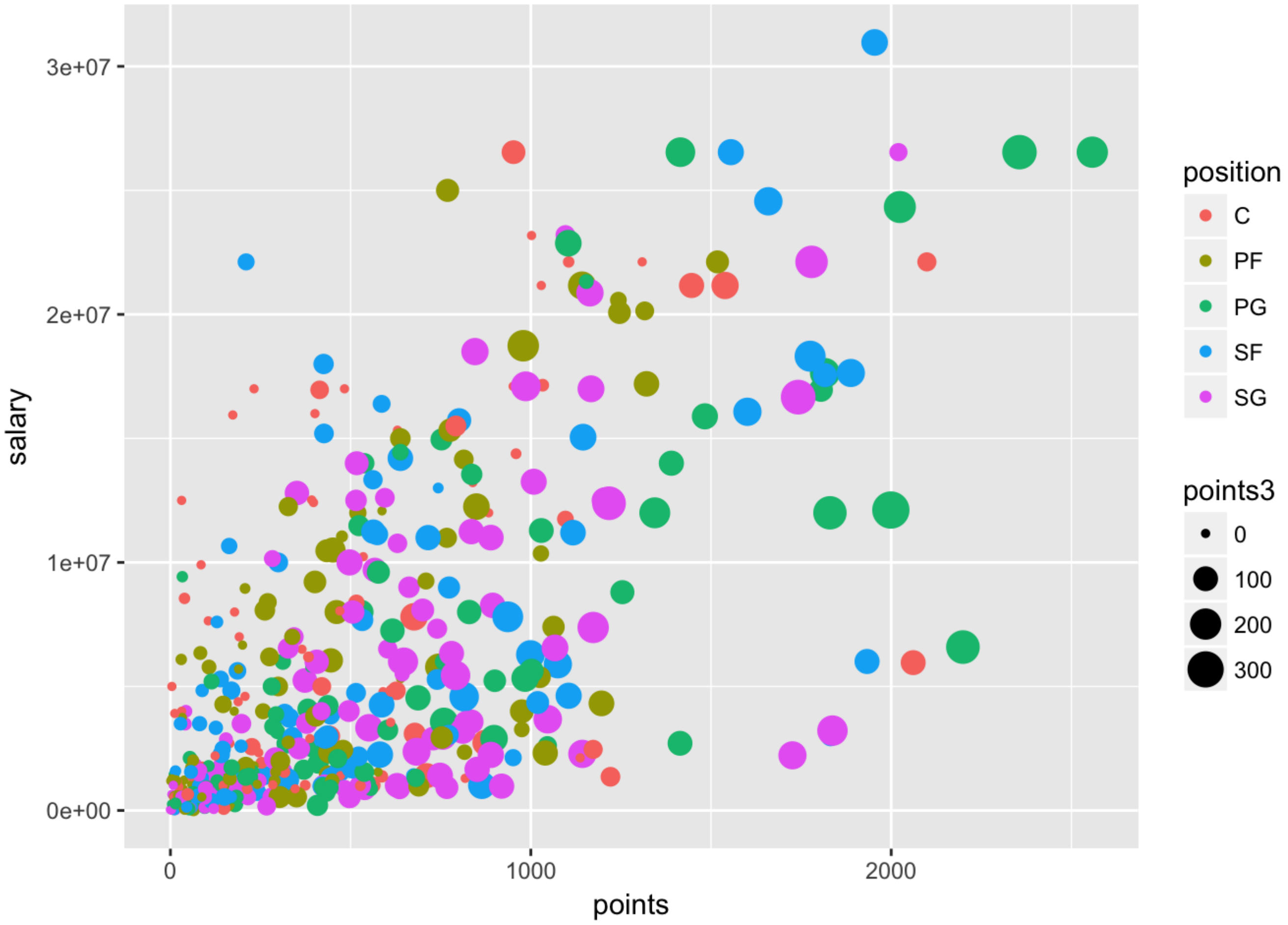
Say you want to color code the points in terms of `position`

```
# colored scatterplot
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point(aes(color = position))
```



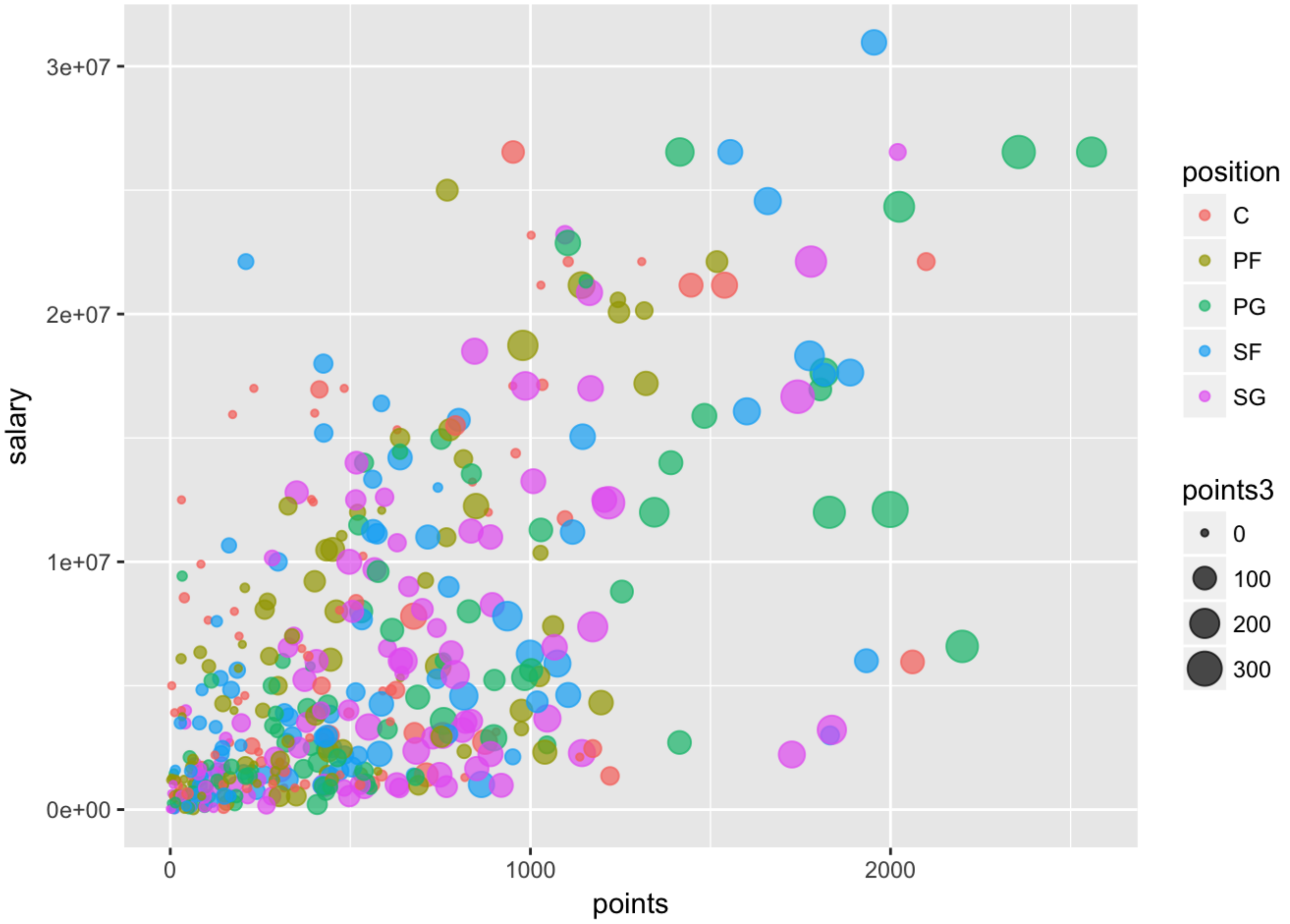
Maybe you want to modify the size of the dots in terms of `points3` :

```
# sized and colored scatterplot
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point(aes(color = position, size = points3))
```



To add some transparency effect to the dots, you can use the `alpha` parameter.

```
# sized and colored scatterplot
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point(aes(color = position, size = points3), alpha = 0.7)
```

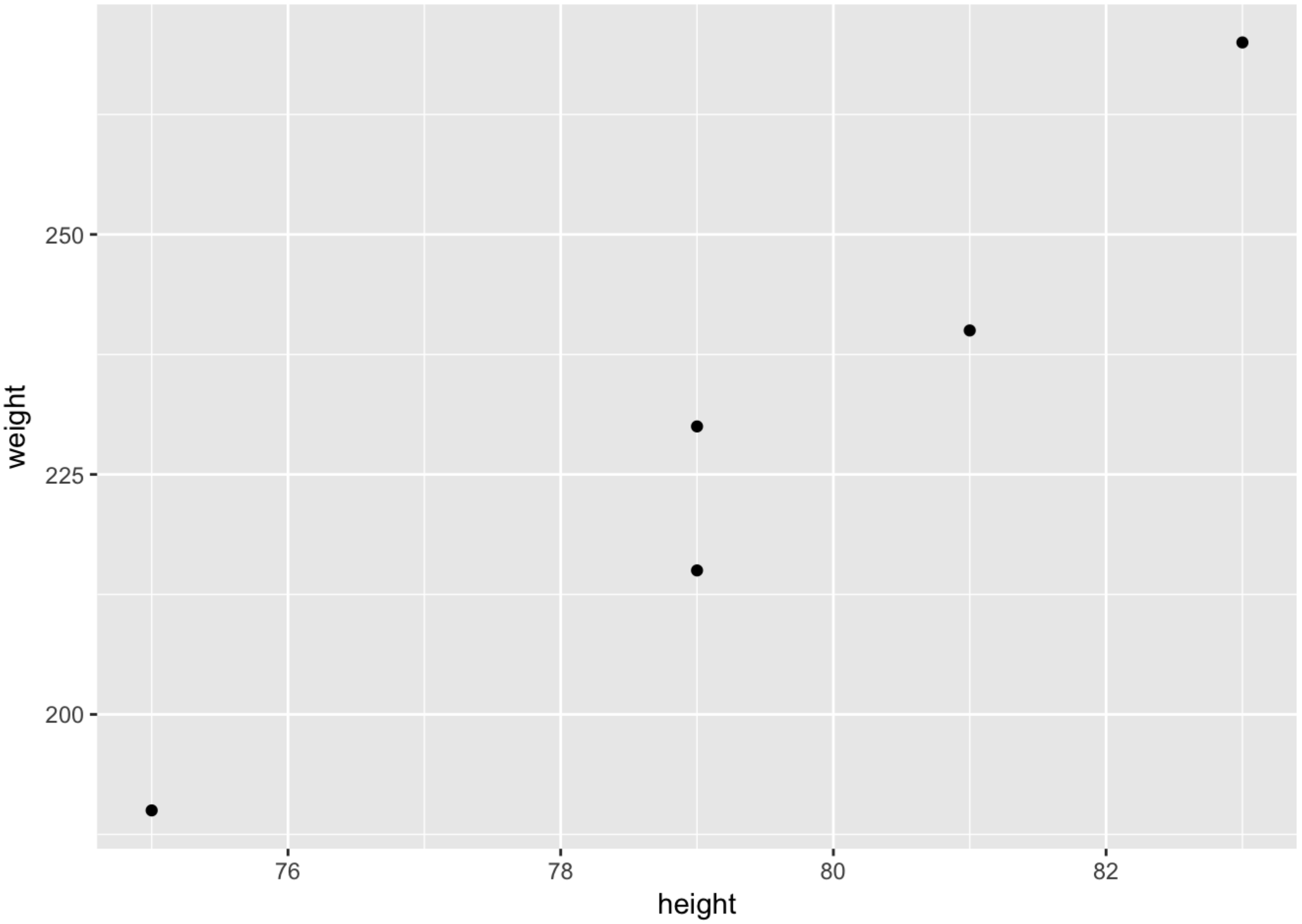


Notice that `alpha` was specified outside `aes()`. This is because we are not using any column for the `alpha` transparency values.

Your turn:

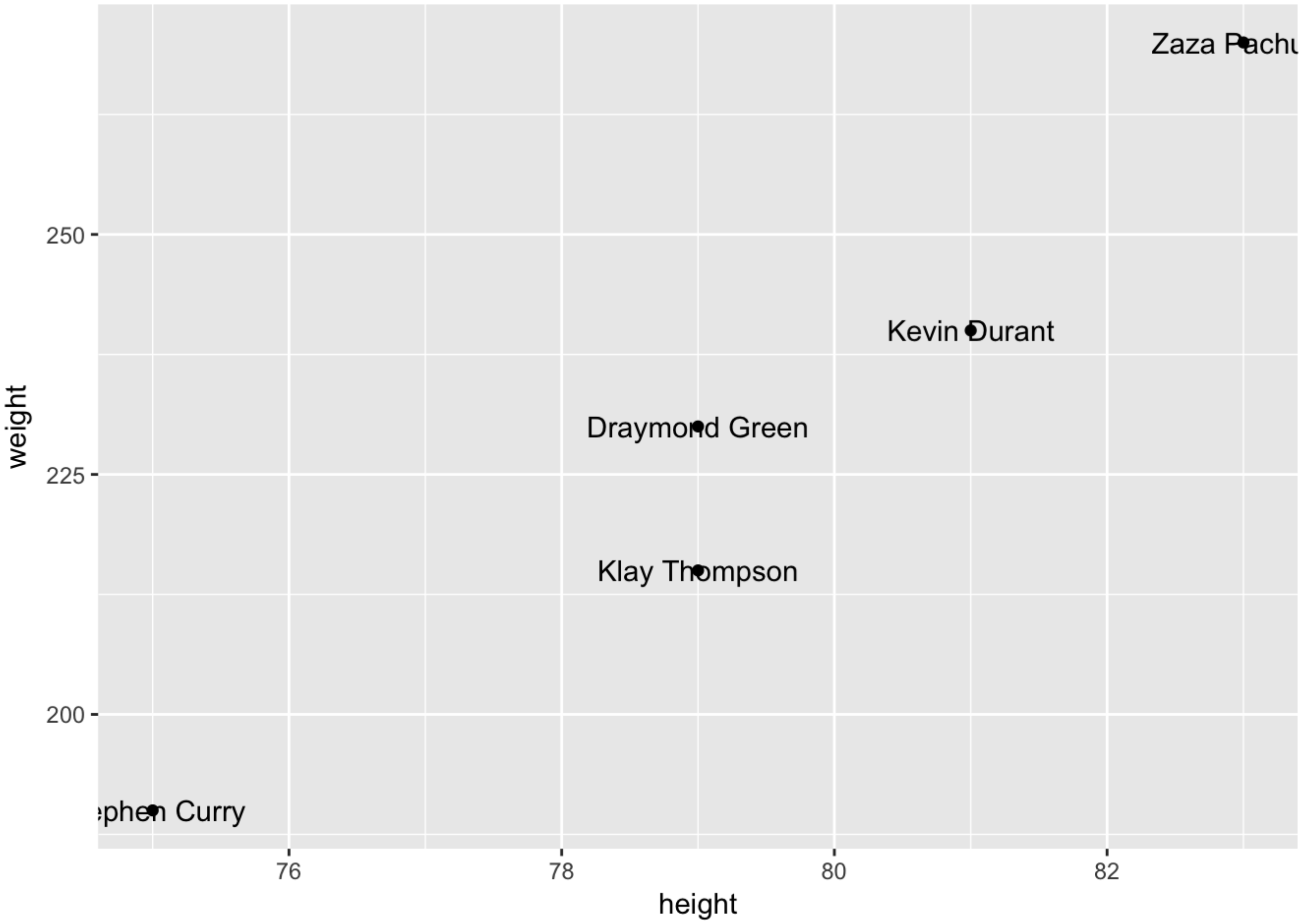
- Open the ggplot2 cheatsheet (../cheatsheets/ggplot2-cheatsheet-2.1.pdf)
- Use the data frame `gsw` to make a scatterplot of `height` and `weight`.

```
ggplot(data = gsw, aes(x = height, y = weight)) +
  geom_point()
```



- Find out how to make another scatterplot of `height` and `weight`, using `geom_text()` to display the names of the players.

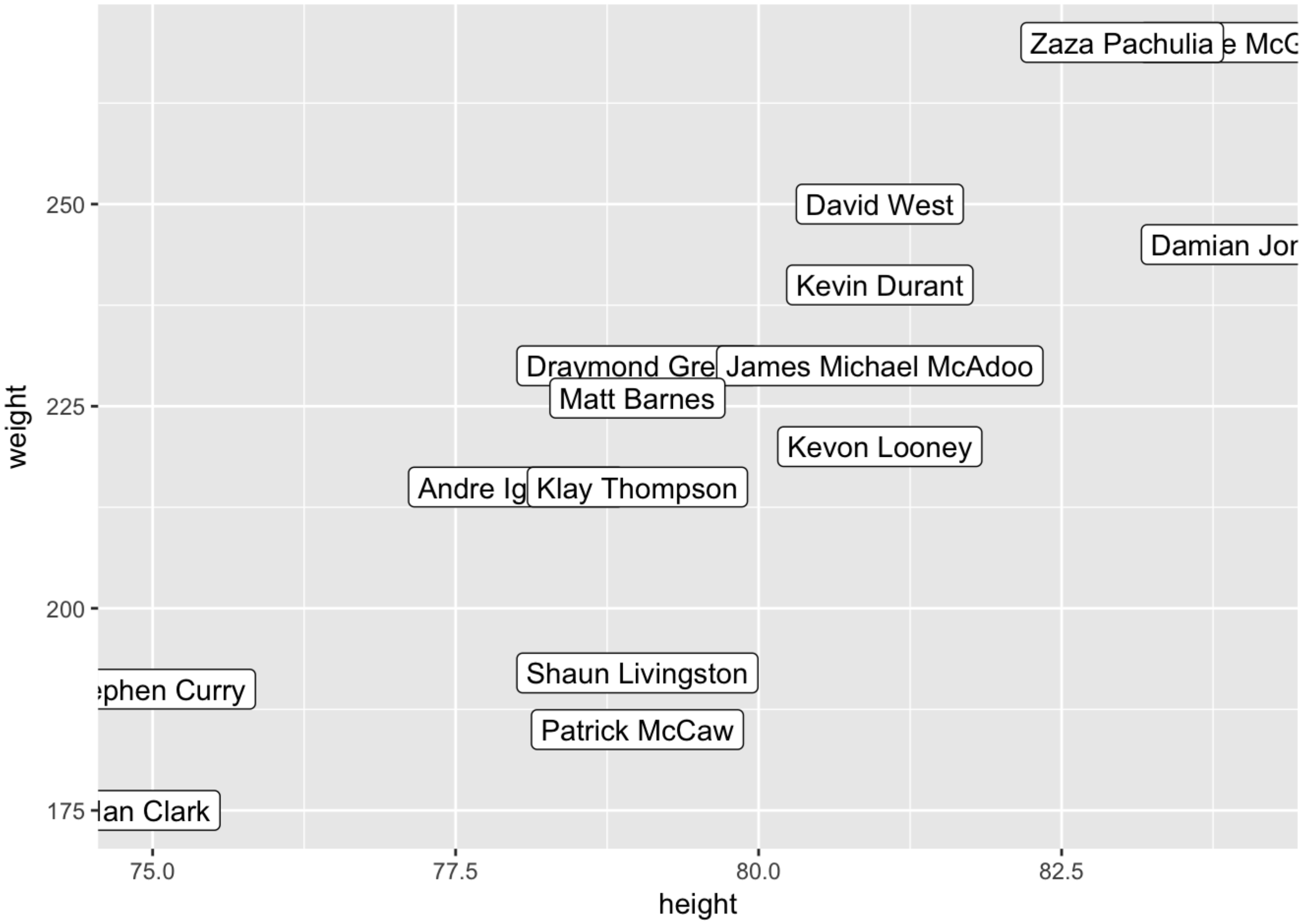
```
ggplot(data = gsw, aes(x = height, y = weight)) +
  geom_point() +
  geom_text(aes(label = player))
```



- Get a scatter plot of `height` and `weight` , for ALL the warriors, displaying their names with `geom_label()` .

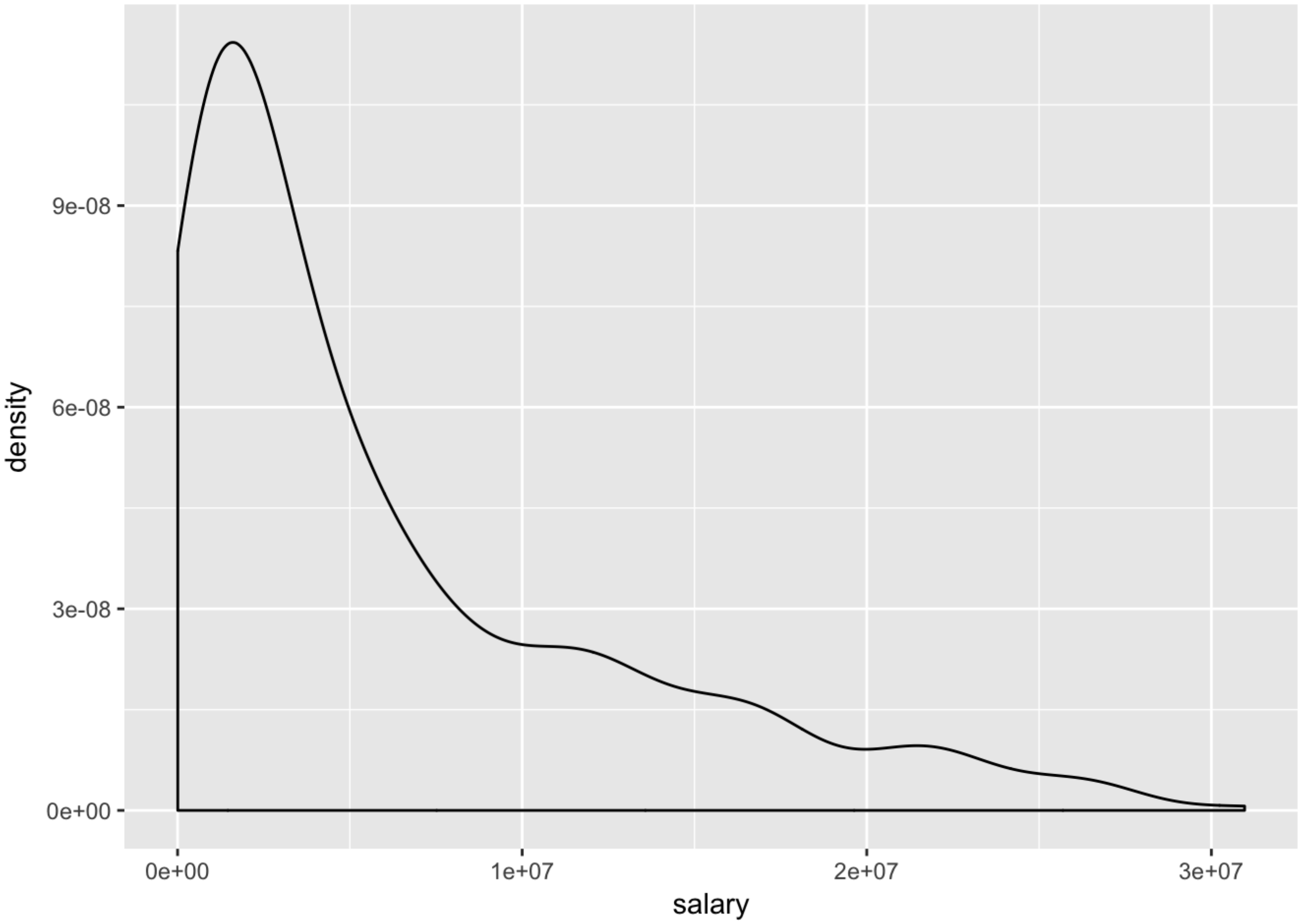
```
GSW <- filter(dat, team == "GSW")
```

```
ggplot(data = GSW, aes(x = height, y = weight)) +
  geom_label(aes(label = player))
```



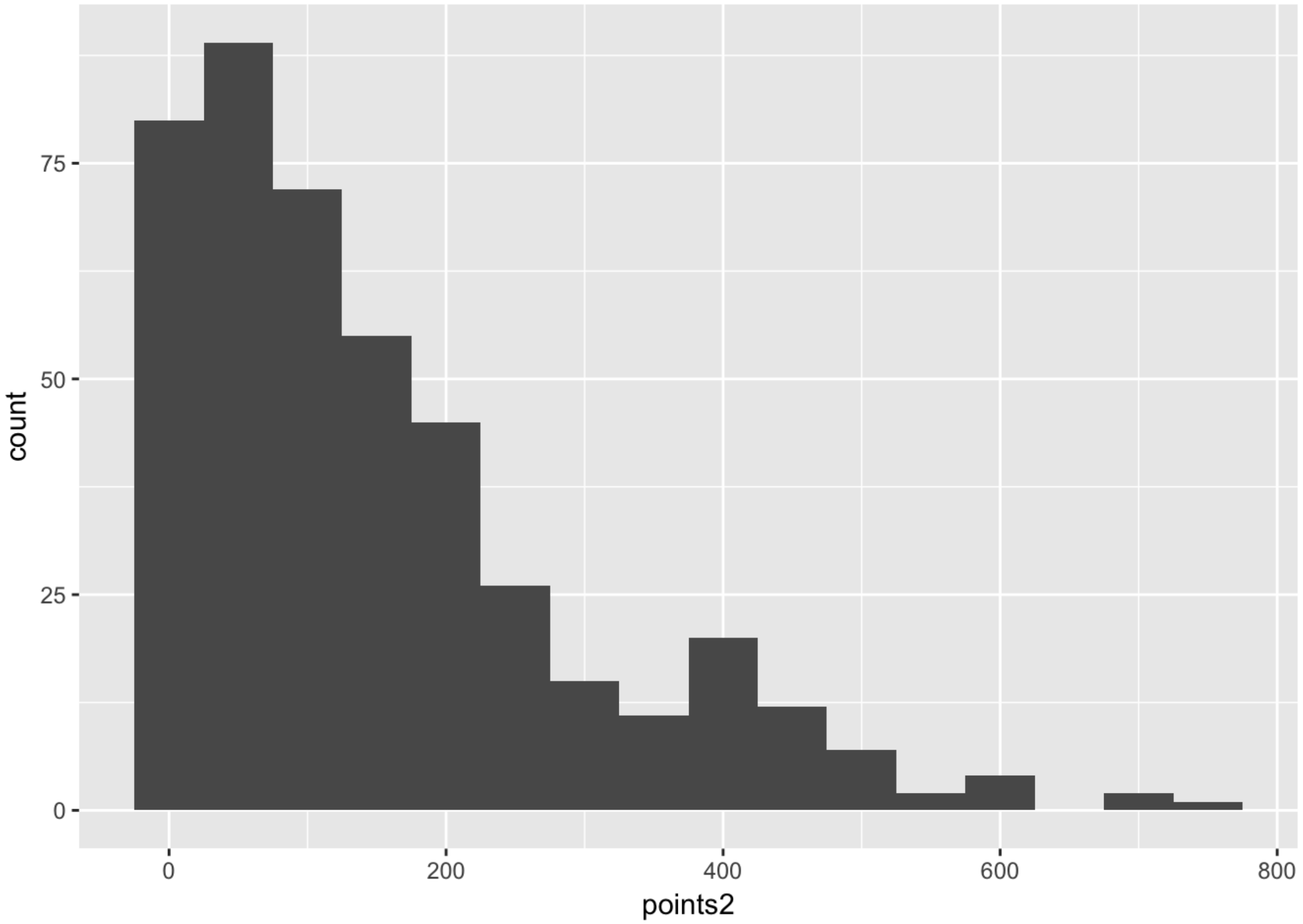
- Get a density plot of `salary` (for all NBA players).

```
ggplot(data = dat, aes(x = salary)) +
  geom_density()
```



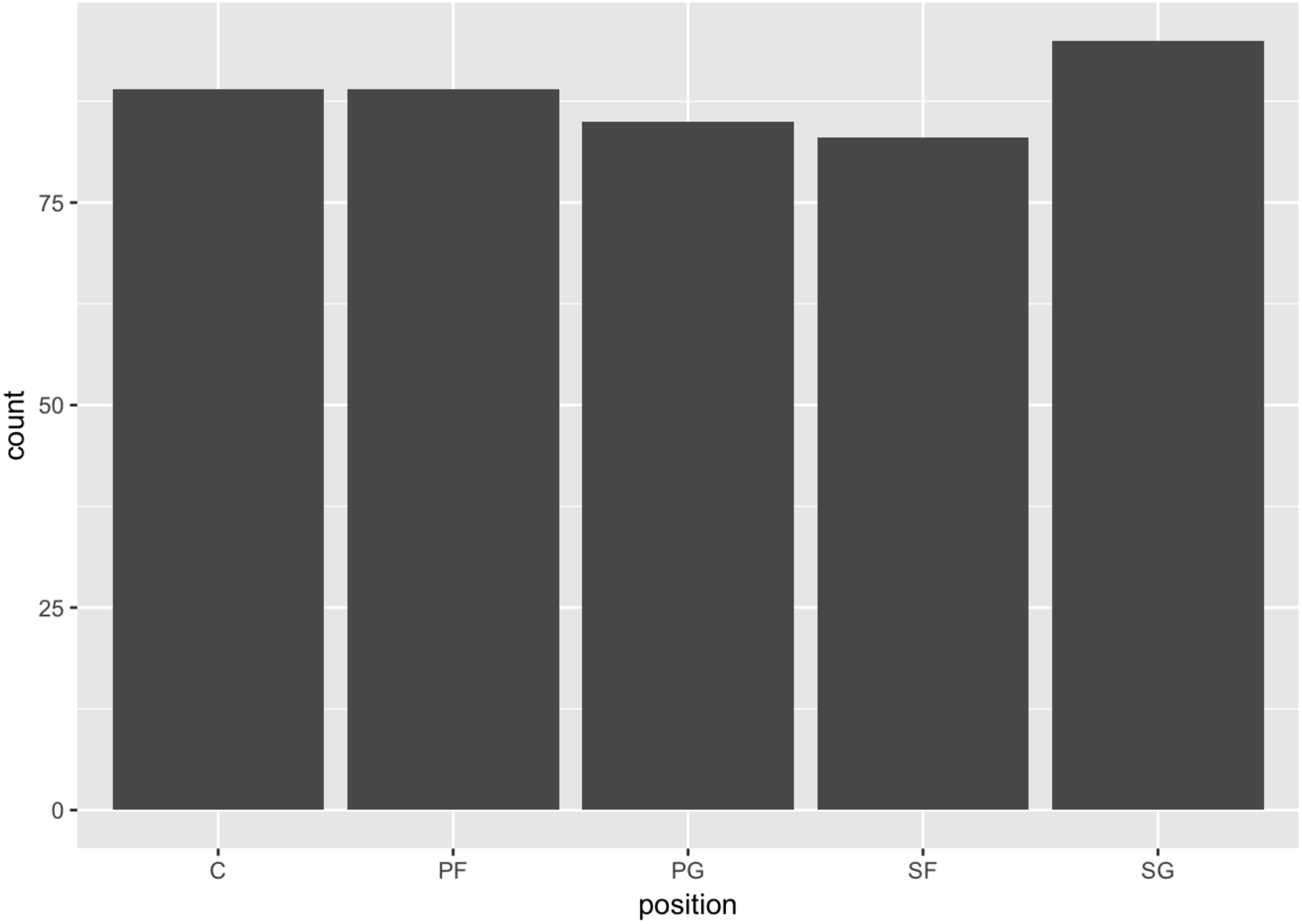
- Get a histogram of `points2` with binwidth of 50 (for all NBA players).

```
ggplot(data = dat) +
  geom_histogram(aes(x = points2), binwidth = 50)
```



- Get a barchart of the `position` frequencies (for all NBA players).

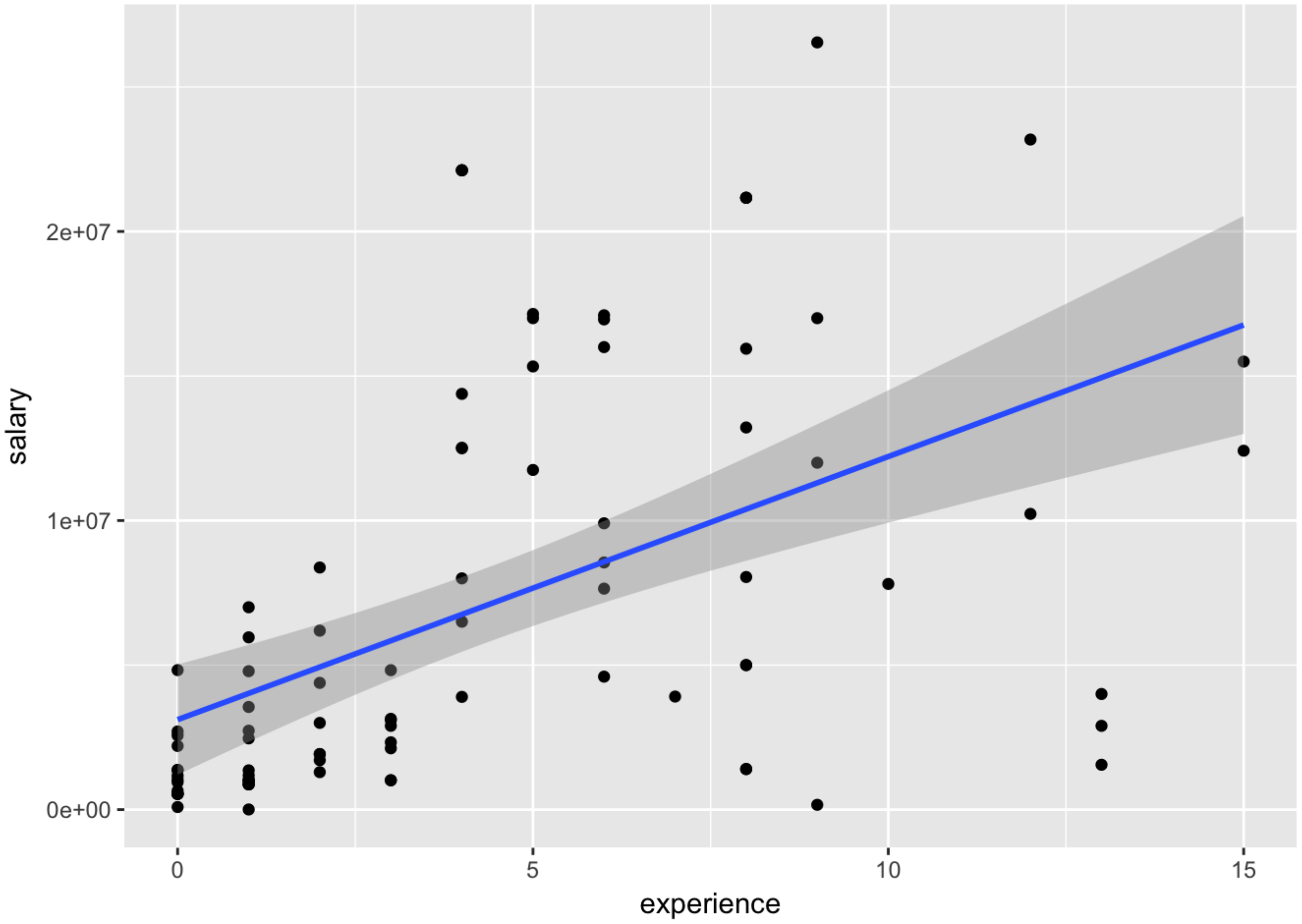
```
ggplot(data = dat) +
  geom_bar(aes(x = position))
```



- Make a scatterplot of `experience` and `salary` of all Centers, and use `geom_smooth()` to add a regression line.

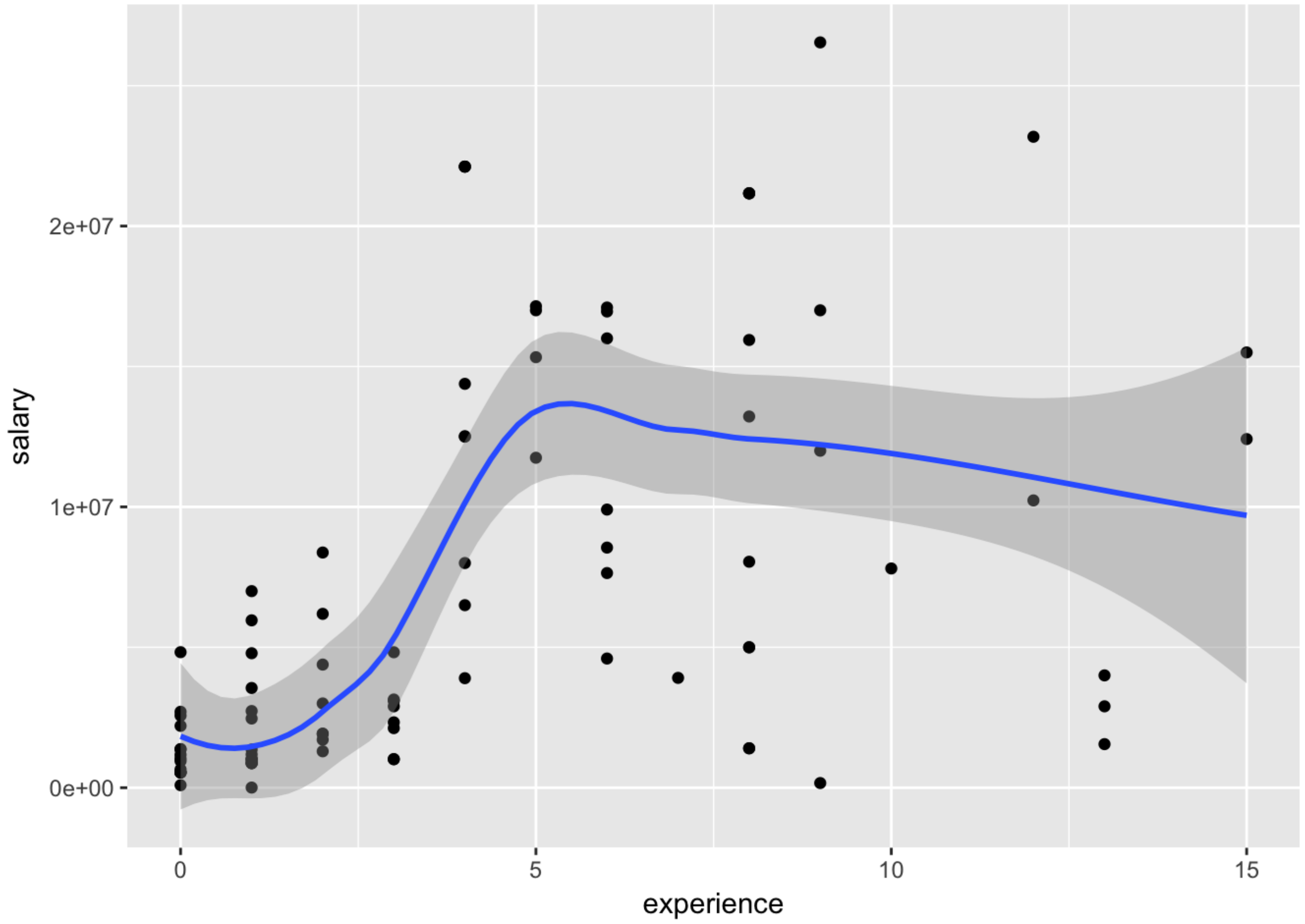
```
centers <- filter(dat, position == "C")
```

```
ggplot(data = centers, aes(x = experience, y = salary)) +
  geom_point() +
  geom_smooth(method = lm)
```



- Repeat the same scatterplot of `experience` and `salary` of all Centers, but now use `geom_smooth()` to add a loess line (i.e. smooth line).

```
ggplot(data = centers, aes(x = experience, y = salary)) +
  geom_point() +
  geom_smooth(method = loess)
```

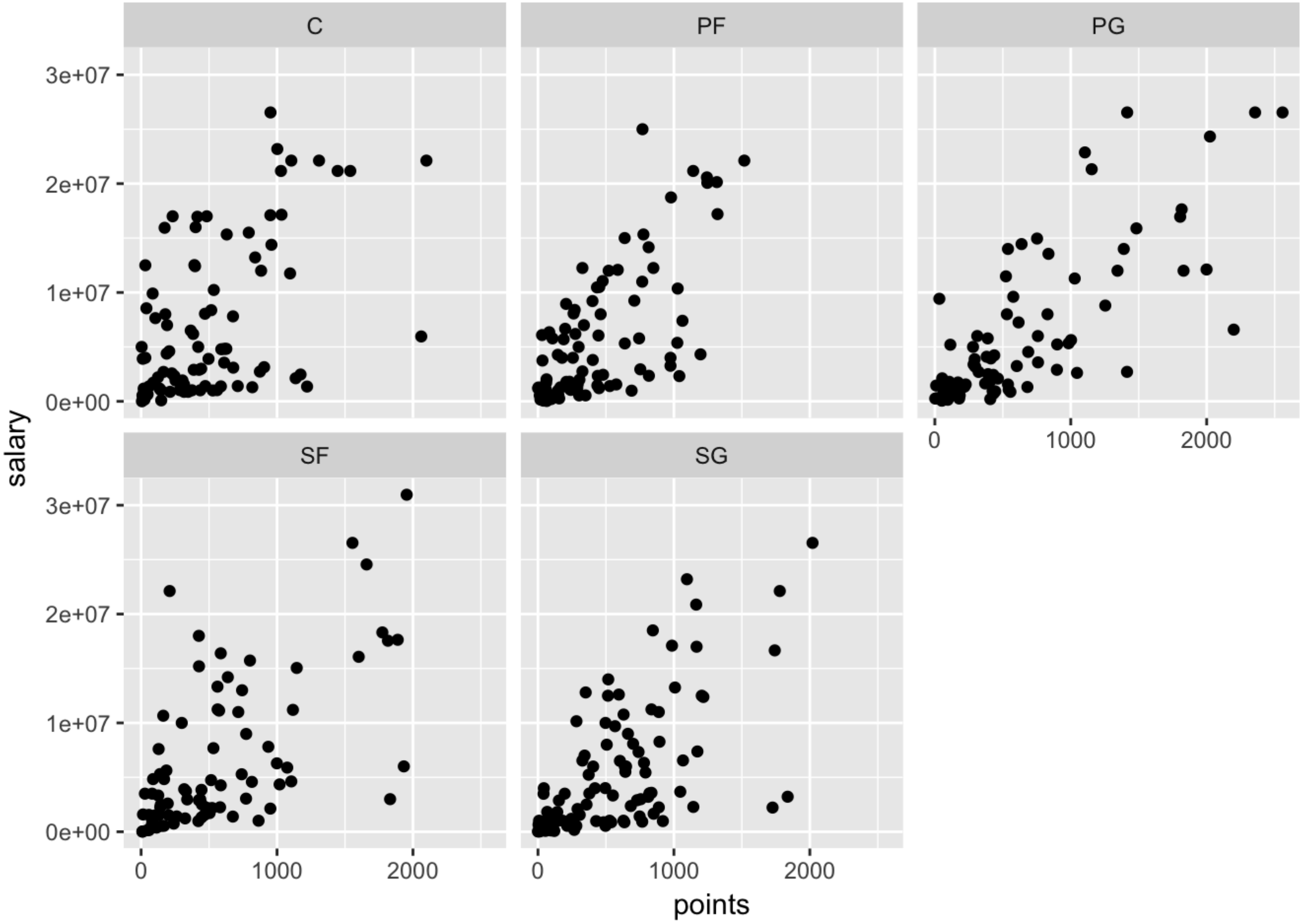


Faceting

One of the most attractive features of `"ggplot2"` is the ability to display multiple **facets.** The idea of facets is to divide a plot into subplots based on the values of one or more categorical (or discrete) variables.

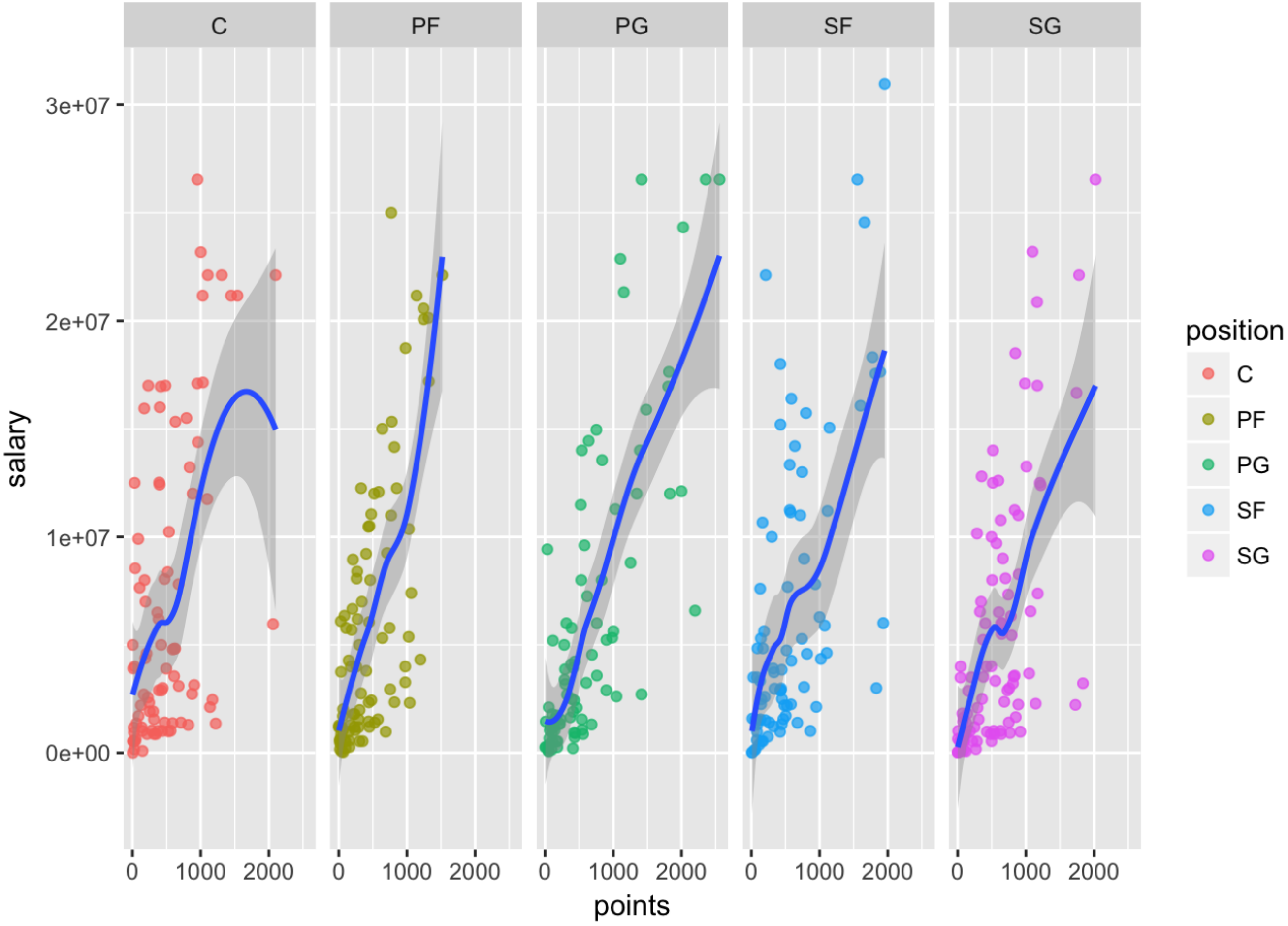
Here's an example. What if you want to get scatterplots of `points` and `salary` separated (or grouped) by `position` ? This is where faceting comes handy, and you can use `facet_warp()` for this purpose:

```
# scatterplot by position
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point() +
  facet_wrap(~ position)
```

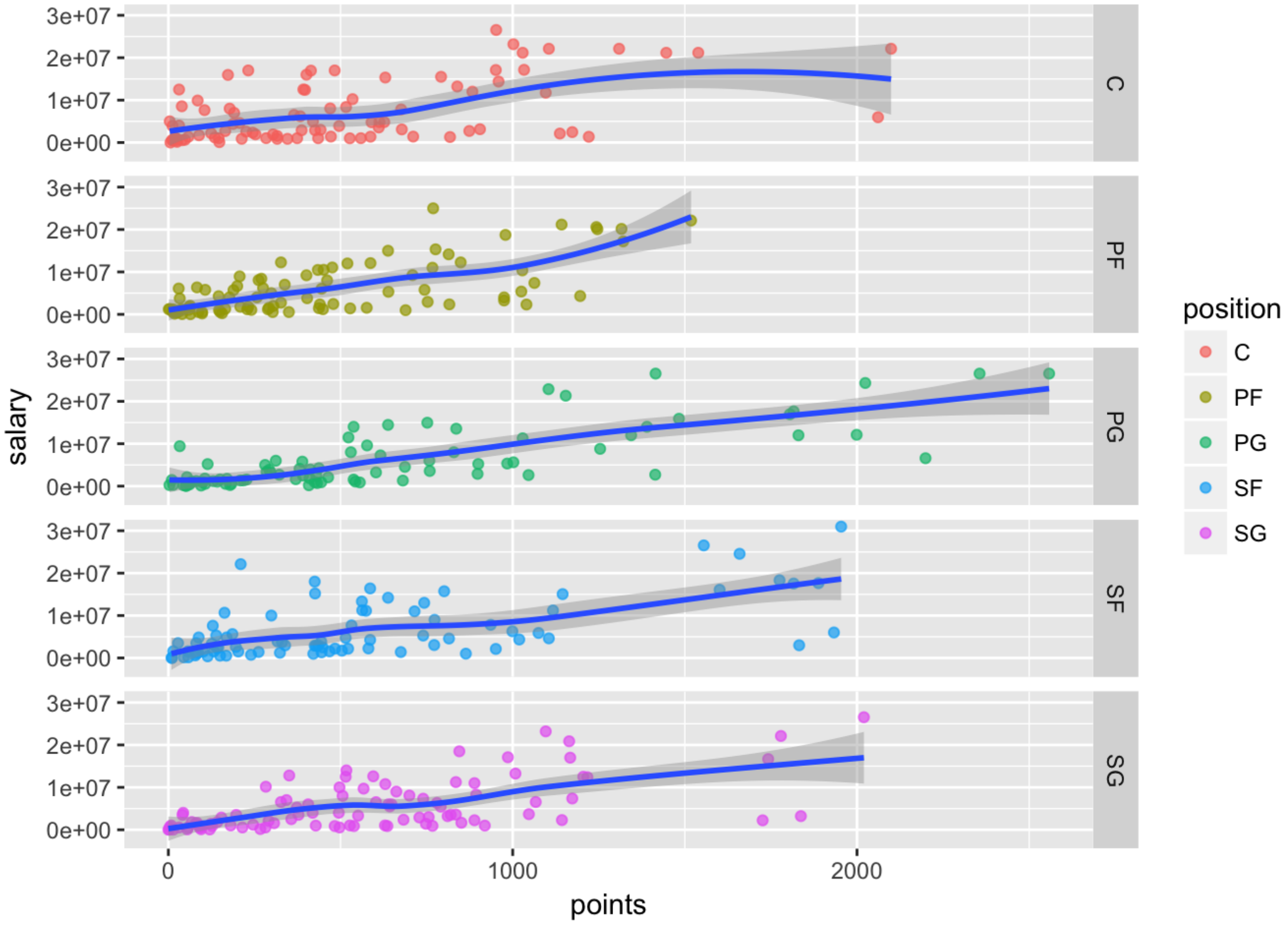


The other faceting function is `facet_grid()` , which allows you to control the layout of the facets (by rows, by columns, etc)

```
# scatterplot by position
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point(aes(color = position), alpha = 0.7) +
  facet_grid(~ position) +
  geom_smooth(method = loess)
```



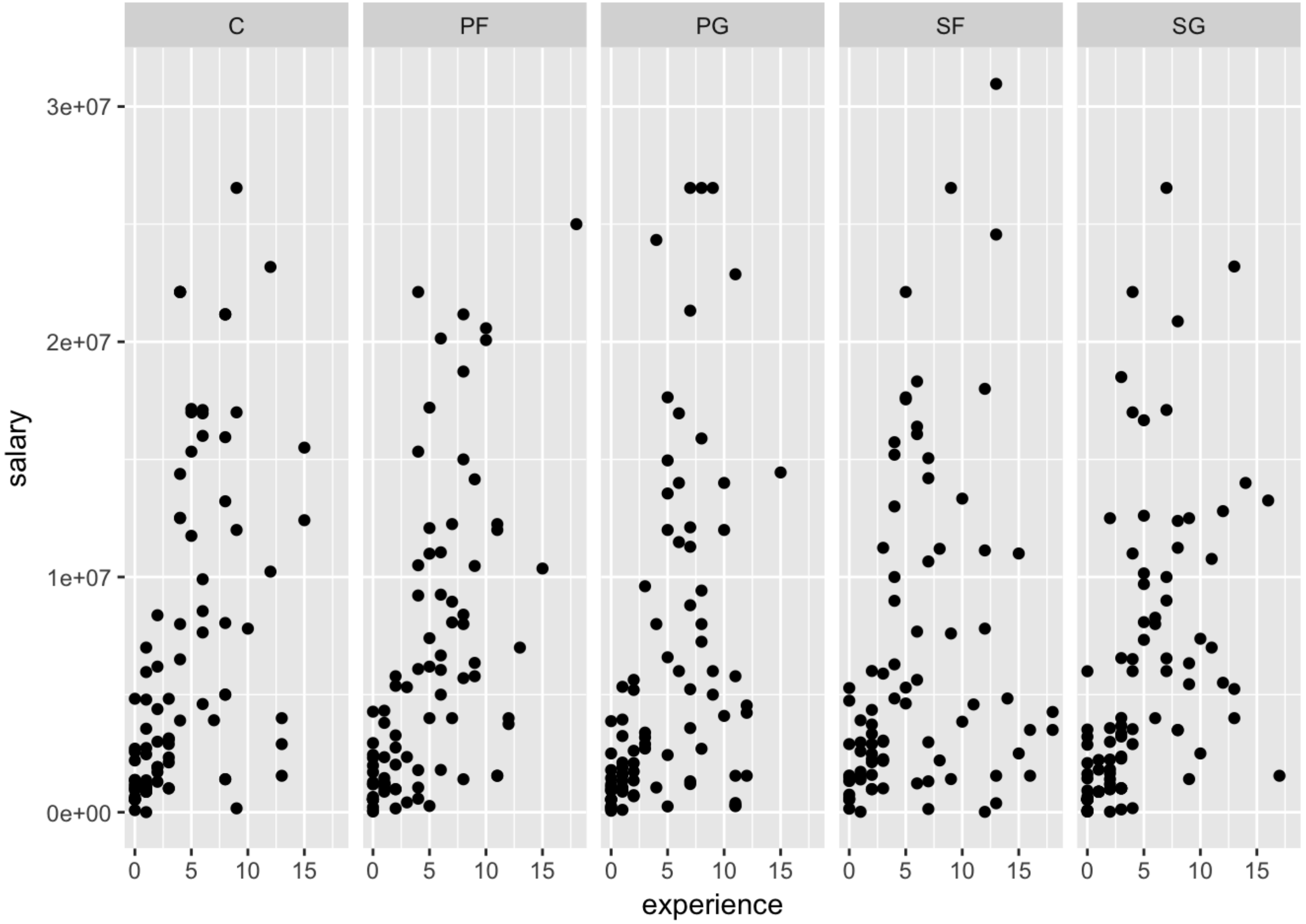
```
# scatterplot by position
ggplot(data = dat, aes(x = points, y = salary)) +
  geom_point(aes(color = position), alpha = 0.7) +
  facet_grid(position ~ .) +
  geom_smooth(method = loess)
```



Your turn:

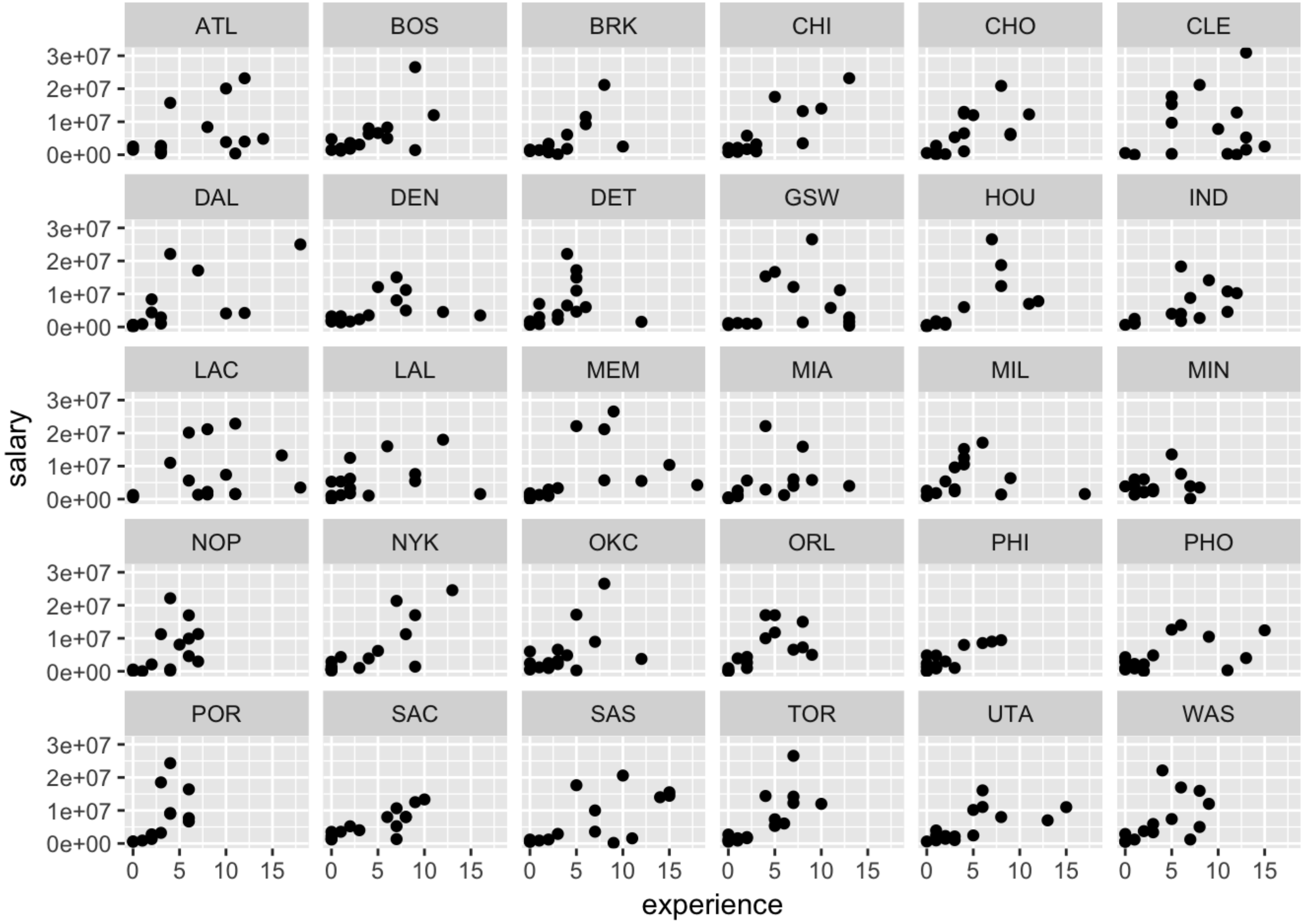
- Make scatterplots of `experience` and `salary` faceting by `position`

```
ggplot(data = dat, mapping = aes(x = experience, y = salary)) +
  geom_point() +
  facet_grid(~ position)
```



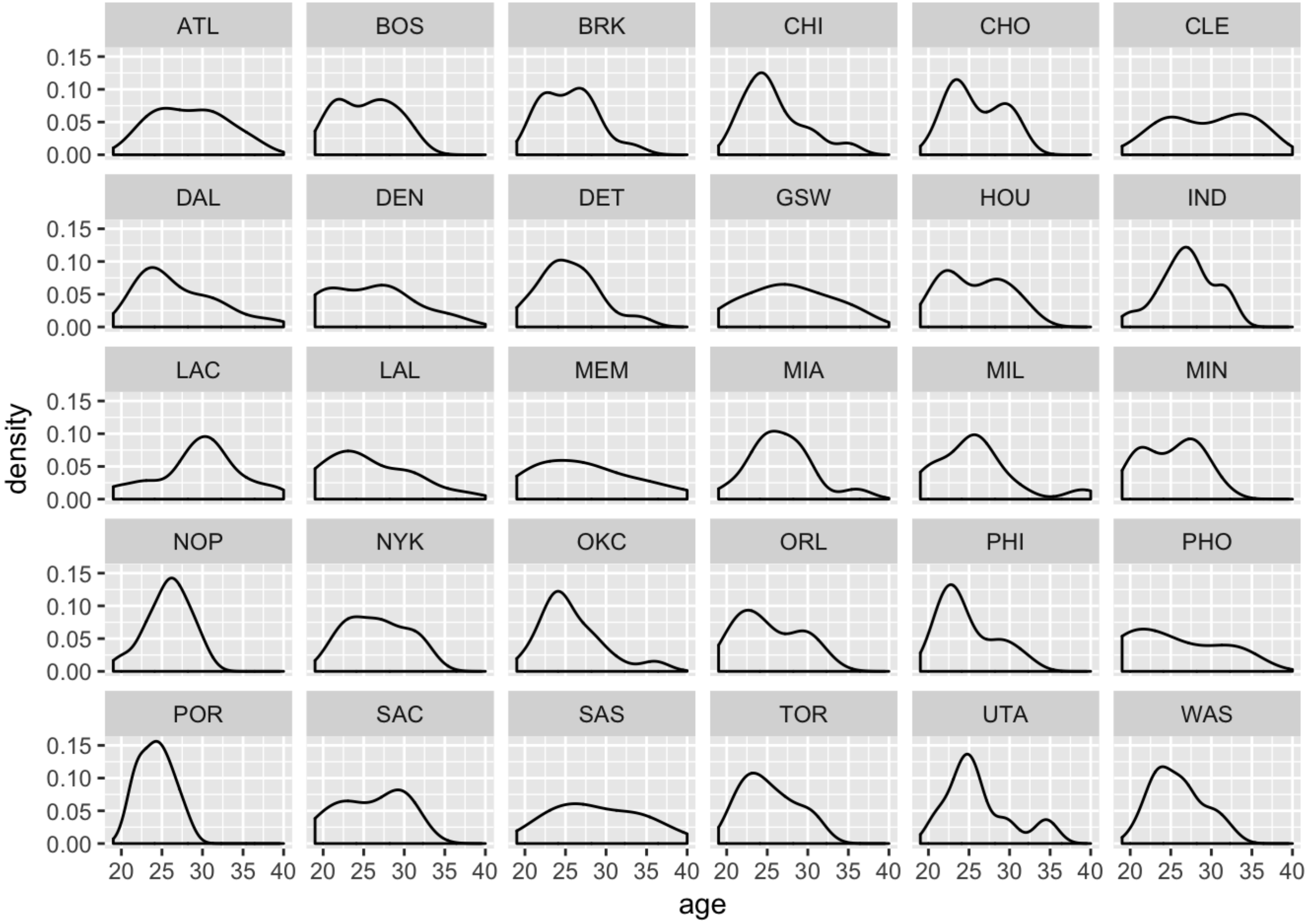
- Make scatterplots of `experience` and `salary` faceting by `team`

```
ggplot(data = dat, mapping = aes(x = experience, y = salary)) +
  geom_point() +
  facet_wrap(~ team)
```



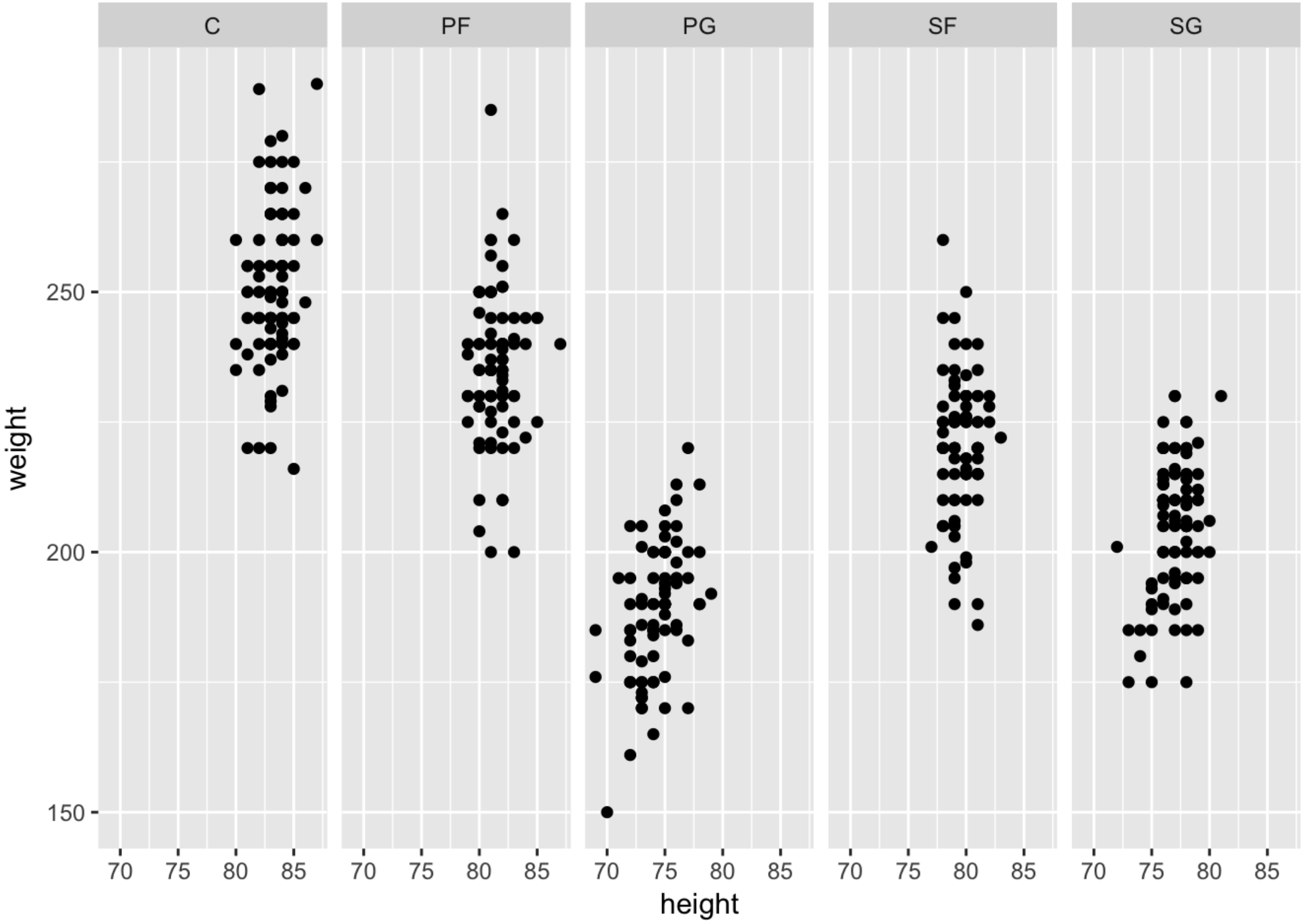
- Make density plots of `age` faceting by `team`

```
ggplot(data = dat) +
  geom_density(aes(x = age)) +
  facet_wrap(~ team)
```



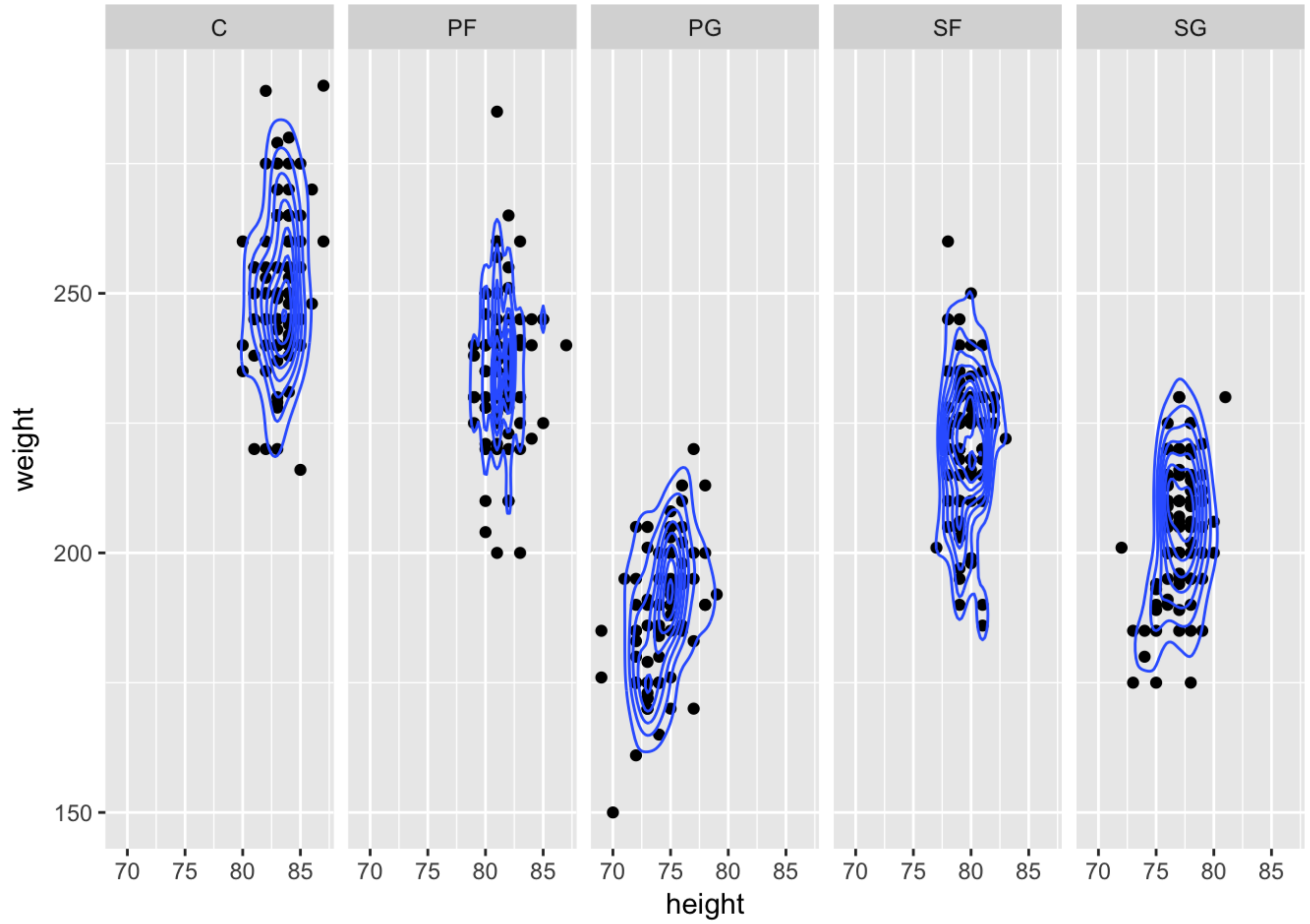
- Make scatterplots of `height` and `weight` faceting by `position`

```
ggplot(data = dat, mapping = aes(x = height, y = weight)) +
  geom_point() +
  facet_grid(~position)
```



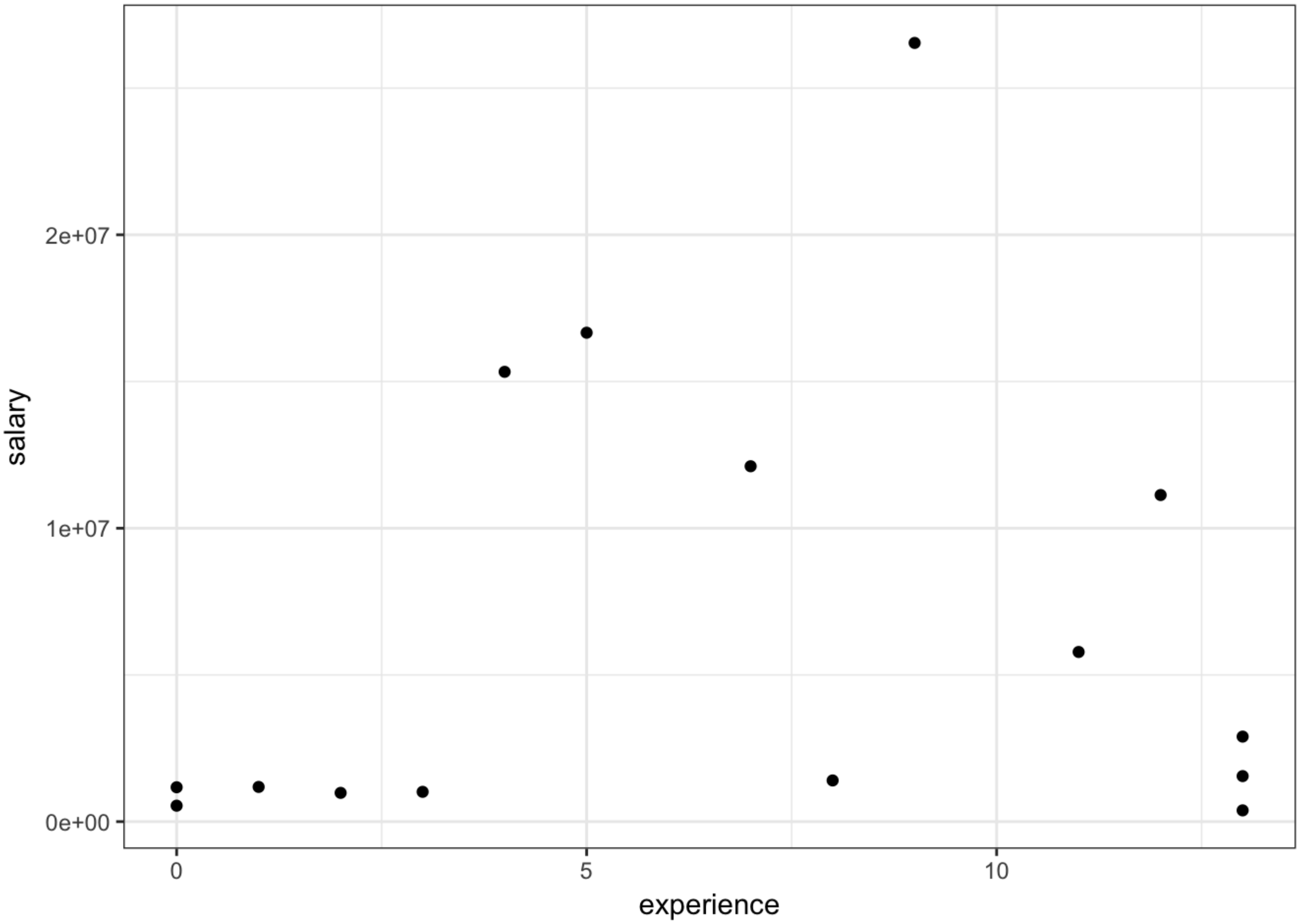
- Make scatterplots of `height` and `weight` , with a 2-dimensional density, `geom_density2d()` , faceting by `position`

```
ggplot(data = dat, mapping = aes(x = height, y = weight)) +
  geom_point() +
  geom_density2d() +
  facet_grid(~position)
```



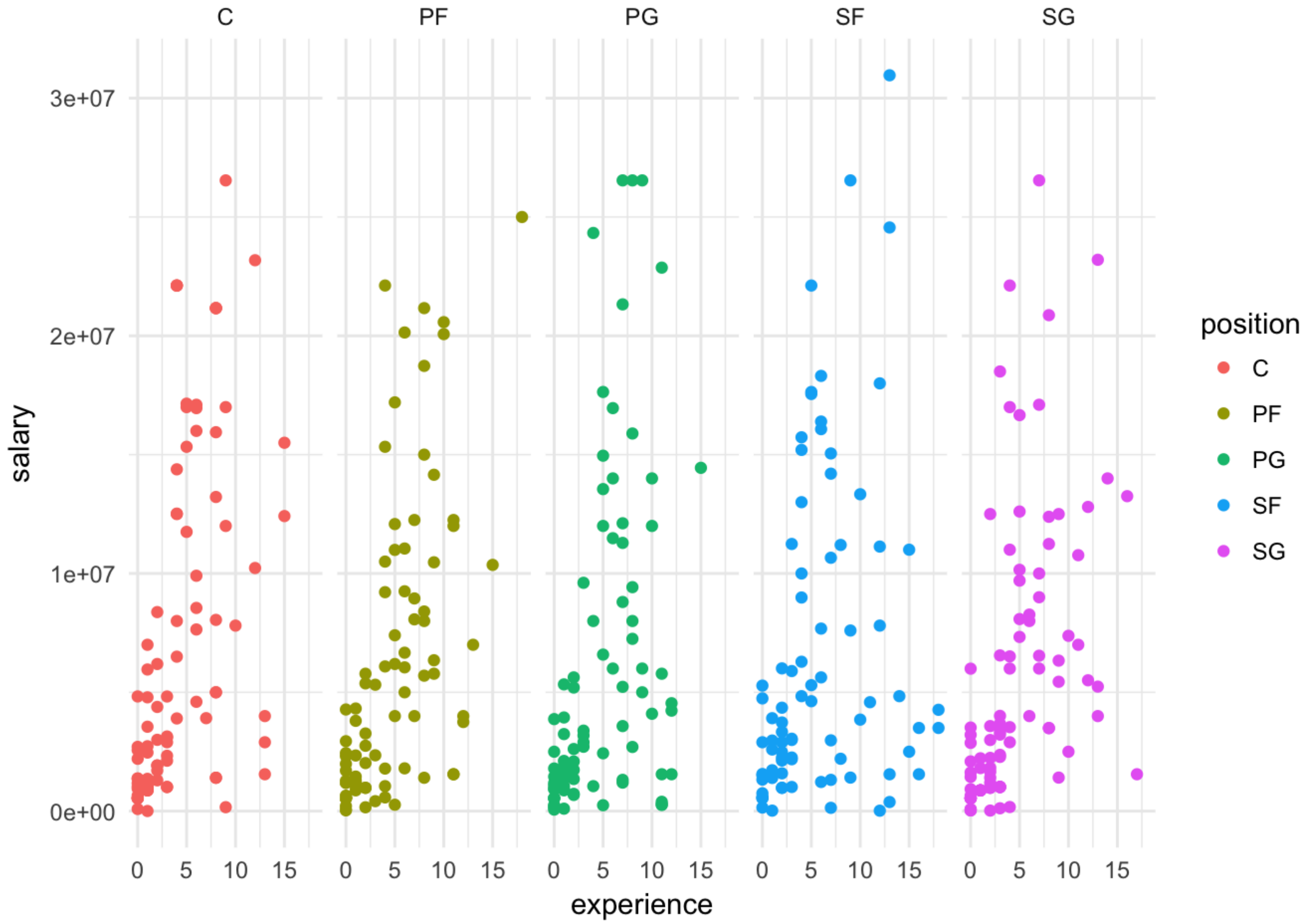
- Make a scatterplot of `experience` and `salary` for the Warriors, but this time add a layer with `theme_bw()` to get a simpler background

```
ggplot(data = GSW, mapping = aes(x= experience, y= salary)) +
  geom_point() +
  theme_bw()
```



- Repeat any of the previous plots but now adding a leyer with another theme e.g. `theme_minimal()` , `theme_dark()` , `theme_classic()`

```
ggplot(data = dat, mapping = aes(x = experience, y = salary)) +
  geom_point(aes(color = position)) +
  facet_grid(~ position) +
  theme_minimal()
```



More shell commands

Now that you have a bunch of images inside the `images/` subdirectory, let's keep practicing some basic commands.

- Open the terminal.
- Move inside the `images/` directory of the lab.
  cd ../images/
- List the contents of this directory.
  ls
- Now list the contents of the directory in *long format*.
  ls -l
- How would you list the contents in long format, by time?
  ls -lt
- How would you list the contents displaying the results in reverse (alphabetical)? order
  ls -r
- Without changing your current directory, create a directory `copies` at the parent level (i.e. `lab05/`).
  mkdir ../copies
- Copy one of the PNG files to the `copies` folder.
  cp reg-1.png ../copies
- Use the wildcard `*` to copy all the `.png` files in the directory `copies`.
  cp *.png ../copies

- Change to the directory `copies` .
  cd ../copies
- Use the command `mv` to rename some of your PNG files.
  mv reg-1.png reg.png
- Change to the `report/` directory.
  cd ../report
- From within `report/` , find out how to rename the directory `copies` as `copy-files` .
  mv ../copies ../copy-files
- From within `report/` , delete one or two PNG files in `copy-files` .
  rm ../copy-files/reg.png
- From within `report/` , find out how to delete the directory `copy-files` .
  rm -r ../copy-files