Data: Wrangle & Display It With (Relative) Ease

Korn Ferry Institute

# Data Cleaning

In this tutorial, we will be reading in two data sets from a fictional company, one being data on the employees’ demographics and the other some fictional test scores. We will be cleaning and reformatting them to make them usable, joining them together, and producing some graphics from them.

## Dependencies and Setup

Make sure you have the following packages installed

* openxlsx
* reshape2
* dplyr
* magrittr
* ggplot2
* ggthemes
* roperators

I’ve attached an installation script that you can run to make sure the packages are all there:

req\_pkgs <- c("openxlsx", "reshape2", "dplyr", "magrittr",   
 "ggplot2", "ggthemes", "roperators")  
   
your\_pkgs <- rownames(installed.packages())  
new\_pkgs <- req\_pkgs[!(req\_pkgs %in% your\_pkgs)]  
  
# installing  
install.packages(new\_pkgs,  
 repos = "https://cran.rstudio.com/")

### Load Required Packages

require(openxlsx)  
require(reshape2)  
require(dplyr)  
require(magrittr)  
require(ggplot2)  
require(ggthemes)  
require(roperators)

### Load Data

To load the data, we want to make sure that we are in the correct directory. If you were able to open your “Rproject” in RStudio, you should type the following code into R.

setwd("exercises")

Otherwise (in RStudio), click: Session –> Set Working Directory –> Choose Directory and select the “exercises” folder within “siop-data-wrangle-and-vis”.

In the data folder, there are two data sets:

* A csv called “employee\_data.csv”
* An excel workbook called “survey\_results.xlsx”

To read in the csv data, we can use base R’s read.csv function (which is using the read.table function, which you might see in other scripts. Csvs are plain-text flat files and can be opened in any program or programming language.

**Here is what is going on in the following code:**

* Create a variable called employees
* Into that value place the output of read.csv()
* read.csv() is going to go back one folder (../) and then look for employee\_data.csv in the data folder
* The stringsAsFactors argument prevents the parsing function to turn character strings into factors (which look like character strings but are treated as numbers in R)

employees <- read.csv("../data/employee\_data.csv", stringsAsFactors = FALSE)

To read in the Excel workbook, we are using a package (openxlsx, although there are many others). Excel workbooks are not equivalent to text files so cannot be read, opened, processed by any program.

survey <- read.xlsx("../data/survey\_results.xlsx")

## Inspect Data

Now that our data has been read into R as data.frames (which are just a combination of vectors all of the same length), we can inspect it. If you’re in RStudio you can go to the environment pane (usually top-right) and click on the names of either dataframe to open a preview of it.

We can also take a quick look at the data programatically by looking at the head (first 6 rows) like so:

head(employees)

## employee\_id gender age h\_date sector  
## 1 100 female 26 7/1/2013 manufacturing  
## 2 101 male 44 12/1/2007 finance  
## 3 102 female .. 2/1/2009 finance  
## 4 103 female 36 2/1/2007 finance  
## 5 104 female 25 8/1/2006 manufacturing  
## 6 105 male 45 9/1/2006 sales

**Note** that the age column has a “..” where there’s a missing value (normally NA in R) and the hiring date h\_date is in a non-standard format (data people like dates to be in yyyy-mm-dd format). We’ll circle back to fix those issues later.

Now, if we look at the survey data, we can see something troubling…

head(survey)

## question 100 101 102 103 104 105  
## 1 q1 2.13149045 3.867981 2.420906 3.289151 2.3485518 3.506212  
## 2 q2 0.51780778 4.132449 1.808998 2.585556 -0.1421079 2.083680  
## 3 q3 0.05791976 5.980333 3.673018 2.149323 1.1145136 2.641385  
## 4 q4 1.00000000 1.000000 1.000000 0.000000 0.0000000 1.000000  
## 106 107 108 109 110 111 112 113  
## 1 3.882452 3.940335 3.506212 3.506212 4.374457 3.578566 3.962041 3.361505  
## 2 3.502425 2.936289 2.148384 2.917961 2.895670 3.147671 7.152177 3.291777  
## 3 4.077896 1.959868 2.798916 4.115294 3.084127 4.100692 8.278358 3.322040  
## 4 1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000 1.000000  
## 114 115 116 117 118 119 120 121  
## 1 2.782674 2.276198 3.433858 3.216797 3.578566 2.565613 3.578566 3.889687  
## 2 2.497465 1.929552 2.989114 3.357682 2.352637 2.457335 3.320879 3.785178  
## 3 3.183620 2.395780 3.316773 3.044401 3.423696 2.560668 3.485354 4.698280  
## 4 0.000000 0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 0.000000  
## 122 123 124 125 126 127 128 129  
## 1 3.144443 3.072090 2.348552 2.855028 3.289151 3.867981 2.782674 2.348552  
## 2 3.071804 1.869359 1.089839 2.293571 2.006269 6.321811 2.425611 1.214563  
## 3 3.194279 2.226997 2.335855 2.313168 3.840041 5.546722 3.317267 1.603046  
## 4 1.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000  
## 130 131 132 133 134 135 136 137  
## 1 1.9867829 3.506212 3.216797 3.867981 3.723273 3.144443 2.855028 2.999736  
## 2 0.9817197 2.613291 3.301003 3.142855 3.049639 2.376751 2.341615 2.418934  
## 3 0.2326721 3.022953 2.568945 3.960762 4.859625 1.595539 3.315714 2.170278  
## 4 0.0000000 1.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000  
## 138 139 140 141 142 143 144 145  
## 1 3.933099 3.918629 3.216797 3.216797 3.289151 2.4209055 2.637967 2.565613  
## 2 3.314585 3.614687 1.992890 2.479216 3.421912 0.5058558 2.338932 2.160124  
## 3 4.040668 5.186367 4.224766 2.214637 4.487737 0.4444191 1.959366 3.108142  
## 4 0.000000 0.000000 0.000000 1.000000 1.000000 0.0000000 0.000000 0.000000  
## 146 147 148 149 150 151 152 153  
## 1 3.867981 3.216797 2.999736 2.999736 2.855028 3.650920 3.578566 3.578566  
## 2 2.905016 2.509791 2.316495 2.803758 1.670794 3.156193 3.102668 3.082759  
## 3 4.424690 3.041618 3.648041 4.424986 3.640709 3.747599 4.846522 5.405982  
## 4 1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000  
## 154 155 156 157 158 159 160  
## 1 2.7103206 3.506212 1.6973678 3.940335 3.216797 3.216797 2.637967  
## 2 1.5013928 3.025786 1.0263598 3.038199 1.980829 2.513204 2.433322  
## 3 0.6999042 4.866015 0.9456502 4.222382 2.045063 3.431738 2.957433  
## 4 0.0000000 1.000000 0.0000000 0.000000 0.000000 0.000000 1.000000  
## 161 162 163 164 165 166 167 168  
## 1 4.229750 3.072090 2.999736 2.710321 2.276198 3.990982 2.999736 3.940335  
## 2 2.790407 2.655431 2.220098 2.740728 1.040550 2.413882 2.574067 5.237835  
## 3 3.569699 2.097536 3.122736 4.141372 1.735470 3.600336 4.472703 4.830236  
## 4 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 1.000000 0.000000  
## 169 170 171 172 173 174 175 176  
## 1 3.990982 3.361505 2.927382 3.578566 3.072090 2.565613 3.990982 2.710321  
## 2 3.644643 2.855320 1.633622 2.206948 1.841612 1.704275 1.929709 1.466348  
## 3 4.185921 1.117503 2.971561 1.956035 2.722470 2.116112 1.889112 1.849038  
## 4 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000  
## 177 178 179 180 181 182 183 184  
## 1 3.433858 3.506212 1.5526603 2.710321 2.927382 2.493259 3.216797 2.782674  
## 2 2.758920 3.635097 0.6917524 1.287638 2.498256 1.070747 2.533788 2.528042  
## 3 2.910113 3.533730 0.8521895 1.338543 3.206894 2.406618 3.664577 2.972188  
## 4 0.000000 0.000000 1.0000000 0.000000 0.000000 0.000000 1.000000 1.000000  
## 185 186 187 188 189 190 191 192  
## 1 3.361505 4.591519 2.782674 2.420906 3.433858 3.947570 3.578566 3.433858  
## 2 2.731754 2.331284 2.469187 1.569686 3.410481 2.526612 2.067535 2.874198  
## 3 2.693358 3.352807 2.474498 4.203045 3.627915 2.756945 2.157705 3.241517  
## 4 1.000000 1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000  
## 193 194 195 196 197 198 199 200  
## 1 2.855028 1.986783 1.8420754 2.710321 3.289151 3.361505 4.157396 3.940335  
## 2 1.850614 1.007718 0.7993656 1.906639 2.051740 2.084022 5.048400 2.519042  
## 3 2.999783 2.694328 1.7993210 1.754545 2.939523 2.095952 5.421519 1.849052  
## 4 0.000000 0.000000 1.0000000 1.000000 1.000000 0.000000 1.000000 0.000000  
## 201 202 203 204  
## 1 3.506212 2.637967 3.216797 2.927382  
## 2 3.726918 1.682378 2.508094 2.488210  
## 3 5.096794 2.364789 3.381088 3.874558  
## 4 1.000000 1.000000 1.000000 0.000000

It looks like some well-meaning but not-data-minded person went and stored the survey results in so-called “wide” format with one column per respondent and one row per question.

Before we can work with this data, **We have the following problems to solve:**

1. Missing values are coded as “..”, which has turned all of the numeric columns into strings!
2. Dates need to be transformed before they can be used.
3. Values from each person are placed in different columns.
4. Variables are separated by row, not column.

## Fix Data

### Fix Missing Values

The first thing we want to do is replace “..” with the standard missing value indicator in R (NA) and then convert the age column into a numeric column. (We can actually do this when reading the data into R, but we kept it here for the purposes of exposition). Think about the logic we want to use: pull out the age columns of employees, turn all cells with ".." into NA, and then convert the columns into numeric.

There are many ways to do this in R:

1. We could use *regular expression* substitution (using the gsub function). Regular expressions are beyond the scope of this workshop, but click <http://stackoverflow.com/questions/tagged/regex> for an example of something similar to this problem.
2. We could simply convert the column to numeric. Anything that is not “number-like” (like the "..") will be turned into NA. This is not ideal because other things might be affected, but it would work.
3. We could find all the cells that are “..” and replace them with missing (NA in R) by **logical indexing**.

Here, we’ll use simple **logical indexing** to find and replace all cells that have “..” with missing values (NA in R)

**Here is what is going on in the following code:**

* Take the age column from employees (accessed with $).
* In the age column, find all elements where age is "..".
* Replace those elements with a missing value, NA
* Turn the age column into a numeric column

employees$age[employees$age == ".."] <- NA  
employees$age <- as.numeric(employees$age)

### Fix Dates

The hiring date column (employees$h\_date) is currently a character string. We want to turn dates into a datetime variable so that we can do things like work out how many years someone has been at the company. To do this, we’ll use the base R date functions. Note that you can work with dates in in packages like lubridate.

In the following code, we need to tell R what format the date is currently in and R will magically convert it to the appropriate type.

**Here is what is going on in the following code:**

* Overwrite the employee\_data$h\_date column with the output of as.Date().
* Within as.Date(), transform the existing h\_date column.
* The original date is formatted “%m/%d/%Y” (which is telling R that we have text as: numeric month (%m) / numeric day (%d) / 4 digit year (%Y).

**Hint:** to see all available date formatting options, run ?strptime in the console

employees$h\_date <- as.Date(employees$h\_date, format = "%m/%d/%Y")

Now, lets work out how long they’ve worked here.

**Read the following code as:**

* Create a column in employee\_data called tenure.
* Into that column put the result of:
* …today’s date (Sys.Date()) minus the date people were hired
* …divided by ~365.25 days to give their tenure in years

employees$tenure <- as.numeric(Sys.Date() - employees$h\_date)/365.2422

And now let’s put people’s tenures into categories, let’s do: 0-1 year, 1-2 years, 2-5 years, 5-10 years, and 10+ years.

To accomplish this, we’ll user R’s cut() function to “cut” out numeric variable into categories.

cut() is using these arguments:

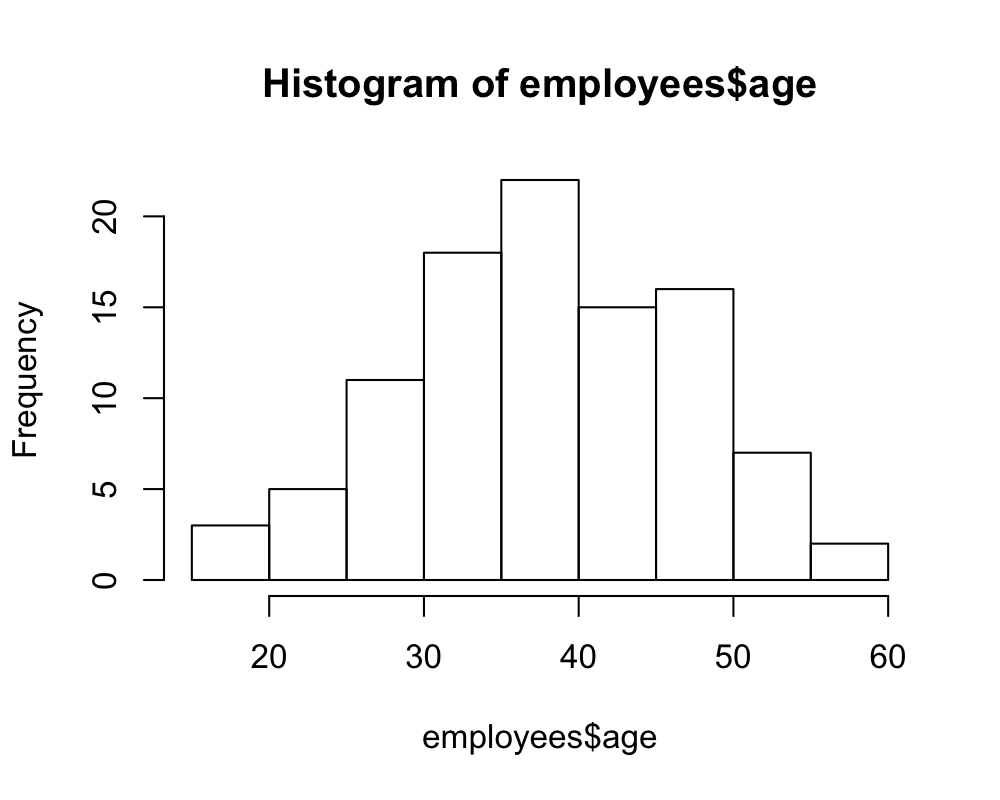
* the variable being transformed
* the break points (note 0 and Inf on the ends)
* the labels corresponding to the intervals between the breaks
* a logical flag telling it whether the breaks are including the number on the right of the interval. If FALSE it’s 0 to less than 1, 2 to less than 3; if TRUE it’s 0 to 1, greater than 1 to 2, greater than 2 to 3, etc.
* another flag telling R that you’d like the resulting factor levels to be ordered the way you specified it.

employees$tenure\_label <- cut(employees$tenure,  
 breaks = c(0, 1, 2, 5, 10, Inf),  
 labels = c("<1 year", "1-2 years",   
 "2-5 years", "5-10 years",   
 "10+ years"),  
 right = FALSE,  
 ordered\_result = TRUE)

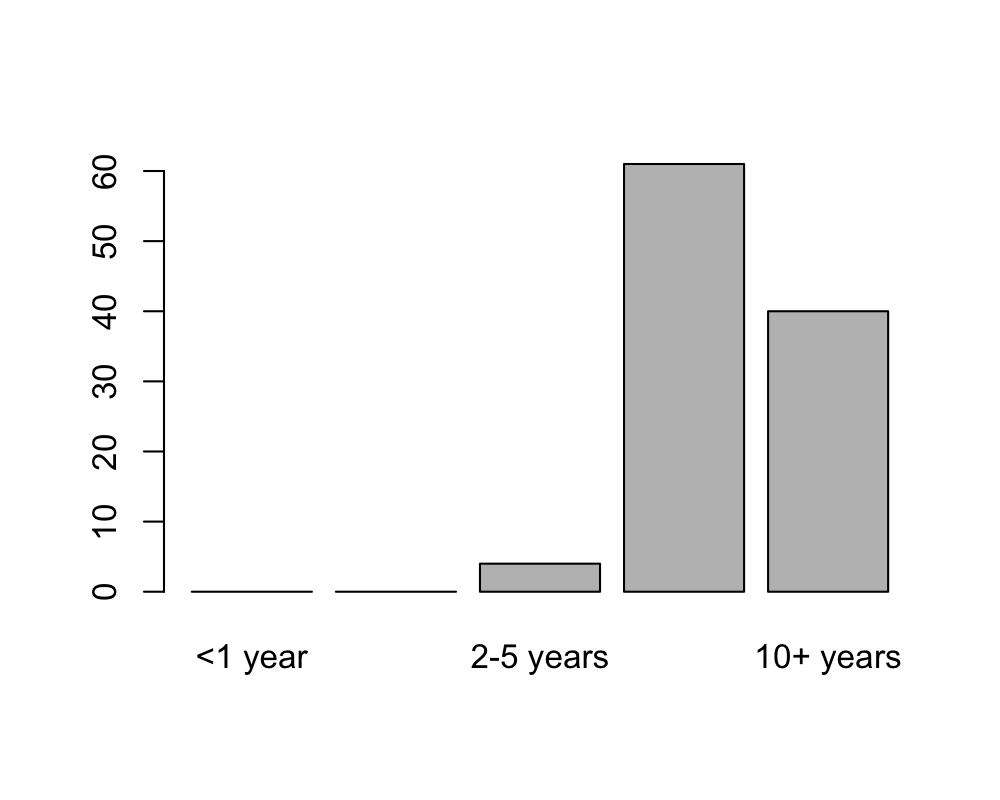
#### Inspect Data

Now checkout the distribution of the variables we just created! We can use quick-and-dirty plots for initial visualizations

hist(employees$age)



plot(employees$tenure\_label)



### Reshape Data

Things are looking promising! But we still have to do something about the scores. It would be nice to be able to break down our scores by employee demographics. But we have a problem: scores are in another dataframe that is in wide format with one column per candidate. We will use a package called reshape2 to turn the dataframe into a nice long format (one row per candidate and one column per question/variable). (There are other packages, like tidyr that can do the same thing for you, but tidyr has been changing a lot recently).

First, let’s melt the data into a long format. To use melt() we need to specify the following:

* the dataframe we want to transform
* the id variable we want the ‘melted’ data mapped to
* the variable.name (i.e., the name of the column where the names of the melted columns will go)
* the value.name (i.e., the name of the column where the values of the melted columns will go)

survey <- melt(survey,   
 id.vars = "question",   
 variable.name = "id",   
 value.name = "score")

Now, let’s take a peek at what survey looks like now:

head(survey)

## question id score  
## 1 q1 100 2.13149045  
## 2 q2 100 0.51780778  
## 3 q3 100 0.05791976  
## 4 q4 100 1.00000000  
## 5 q1 101 3.86798103  
## 6 q2 101 4.13244890

Now scores from each question are all in the same column. That will make it difficult if we want to compare question to question. Let’s put scores from each question into separate columns. To do this we use dcast() to **cast** the **d**ata.

To use dcast() we need to specify:

* the data we want to transform
* a formula that looks like: <row(s) for observations> ~ <column(s) to seperate out>
* the value.name (i.e., the name of the column where the values are that should be casted)

survey <- dcast(survey, id ~ question, value.var = "score")

Now, let’s take a look…

head(survey)

## id q1 q2 q3 q4  
## 1 100 2.131490 0.5178078 0.05791976 1  
## 2 101 3.867981 4.1324489 5.98033344 1  
## 3 102 2.420906 1.8089977 3.67301803 1  
## 4 103 3.289151 2.5855558 2.14932327 0  
## 5 104 2.348552 -0.1421079 1.11451361 0  
## 6 105 3.506212 2.0836801 2.64138535 1

Perfect! Now we have one row per observation and one column per variable! Next, let’s join the scores to the employee data by their ID number.

## Join Data

Alright, now we just need to join our dataframes, employees and survey, together. While you could force them together by sorting by id and using cbind() to effectively slap the columns together, that isn’t really a good idea in most real-world situations as the employee id numbers may not be perfectly aligned due to discrepancies between people in the employee database and people who took the survey. That’s why we’ll do a formal join on the dataframes.

If you’ve used SQL to query and manipulate data or VLOOKUP in Excel, this may be a familiar concept. Put simply, we want to find rows where the id variable matches between our two dataframes and copy over the corresponding survey results. In this case, since we’re appending survey results onto the employee data, we’re technically going to do a left outer join - i.e. keep all employee data on the left hand side and add survey results (on the right-hand side) if we have them.

The main types of join you’ll see used are:

* left outer (keep everything on the left, add data from the right if there’s a match)
* right outer (keep everything on the right, add data from the left if there’s a match)
* inner (only keep rows where there’s a match between the left and right tables)
* outer (keep all data and fill non-matched records with NAs)

Within R there are three main ways to join data:

* merge() in base R - it works, but the syntax is a little difficult
* data.table join by reference - lightning fast and great for big data but the syntax isn’t intuitive
* \_join functions from dplyr - easy to use, albeit not nearly as fast as data.table

Here, we’ll use left\_join from dplyr with the following logic:

* Make sure the id variables in both dataframes are the same type - character is a good thing to convert them to.
* Call left\_join() and supply the left and right data tables (employees and survey respectively), and by as a named character vector in the form of c(left\_id\_column = right\_id\_column)

employees$employee\_id <- as.character(employees$employee\_id)  
survey$id <- as.character(survey$id)  
all\_data <- left\_join(employees, survey,   
 by = c("employee\_id" = "id"))  
head(all\_data)

## employee\_id gender age h\_date sector tenure tenure\_label  
## 1 100 female 26 2013-07-01 manufacturing 5.763299 5-10 years  
## 2 101 male 44 2007-12-01 finance 11.345896 10+ years  
## 3 102 female NA 2009-02-01 finance 10.174071 10+ years  
## 4 103 female 36 2007-02-01 finance 12.175482 10+ years  
## 5 104 female 25 2006-08-01 manufacturing 12.679258 10+ years  
## 6 105 male 45 2006-09-01 sales 12.594383 10+ years  
## q1 q2 q3 q4  
## 1 2.131490 0.5178078 0.05791976 1  
## 2 3.867981 4.1324489 5.98033344 1  
## 3 2.420906 1.8089977 3.67301803 1  
## 4 3.289151 2.5855558 2.14932327 0  
## 5 2.348552 -0.1421079 1.11451361 0  
## 6 3.506212 2.0836801 2.64138535 1

Hooray! Now we can do some aggregations!

## Basic Aggregations

Let’s dive into using dplyr, a popular R package for manipulating dataframes. While you can do pretty much everything dplyr does in base R, dplyr has a nice, easy to read syntax of plain-English verbs. Once we add pipes into the mix, you’ll be writing slick-looking and performant code in no time!

**Note:** When you’re more comfortable with R, I’d recommend looking into data.table which is admittedly harder to learn, but out performs dplyr (especially when working with datasets that are over a gigabyte in size) and is more suitable for production code due to its lack of dependencies and stable API.

### dplyr verbs

Here are the main functions (verbs) in dplyr:

***filter*** *Get a subset of* rows**select** Get a subset of *columns* ***group\_by*** *Determine columns to group by for grouped calculations* **summarise** Create aggregated data summaries and apply functions to data ***mutate*** *Add and modify a column (also works with grouped data)* **ungroup\*\*\t- Removes the effect ofgroup\_by\*\*\*rename\*\*\t- Rename columns \*\*\*arrange\*\*\t- Sort the data by selected columns \*\*\*do`** Do an arbitrary thing (advanced)

dplyr has a lot more functionality, but especially at the beginning, these are most of what you really need to know.

For example, let’s pull up all records of females in manufacturing who are over 40.

Within filter, we pass it the data as the first argument followed by logical statements to apply to rows. Similar to Excel or SQL we can use AND/OR logic, the only difference being that R (and most other programming languages), we use & for AND, and | for OR.

filter(all\_data,  
 gender == "female" & sector == "manufacturing" & age > 40)

## employee\_id gender age h\_date sector tenure tenure\_label  
## 1 168 female 53 2012-06-01 manufacturing 6.844773 5-10 years  
## 2 201 female 43 2006-10-01 manufacturing 12.512245 10+ years  
## q1 q2 q3 q4  
## 1 3.940335 5.237835 4.830236 0  
## 2 3.506212 3.726918 5.096794 1

The filter function allows us to pass statements as different arguments, and it will automatically combine them with &.

# we can also do the above this way ...  
filter(all\_data,  
 gender == "female",  
 sector == "manufacturing",  
 age > 40)

## employee\_id gender age h\_date sector tenure tenure\_label  
## 1 168 female 53 2012-06-01 manufacturing 6.844773 5-10 years  
## 2 201 female 43 2006-10-01 manufacturing 12.512245 10+ years  
## q1 q2 q3 q4  
## 1 3.940335 5.237835 4.830236 0  
## 2 3.506212 3.726918 5.096794 1

What if we wanted to filter and then select? We could do something like this:

# we can also use %in% for "gender == 'female' | gender == 'male'"  
filtered\_data <- filter(all\_data,   
 gender == "female" | gender == "male",  
 sector == "manufacturing",  
 age > 40)  
select(filtered\_data,  
 gender, age, sector, q1:q4)

## gender age sector q1 q2 q3 q4  
## 1 male 50 manufacturing 3.962041 7.152177 8.278358 1  
## 2 male 47 manufacturing 3.578566 3.320879 3.485354 1  
## 3 male 45 manufacturing 3.867981 6.321811 5.546722 0  
## 4 male 41 manufacturing 3.216797 3.301003 2.568945 0  
## 5 male 44 manufacturing 3.578566 3.102668 4.846522 0  
## 6 female 53 manufacturing 3.940335 5.237835 4.830236 0  
## 7 female 43 manufacturing 3.506212 3.726918 5.096794 1

# Note the little trick on q1:q4!!

…But, that starts to get a bit cumbersome to read. Instead we can use pipes! Pipes help us string functions together so that the output of one function is the input of the next.

## Using Pipes (%>%)

magrittr provides the pipe opperator, %>% to funnel the output of one function directly into another by automatically setting the first argument of any function after a pipe to be the output from the previous function (although we can do more complicated things with pipes).

To read a pipe simply, just make a mental map that %>% = “THEN”

For example, if I take the above query, I can write a pipeline that says:

* Take all\_data THEN (%>%)
* filter it THEN (%>%)
* select the columns that I want

all\_data %>%   
 filter(gender == "female" | gender == "male",  
 sector == "manufacturing",  
 age > 40) %>%  
 select(gender, age, sector, q1:q4)

## gender age sector q1 q2 q3 q4  
## 1 male 50 manufacturing 3.962041 7.152177 8.278358 1  
## 2 male 47 manufacturing 3.578566 3.320879 3.485354 1  
## 3 male 45 manufacturing 3.867981 6.321811 5.546722 0  
## 4 male 41 manufacturing 3.216797 3.301003 2.568945 0  
## 5 male 44 manufacturing 3.578566 3.102668 4.846522 0  
## 6 female 53 manufacturing 3.940335 5.237835 4.830236 0  
## 7 female 43 manufacturing 3.506212 3.726918 5.096794 1

Now, let’s say I want to aggregate that last output by gender and summarize the average scores for each question. We can just add more pipes.

* Take all\_data THEN (%>%)
* filter it THEN (%>%)
* select the columns that I want THEN (%>%)
* group\_by gender (so everything we do gets done to males and females seperately) THEN (%>%)
* summarize the average survey results

all\_data %>%   
 filter(gender == "female" | gender == "male",  
 sector == "manufacturing",  
 age >= 40) %>%  
 select(gender, age, sector, q1:q4) %>%  
 group\_by(gender) %>%  
 summarise(avg\_q1 = mean(q1),   
 avg\_q2 = mean(q2),  
 avg\_q3 = mean(q3),  
 avg\_q4 = mean(q4))

## # A tibble: 2 x 5  
## gender avg\_q1 avg\_q2 avg\_q3 avg\_q4  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 female 3.72 4.48 4.96 0.5  
## 2 male 3.64 4.64 4.95 0.4

For the last part, there are actually functions to help with summarizing lots of statements (such as summarize\_at), but those are beyond the scope of this tutorial.

**Now spend 10-15 minutes playing around creating your own pipelines tofilter, select, group\_by, summarize, and arrange your data to “gain actionable insights”** Get creative, work together, and feel free to ask for help if you get stuck!

## Plotting with ggplot2

When people talk about R’s graphics abilities, a lot of the time they are referring to a package called ggplot2 (often simply called ggplot) created by Hadley Wickham, one of the R Studio developers who is also the the author of dplyr (among other R packages). What sets ggplot2 apart from other graphics packages is that it allows you to create layeredgraphics with its own syntax commonly referred to as the “Grammar of Graphics”.

Code for ggplot2 can seem off putting at first glance – think of creating a plot in R the same way you’d paint a picture on canvas:

1. You need to get your canvas
2. You add the basic shapes and objects
3. You add details
4. You apply color
5. You add finishing touches

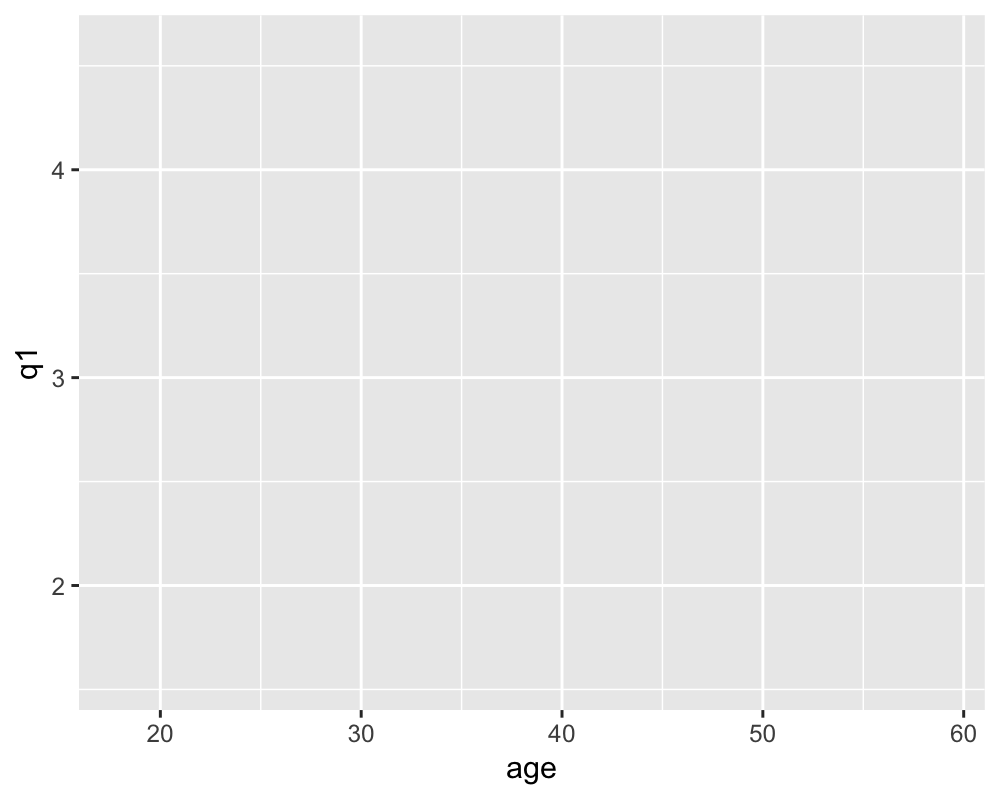
You can either create your plot directly or save it as an object. In the next few examples we will draw our plots directly to get some practice with the syntax.

### Creating a Canvas

The first step in making a pretty picture is setting out where it’s going to be drawn. This is done by creating a basic ggplot() object which is told what the data are and (usually) what the x and y axes will be. To do that, you need to feed in the data as the first argument (or with data = your\_data) and set up some **aesthetic mapping** in the aes() argument.

Anything inside the aes() argument tells ggplot how to map objects in the plot, for example what x and y are, what variable to color by, etc. Let’s start setting up a basic plot of responses for q1 vs age in all\_data

my\_plot <- ggplot(all\_data, aes(x = age, y = q1))  
my\_plot



Note that while we specified the axes, did not indicate what to put on them, and are left with a blank plot. To add a graph, we need to specify the **geometry** you want to add. To change things like the background, font, and so on, you need to specift a **theme**.

### Geometries

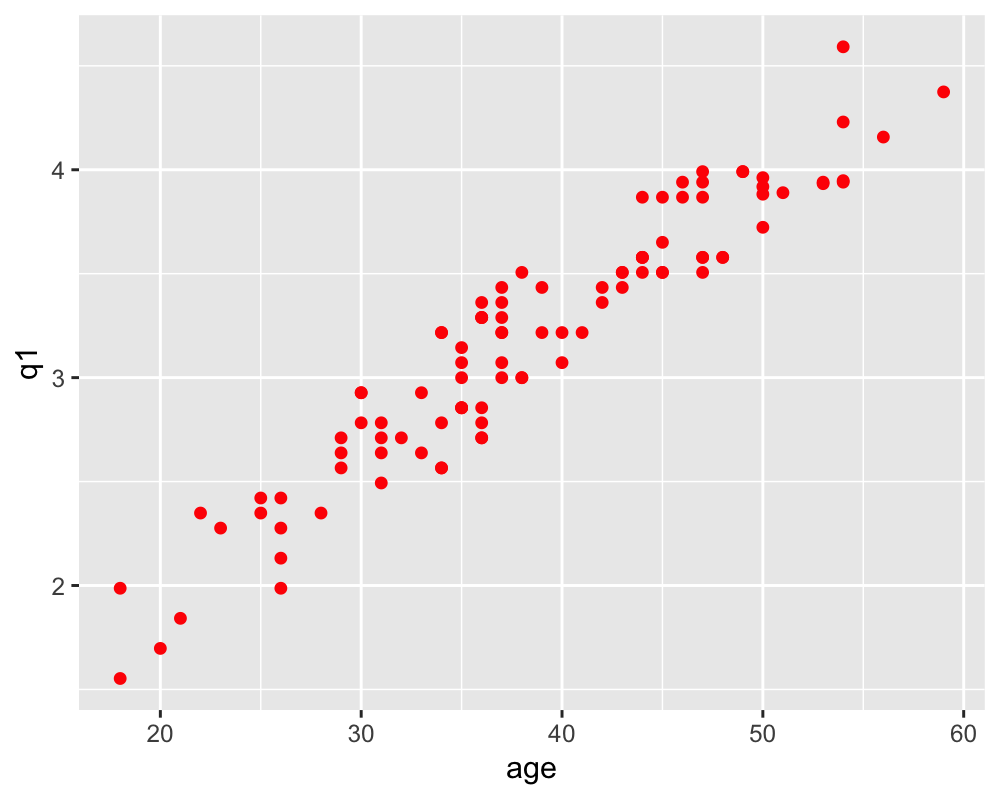
Many different types of graphs can be drawn with ggplot2 - far more than we can cover in this tutorial. We will only be able to cover some of the fundamental chart types. 3D plotting is possible, but it takes some additional steps, see <http://blog.revolutionanalytics>. com/2014/11/3-d-plots-with-plotly.html for example.

By default, each additional **geometry** inherits the **aesthetic mapping** from the main ggplot object, but you can specify these individually as well.

### Scatter Plots: geom\_point

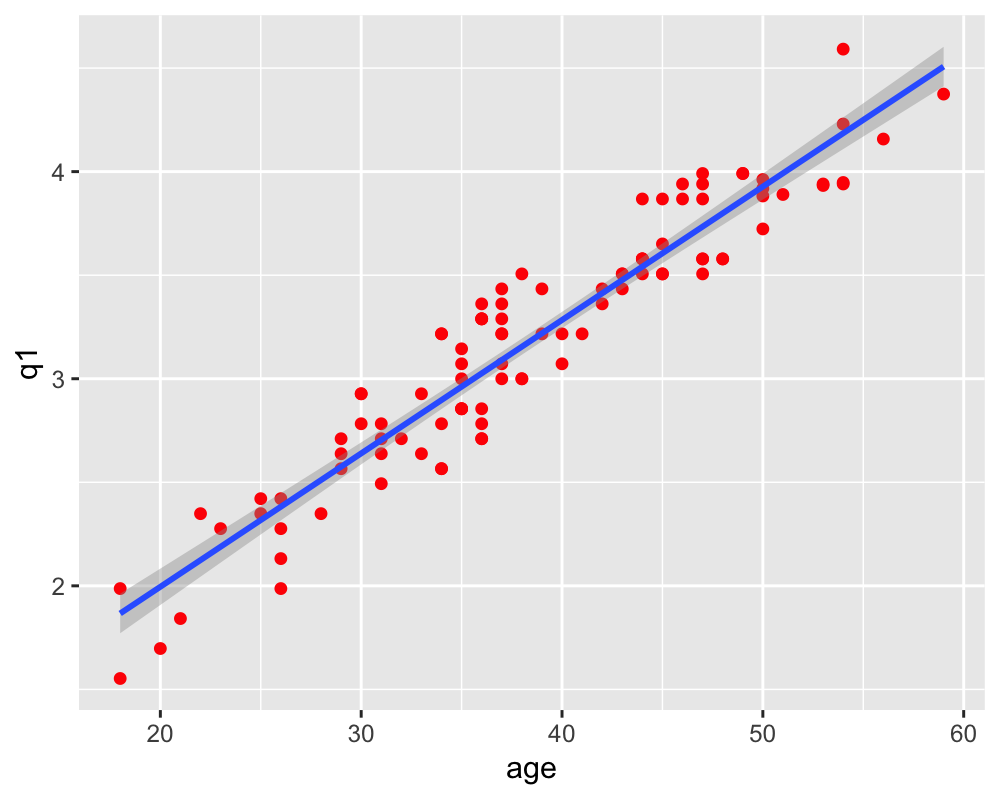
Let’s begin by adding a **layer** of red points (geom\_point) to our empty plot. To add layers in ggplot2 we just use + like so:

my\_plot <- my\_plot + geom\_point(color = "red")  
my\_plot



Adding a custom trendline in ggplot is flexible. By default, a LOESS smother is used, but this can easily be changed to become a simple linear trend, GAM, GLM, and so on. For example, to fit a linear trendline, you woud add a **geom\_smooth()** and specify that the smoothing method is a linear model (lm)

my\_plot <- my\_plot + geom\_smooth(method = "lm")  
my\_plot



**Remember** there are many more arguments to use with geom\_smooth, resulting in lots of different fits!

### Bar charts: geom\_bar

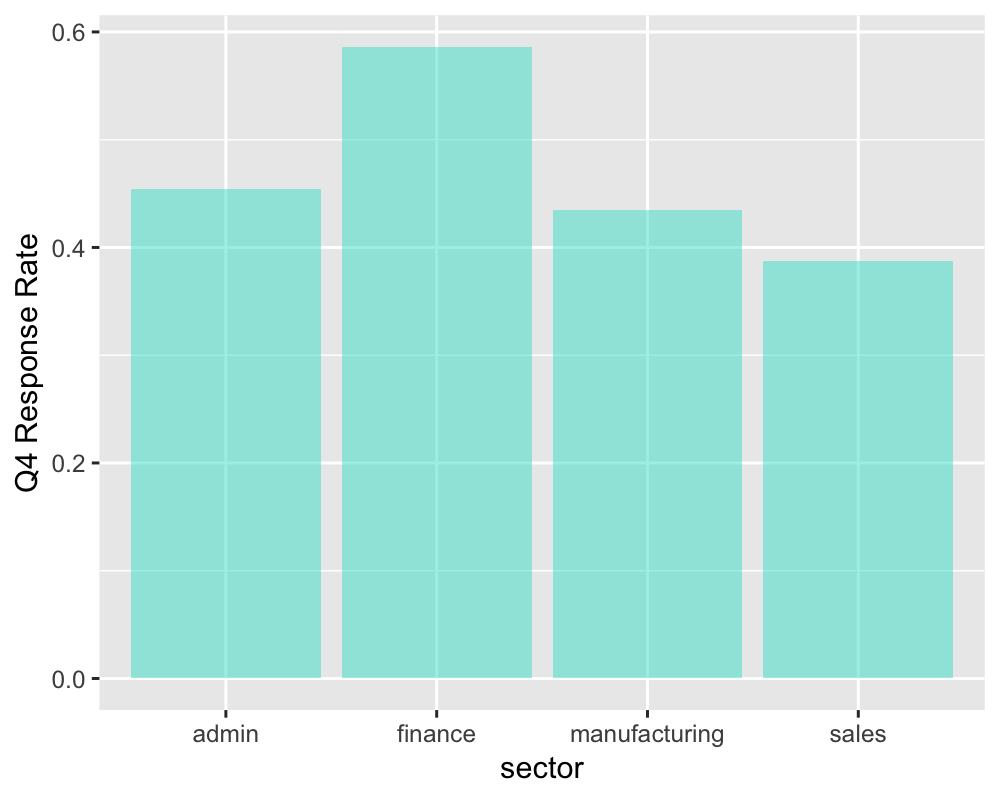
To create a bar chart, to make things easy you will need to add a geom\_bar object and specify stat = "identity" (by default geom\_bar wants aggregate by counting, but changing stat to “identity”" just means draw a specific value) and use a subset of data that you have pre-made.

For example, to make a bar chart out of the average response rate to q4 in our data:

1. Make a data set to plot (by now you should be realising that data manipulation goes hand in hand with plotting)
2. Create a plot
3. Add a geom\_bar to it.

For simplicity here, we are combining data aggregation and plotting. I’ve found it confusing to read a pipeline that is very complicated. In general, you should not combine data aggregation and plotting or it could be difficult to see what steps are being taken to construct a plot and where the data manipulation ends and plotting begins. We’ll also set the fill color of the bars to be turquoise and set them to be partially transparent with alpha. Then, we’ll also add a custom y-axis label too!

# data aggregation  
all\_data %>% #take all\_data THEN  
 group\_by(sector) %>% #group it by sector THEN  
 summarise(p = mean(q4)) %>% #average of q1 across each group THEN  
   
 # plotting  
 ggplot(aes(x = sector, y = p)) + # put it into a plot and add...  
 geom\_bar(stat = "identity",  
 fill = "turquoise",  
 alpha = 0.5) + # ...bars and...  
 ylab("Q4 Response Rate") # ...a custom y axis label



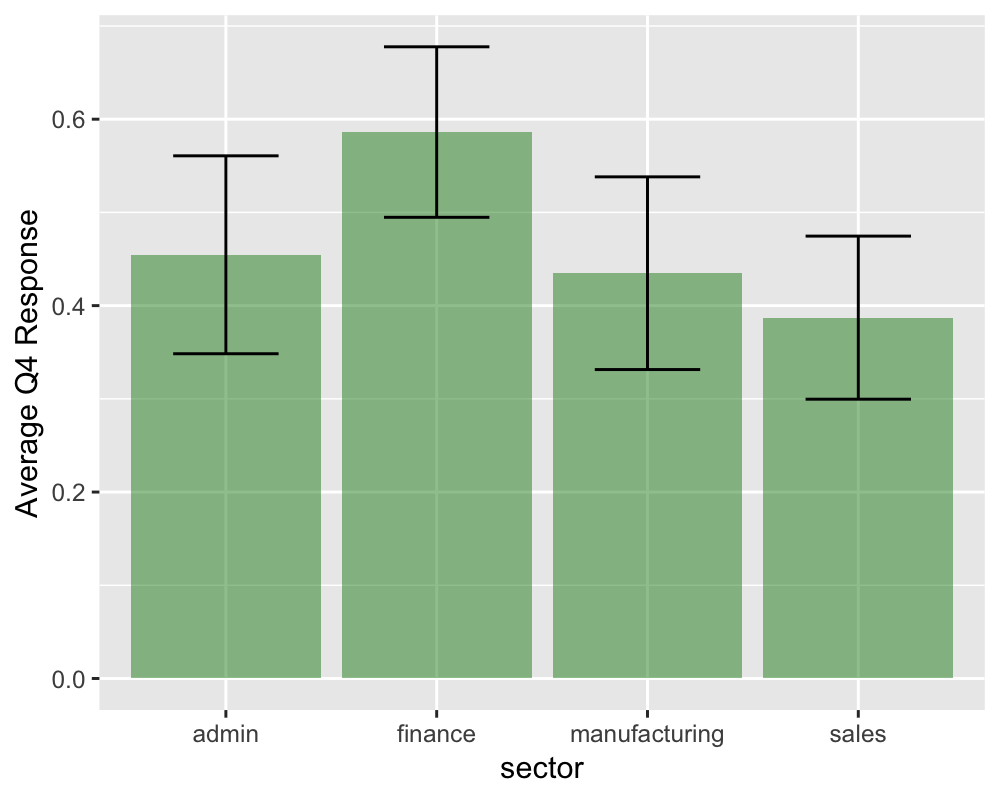
# Normally you should do the following for clarity  
# agg\_data <- all\_data %>%  
# group\_by(sector) %>%  
# summarise(p = mean(q4))  
#   
# ggplot(data = agg\_data,  
# aes(x = sector, y = p)) +   
# geom\_bar(stat = "identity",  
# fill = "turquoise",  
# alpha = 0.5) +  
# ylab("Q4 Response Rate")

### Error Bars: geom\_errorbar

To add an error bar to the above example, you would use geom\_errorbar() with a specific ymin and ymax in the aesthetic mapping. We’ll create the standard deviation as:

. Repeat the previous plot and add error bars with ymin and ymax set in the aes() argument and set the width of the bar caps to be 0.5 (i.e. they total half the bar’s width).

all\_data %>% #take all\_data THEN  
 group\_by(sector) %>% #group it by sector THEN  
 summarise(p = mean(q4),  
 q = 1 - p,  
 n = n(),  
 sd = sqrt((p \* q) / n)) %>% #average of q1 across each group THEN  
 ggplot(aes(x = sector, y = p)) + # put it into a plot and add...  
 geom\_bar(stat = "identity", fill = "forest green", alpha = 0.5) + # ...bars and...  
 geom\_errorbar(aes(ymin = p - sd, ymax = p + sd), width = 0.5) +   
 ylab("Average Q4 Response") # ...a custom y axis label



### Other Useful Geometries:

ggplot2 has many more types of plot that it can draw. Arguably, we could spend an entire day and still not cover them all in depth!

Here’s a list of other handy geometries to play around with in your spare time:

* Line charts: geom\_line
* Area charts: geom\_area
* Histograms: geom\_hist
* Boxplots: geom\_boxplot
* Density curves: geom\_density
* Violin plots: geom\_violin
* Dotplots: geom\_dotplot
* Area around a line: geom\_ribbon
* Hexagonal binning: geom\_hex
* Contour plots: geom\_contour
* Heatmaps: geom\_tile and geom\_density\_2d
* Labels and text: geom\_label and geom\_text

Look in R Studio’s Help > Cheatsheets > Data Visualization with ggplot2 for a handy guide or consult *The R Graphics Cookbook* for handy ggplot2 recipies.

### Colors, Shapes, and Sizes

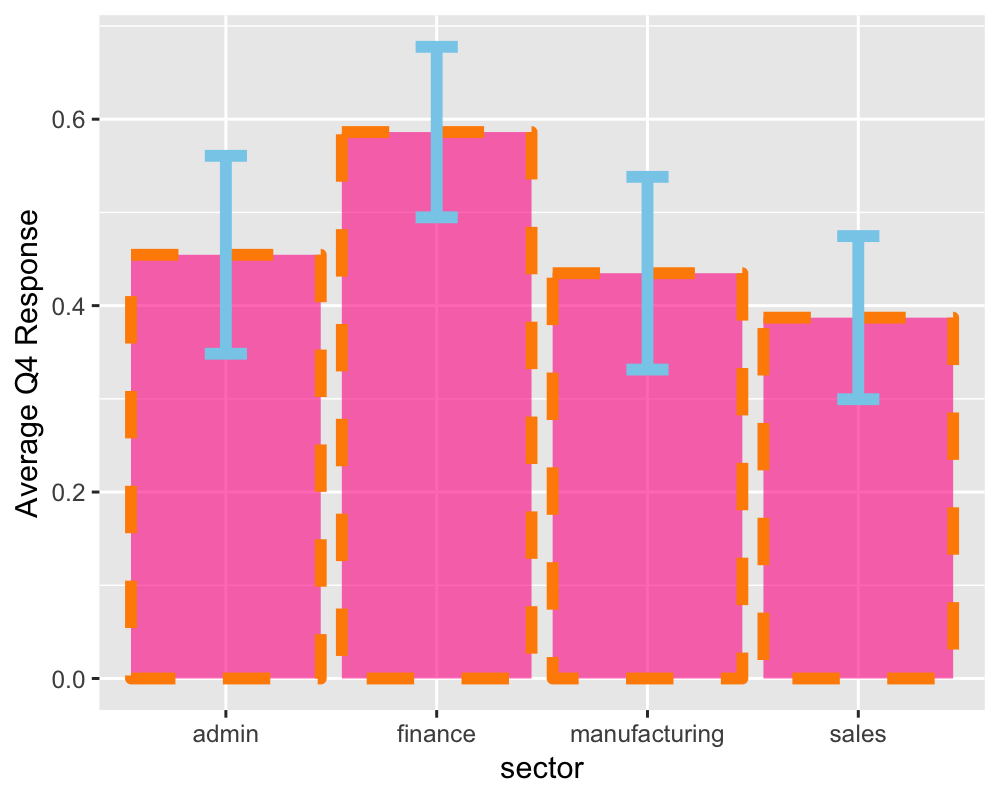
To apply a color, or specify the shape (for points), line type (for lines), and size (points and outlines) of an object, simply pass in the arguments:

* color = “color name”
* fill = “colour name” for colouring the inside of shapes
* shape = #
* linetype = “style”
* size = #

See <http://www.cookbook-r.com/Graphs/Shapes_and_line_types/> for more details on shapes and line types.

**Play around with the following code to improve on the plot it generates:**

all\_data %>% #take all\_data THEN  
 group\_by(sector) %>% #group it by sector THEN  
 summarise(p = mean(q4),  
 q = 1-p,  
 n = n(),  
 sd = sqrt((p\*q)/n)) %>%  
 ggplot(aes(x = sector, y = p)) +  
 geom\_bar(stat = "identity", fill = "deeppink", color = "darkorange",  
 size = 2, alpha = 0.6, linetype = "dashed") +   
 geom\_errorbar(aes(ymin = p - sd, ymax = p + sd),   
 width = 0.2, color = "skyblue", size = 2) +   
 ylab("Average Q4 Response") # ...a custom y axis label

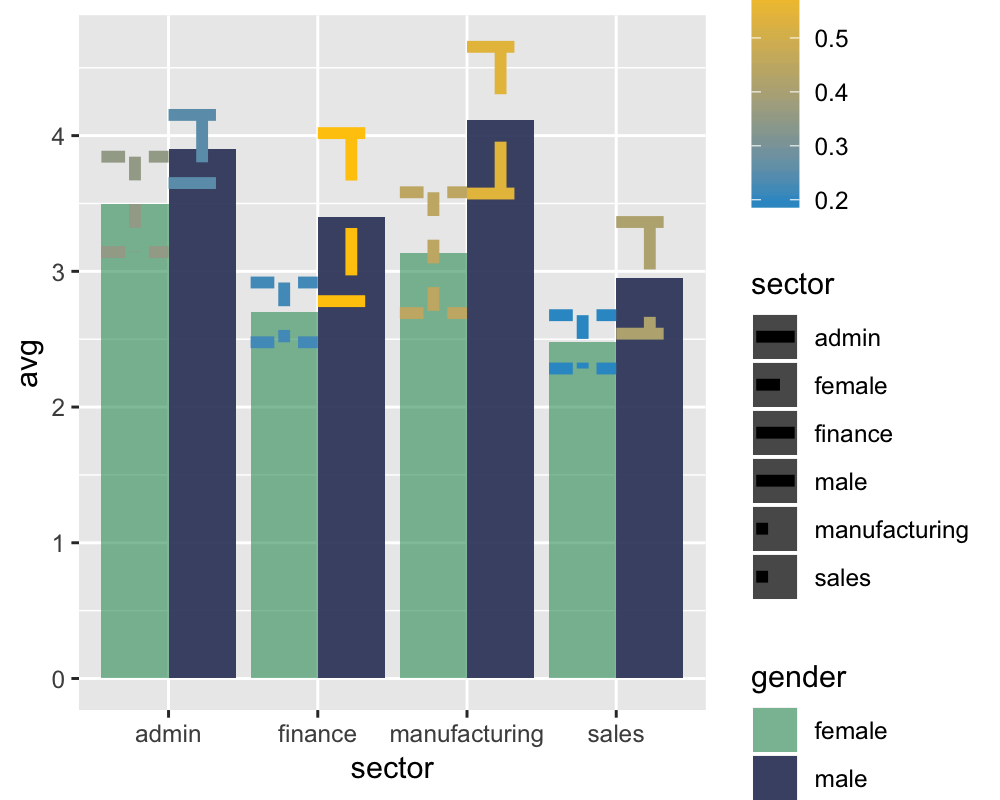


### Setting Colors, Shapes, and Sizes by Group

To set a graphical parameter by group (i.e to map the colour to something), you need to include the color mapping (outline or fill color) in the aes() argument like so:

**This one’s worse than before! Looks like there are some other arguments snuck in too – and a couple of scale objects! Play around and see what you do to make this look better!**

all\_data %>% #take all\_data THEN  
 group\_by(gender, sector) %>% #group it by sector THEN  
 summarise(avg = mean(q3),  
 n = n(),  
 se = sd(q3)/sqrt(n)) %>%  
 ggplot(aes(x = sector, y = avg)) +   
 geom\_bar(stat = "identity", position = "dodge",  
 aes(fill = gender, linetype = sector)) +  
 geom\_errorbar(aes(ymin = avg - se, ymax = avg + se,   
 linetype = gender, size = sector, color = se),  
 size = 2, position = "dodge") +  
 scale\_fill\_manual(values = c("#33996699", "#3a456aEE")) +   
 scale\_color\_continuous(low = "#3399CC", high = "#FFC80A")



### Themes and Finishing Touches

This is where the 80/20 principle meets plotting with ggplot - the possibilities for finishing touches are vast, so I will ony give a couple of quick examples here. As you get more experienced, they will become like second nature for you, but until then, *Google and Stack Exchange are going to be your friends*.

**Themes:** You can personalise a ggplot to the nth degree if you so choose. Mostly this is done by adding a theme() - since that’s a relatively advanced topic, we have loaded the ever useful ggthemes package which comes with some pre-made themes and color schemes ready to go.

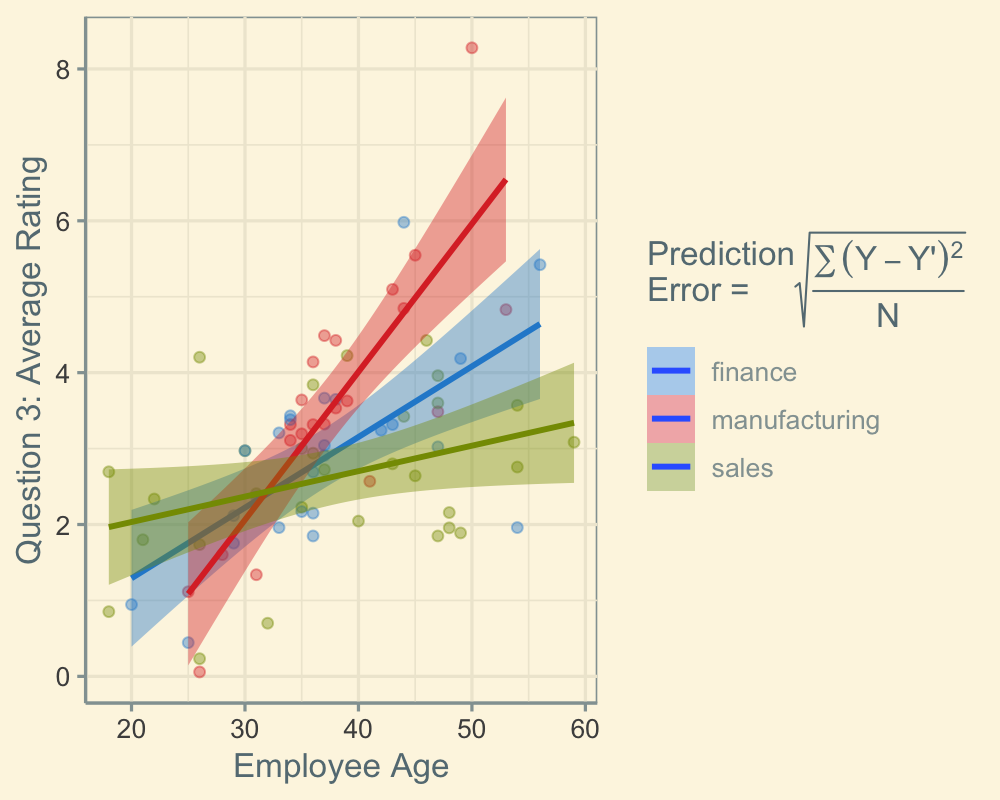
**Labels:** You can specify label names either in the scale\_x/y\_continious objects or by adding an xlab and/or a ylab

**Color schemes:** To specify a colour scheme, add a scale\_fill\_… or scale\_colour\_… to your plot (select based on the axis type). For manual discrete scales, set the colors by values = c(col1, col2, col3,...) in order of appearance.

Play around with the following code to understand what it is doing. If you can follow what this chunk of code is doing, you’re well on your way to mastering data wrangling and visualization in R!

*(hint - use R Studio’s code completion suggestions to your advantage)*

all\_data %>%  
 filter(sector %in% c("sales", "manufacturing", "finance")) %>%  
 ggplot(aes(x = age, y = q3, color = sector)) +  
 geom\_point(alpha = 0.4, aes(color = sector)) +  
 geom\_smooth(method = "lm", aes(fill = sector), na.rm = TRUE) +  
 theme\_solarized() +  
 ylab("Question 3: Average Rating") +  
 xlab("Employee Age") +  
 labs(fill = expression(paste("Prediction\nError = ",   
 sqrt(over(sum(("Y" - "Y'"))^2,"N"))))) +  
 scale\_fill\_solarized() +  
 scale\_color\_solarized(guide = FALSE)



Note that for the purposes of your own code, you should comment your code well so that other people can follow it and probably break super complicated code chunks into parts (including when piping) to make it easier to modify and edit.