Data: Wrangle & Display It With (Relative) Ease

Korn Ferry Institute

# Data Cleaning

In this tutorial, we will be taking in two data sets from a fictional company, one being data on the employees’ demographics and the other is some fictional test scores. We will be cleaning and reformatting them to make them usable, joining them together, and producing some graphics from them. Happily, all of this is actually pretty easy in R.

## 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:

pkgs <- c("openxlsx", "reshape2", "dplyr", "magrittr",   
 "ggplot2", "ggthemes", "roperators")  
  
# Hackfix tip:   
# Opening the CRAN mirror in a browser can help with some restricted networks  
utils::browseURL("http://cran.stat.auckland.ac.nz/")  
utils::browseURL("https://meta.wikimedia.org/wiki/List\_of\_countries\_by\_regional\_classification")  
  
# Another hackfix for restricted networks.... just in case  
httr::set\_config(httr::config( ssl\_verifypeer = 0L ))  
  
for (package in pkgs){  
 if(package %in% rownames(installed.packages()) == FALSE)  
 # Try will attempt to do what is in parentheses, but won't die if it doesn't work  
 try(install.packages(package, repos = "http://cran.stat.auckland.ac.nz/"), silent = TRUE)  
}

### Load your required packages

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

### Load The Data

In our data folder, there are two data sets, a csv called employee\_data.csv and an excel workbook called survey\_results.xlsx

To read in the csv data, we can use base R’s read.csv function, which is the same as read.table, which you might see in other scripts, only with different default arguments. What’s nice about data stored in csv files is that because they’re just plain-text flat files, they can be opened in any program or programming language.

**The following code reads:**

* Create a variable called employees
* Into that value place the output of read.csv()
* …Where read.csv() is going to go out one folder (../) and then look for employee\_data.csv in data
* And finally, don’t turn text variables into factors automatically!

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

Excel’s files are a little trickier to read in which is why we loaded the openxlsx package to handle it. There are other, older packages to read in .xlsx documents, however, they often have difficult Java dependencies, hence we prefer openxlsx

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

## Inspect The Data

Now that our data has been read into R as dataframes (which are just glorified lists of vectors that are all 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  
...

**Yeesh** it looks like some well-meaning but not-data-minded person went and stored the survey results in 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 text!
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 meaning in each row are from different variables hence the yearly column statistics are meaningless

## Fix The Data

### Fix missing values

Firstly, let’s replace those “..” missing value codes with missing values (NA). We also want to convert the age column into a numeric type. Think about the logic we want to use: we want to go through the age columns of employees turn all cells with .. into NA, and then convert the columns into numeric.

Happily, this is all rather easy,

**Note** In some cases, such as using gsub() text that contains dots, such as our ..s will require a *regular expression*. *Regular expressions* are a tricky subject, basically they’re specially formatted text used to select specific formats of text. You can use *regular expression*`s to do some very complicated text manipulation - but it’s more efficient to just look on <http://stackoverflow.com/questions/tagged/regex> for an existing example of something similar to what you’re wanting to do.

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

**The following code reads:**

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

employees$age[employees$age == ".."] <- NA  
employees$age <- as.numeric(employees$age)  
## Read as:   
## employee$age[where employees$age is equal to ".."] replace with NA  
## ....but we want to teach you new tricks!

### Fix dates

The hiring date column (employees$h\_date) is currently a character string AKA a text variable. We’ll want to make that into a proper datetime variable first so that we can do interesting stuff, 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 which just make some things a little more straightforward :)

In the following code, we need to tell R what format the date is currently in!

The code reads:

* Overwrite the employee\_data$h\_date column with the output of as.Date()
* Within as.Date(), transform the existing h\_date column
* …And read the 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 for

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 (turned into numeric from a difftime class)
* …then divide that 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:

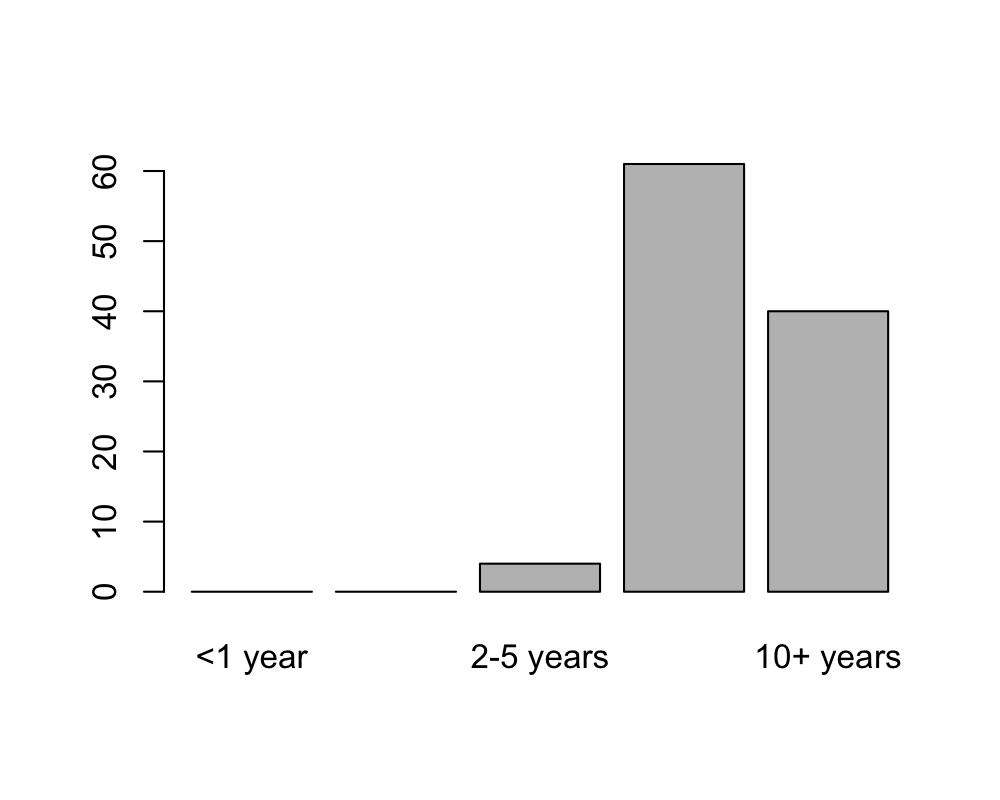
* the variable being transformed
* the break points (note 0 and Inf on the ends)
* the labels corresponding to the intervals between the breaks
* a logical flag telling it whether the breaks are including the number on the right of the interval. If FALSE it’s 0 to less than 1, 2 to less than 3; if TRUE it’s 0 to 1, over 1 to 2, over 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 the data

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

# A histogram of employee age  
# hist(employees$age) # if you want to look at the distribution of employee ages  
plot(employees$tenure\_label)



### Reshape the scores data

Things are looking promising! But we still have to do something about the scores dataframe. It’s be nice to be able to break down our scores by employee demographics, however, that isn’t so straight-forward given that the scores are in another dataframe that is in wide format with one column per candidate. Happily, that won’t take us long to fix! We’ll use a package called reshape2 to turn the dataframe into a nice long format (one row per candidate and one column per question/variable).

First, let’s melt the data into a long format (think of cheese melting off of a delicious pizza if it makes the function name more intuitive). 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 is the name of the column where the names of the melted columns will go
* the value.name is 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

Hmm, it looks like we’re getting there, however now the data format is a bit too long to be practical since the scores from each question are all in the same column. Let’s widen it up just a little bit to get the scores from each question into their own columns. To do this we use dcast() to **cast** the **d**ata (think of casting a fishing line).

To use dcast() we need to specify:

* the data we want to transform
* a formula that looks like: <row for observations> ~ <column to seperate out>

survey <- dcast(survey, id ~ question)

## Using score as value column: use value.var to override.

Now, let’s take a look-see…

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, one column per variable! Next up, let’s join the scores to the employee data by their ID number.

## Join the dataframes together

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 gross
* 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 that our data is all pretty like, we can do some aggregations!

## Make 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 that makes your code super easy to read. 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.

### dplyr verbs

Here are the main functions (verbs) in dplyr:

***filter*** *Get a subset of* rows\*

***select*** *Get a subset of* columns\*

\***group\_by** Tag data 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** Rename columns

\***arrange** Sort the data by selected columns

\***do** Do an arbitrary thing (advanced)

There’s a lot more to dplyr than these, but for the most part, these are all you really need to know.

Let’s try an 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 is in 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

That’s nice and easy, but what if we wanted to filter and then select? We could do something like this:

filtered\_data <- filter(all\_data,   
 (gender == "female" & sector == "manufacturing" & age > 40)|  
 (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, especially if we want to do a lot of stuff. Instead we can use pipes! What are pipes you ask? They are beauty incarnate.

## Use pipes (%>%) to make easy advanced aggregations!

magrittr is an amazing R package. It allows use to use 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 data from the previous function. 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" & sector == "manufacturing" & age > 40)|  
 (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 wanted to add some steps to that. Let’s say I want to aggregate that last output by gender and summarize the average scores for each question. With pipes, it’s easy, you just add more!

* 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" & sector == "manufacturing" & age > 40)|  
 (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

**Now spend 10-15 minutes playing around creating your own pipelines to filter, 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 :)

## Make pretty pictures 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 layered graphics with its own syntax commonly referred to as the Grammar of Graphics.

Code for ggplot2 can seem off putting at first glance - as such it’s important to 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

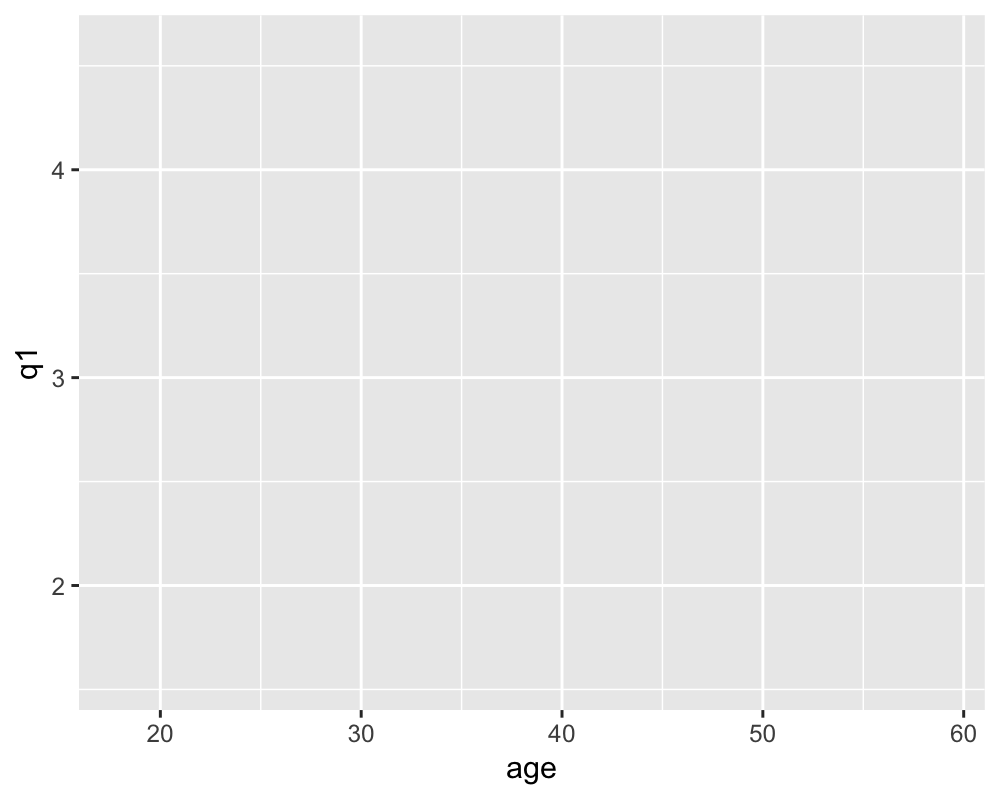
You can either create your plot directly or save it as an object, in the next few examples we will draw our plots directly to get some practice with the syntax (yay, repetition!)

### 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 have specified the axes, we said nothing about what to put on them, and are left with a blank plot. To add a graph to it, 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

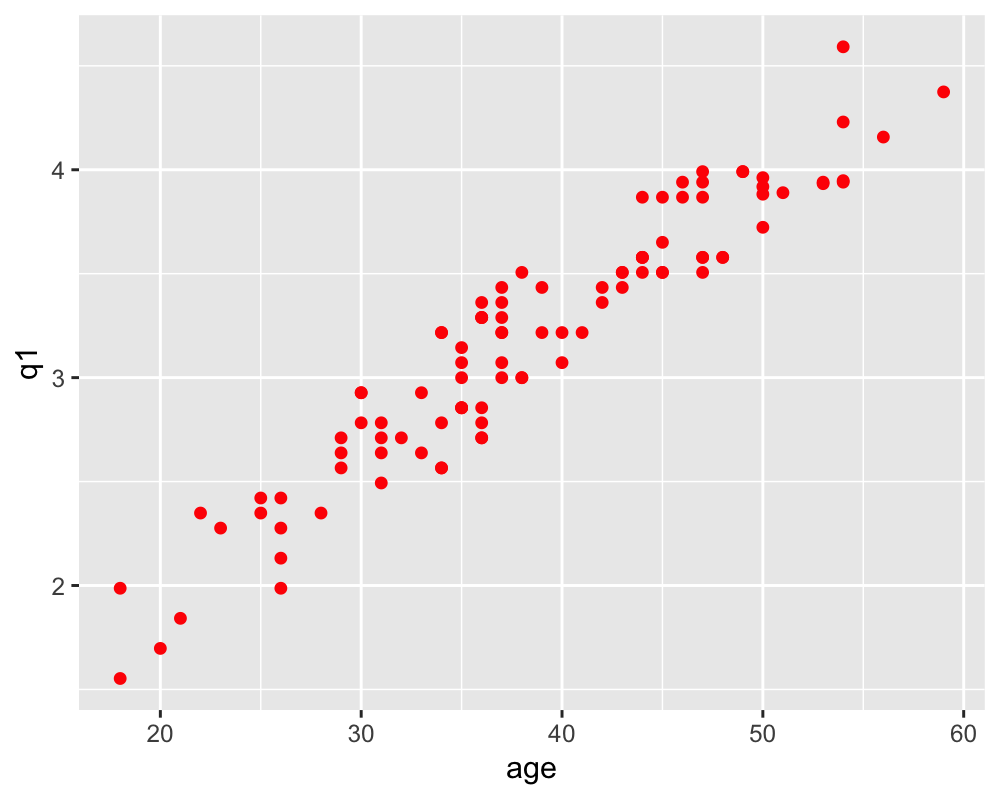
There are many different types of graph that can be drawn with ggplot2 - far more than we can cover in this tutorial. We will only be able to cover some of the fundamental chart types, however with the exception of multiple y axes and 3D barcharts\* if you can imagine it, it can be done. 3D plotting 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** inherrits the **aesthetic mapping** from the main ggplot object, however you can specify these individually as well.

### Scatter plots: geom\_point

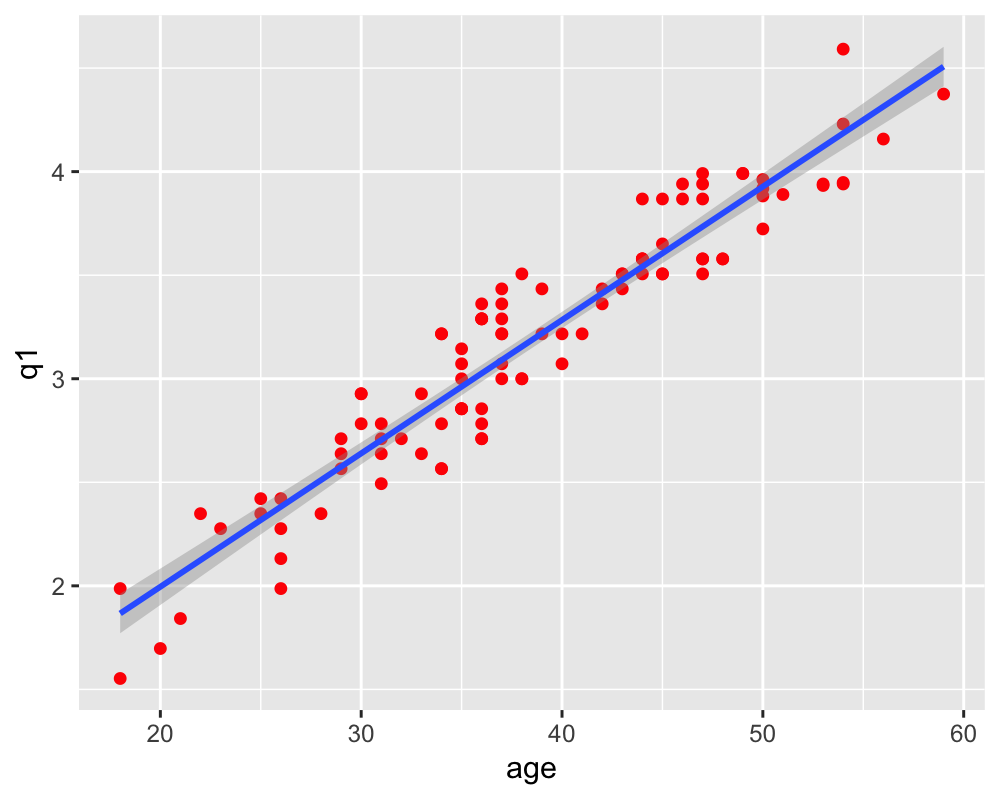
Let’s begin by adding a **layer** of red points (geom\_point) to our empty plot. To add layers in ggplot2 we just use + like so:

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



Adding a custom trendline in ggplot is mercifully flexible. By default, a LOESS smother is used, however 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



**Rember** there are many more arguments to use with geom\_smooth - if you need it done, it can be done.

### Bar charts: geom\_bar

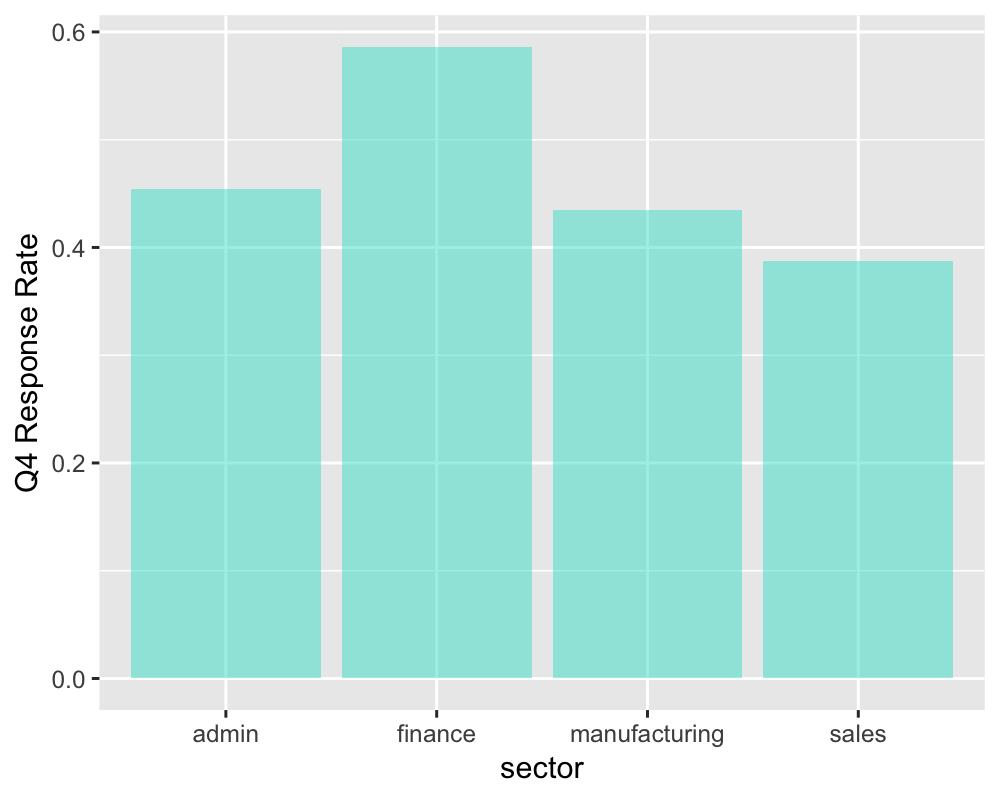
To create a bar chart, to make things easy\* you will need to add a geom\_bar object and must specify stat = "identity" (by default it wants to do a count, identity just means draw a specific value) and use a subset of data that you have pre-made.

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.

Let’s spice things up a little and pipe directly from dplyr into ggplot2. 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!

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  
 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

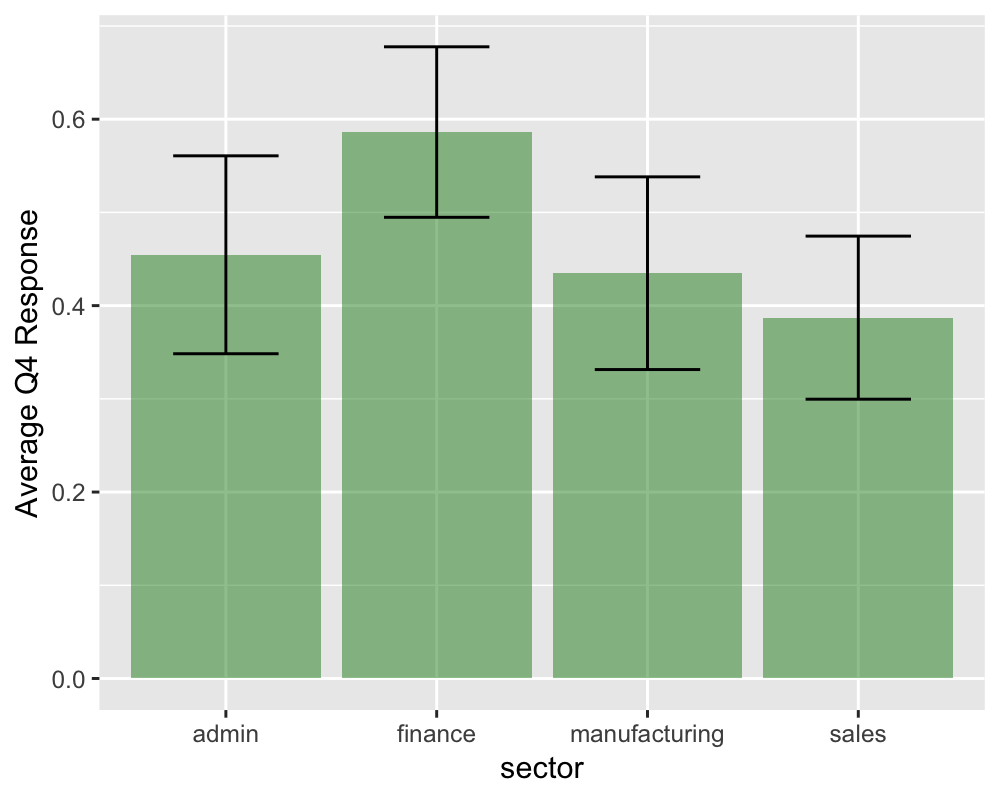


### Error bars: geom\_errorbar

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

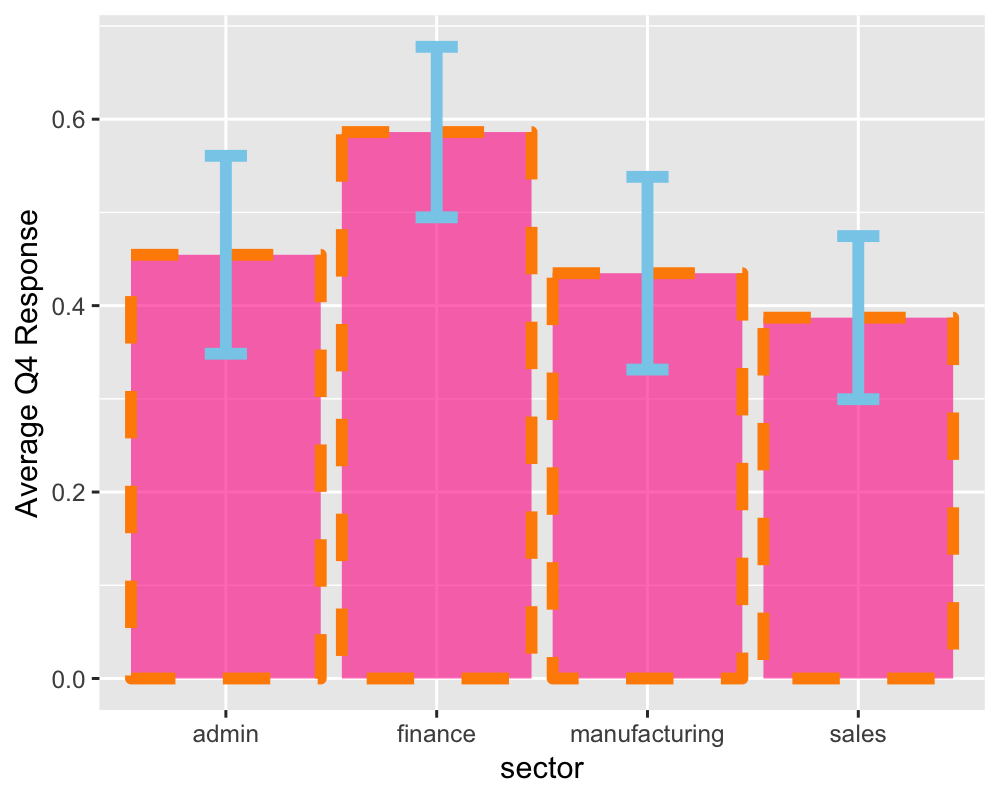
To apply a color, or specify the shape (for points), line type (for lines), and size (points and outlines) of an object, simply pass in the arguments:

* 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

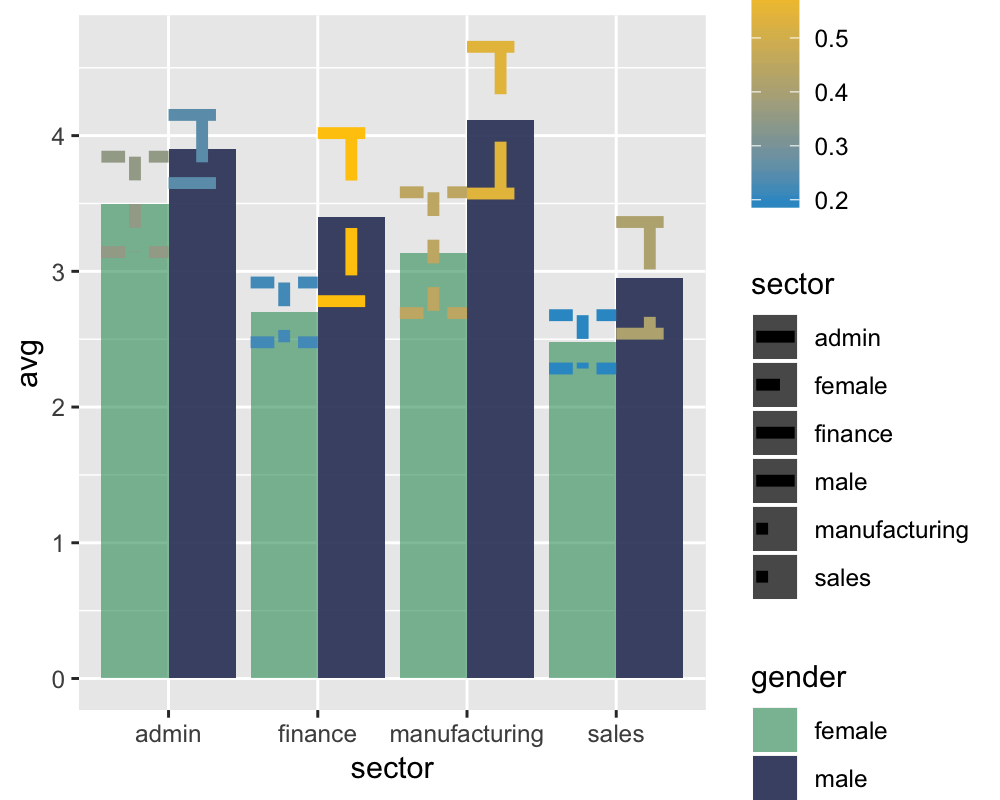


### Setting colors, shapes, and sizes by group

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:

**Eww, 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

**Colorschemes:** To specify a colour scheme, add a scale\_fil\_… 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 monster chunk of code is doing, you’re well on your way to mastering data wranglin 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)

