Loading Data and Feature Engineering
Workshop and Exercises

Contents

1 Introduction

In this applied homework assignment you will learn how to work with data in R and to visualize real data sets. You can find the homework questions at the end of this document.

2 Workshop Procedures

This workshop covers data manipulation and data visualization. Because data rarely arrives in a form that can be directly analyzed with different data science procedures. The first stages of data analysis are almost always figuring out how to load the data into R and then figuring out how to transform it into a shape you can readily analyze. The code in this workshop requires the *magrittr* and *ggplot2* packages.

To do this, you can use the *install.packages* command:

- *install.packages(magrittr)*
- *install.packages(ggplot2)*

Then, you should call them using the *library* command:

- *library(magrittr)*
- *library(ggplot2)*

For the datasets, you will want the package *mlbench* and the datasets package:

- *install.packages(mlbench)*
- *library(mlbench)*
- *library(datasets)*

3 Obtaining Data

There are two ways to obtain data in R. You can either import it or use one of the pre-loaded datasets.

3.1 Pre-Loaded Data

For instance, one of the datasets available in R is about cars. It is part of the *datasets* package, which once called will allow you to load datasets with the *data* command.

```
data(cars)
head(cars)
cars %>% qplot(speed, dist, data = .)
cars %>% head(3)
cars %>% tail(3)
cars %>% summary()
```

The $\% > \%$ is called an operator where the part on the left is the first command, calling the cars data, while the part on the right is the second command and applies to the data. This is called a pipeline as it allows you to do multiple commands at once.
We can also call a data set on breast cancer which gives an opportunity to understand the assignment code: $< -$. For instance, call the BreastCancer data and then assign it to a dataset so that it appears in the environment. R calls this a "dataframe".

```
data(BreastCancer)
BreastCancer %>% head(3)

breast_cancer <- BreastCancer
```

3.2 Reading In Data

It is also possible to load datasets into R with a read command. You should download the Breast Cancer dataset from Moodle. On my computer, the following command reads the data as a comma separated file. You will need the file path specific to your computer. You can find that initially with the import button in your environment on R. After you click the import button you need to find the file location on your computer where you stored the data.

```
breastcancer <- read.csv("~/Dropbox/Baker Classes/MBA542/
Week 1/BreastCancer.txt", header=FALSE)
```

4 Feature Engineering/Data Manipulation

4.1 Binary Variables

With binary variables, like whether a tumor is malignant or not, it is sometimes useful to go between numeric and character variables. The following function uses a nested if command to go from a character variable with values benign/malignant to a numeric variable with values 0/1:

```
map_class <- function(x) {
  ifelse(x == "benign", 0,
         ifelse(x == "malignant", 1,
                NA))
}
```

We can count the number of observations in each category. First assign the count to something called "mapped". After the assignment code, you can see R running the variable "Class" from the dataframe *breast_cancer* through the function *map_class*. Once "mapped" is created, you can put the counts in a table with the command below.

```
mapped <- breast_cancer$Class %>% map_class()

mapped %>% table
```

The nested if command is quite nice but can become cumbersome if there are too many different possible values. When numerous values are involved, it can be nice to create a dictionary - a table where R can look up words. The code is as follows:

```
dict <- c("benign" = "2", "malignant" = "4")
map_class <- function(x) dict[as.character(x)]

mapped <- breast_cancer$Class %>% map_class

mapped %>% table
mapped[1:5]
```

Here, the c() means concatenate the list inside the parentheses. So, we are creating a dictionary concatenated from the two different words inside the parentheses. Then, *map_class* becomes a function which applies the dictionary to the characters in a particular column of data. When we now assign to "mapped" the function *map_class* looks at the "Class" column of data inside the dataframe *breast_cancer* and applies the dictionary to it.

4.2 Housing Dataset

One of the famous datasets in data science has to do with Boston house prices. The dataset is pre-loaded into R and can be found in the mlbench library. The code below calls it into R.

```
library(mlbench)
data(BostonHousing)
str(BostonHousing)
```

The str() function gives you the internal structure of an object in R. Here, we are asking for the structure of a dataframe with the Boston housing data. One item of interest from this is that you can see what type of data each variable is (numeric, integer, factor, etc.). You should also practice loading this into R with a read command from a proper dataset. Again, the code will look different on your computer because of the file path. On mine, it looks like this:

```
boston_housing <- read.csv("~/Dropbox/Baker Classes/MBA542/
Week 1/boston_housing.csv")

str(boston_housing)
```

You should see that most variables are integer or numeric. However, in the data we loaded in from a csv file, the chas variable is numeric and it should be a factor. We can fix this with the following code:

```
col_classes <- rep("numeric", length(BostonHousing))
col_classes[which("chas" == names(BostonHousing))] <- "factor"

boston_housing <- read.csv("~/Dropbox/Baker Classes/MBA542/Week 1/boston_housing.csv",
                           col.names=names(BostonHousing),
                           colClasses = col_classes)
str(boston_housing)
```

Here, you use the same names for the pre-loaded data and you can use the class for the variable chas. Check with *str*(*boston_housing*) again and you should see that the chas variable is now a factor.

4.3 Iris Dataset

An important package with many useful commands is the dplyr package. We will practice these on the Iris dataset, another of the famous data science datasets.

```
library(dplyr)
iris <- iris
iris %>% tbl_df
```

Import the package with the *library*(*dplyr*) command and then assign the dataset iris so that it appears in your environment. The *tbl_df* command will print out a table that gives the first ten rows and some of the columns. It gives you a nice idea of the parameters of a dataframe with which you are working.

4.4 Commands

4.4.1 Select

It is sometimes useful to consider only the data of a particular column. You can do this with the select() command. If you want to look at multiple columns, you can include two or more inside the select command. The head(3) command you see gives you the first three rows in the dataset. This helps you to see that R is doing what you intend.

```
iris %>% tbl_df %>% select(Petal.Width) %>% head(3)

iris %>% tbl_df %>% select(Petal.Width, Petal.Length) %>% head(3)

iris %>% tbl_df %>% select(Sepal.Length:Petal.Length) %>% head(3)

iris %>% tbl_df %>% select(starts_with("Petal")) %>% head(3)

iris %>% tbl_df %>% select(ends_with("Width")) %>% head(3)

iris %>% tbl_df %>% select(contains("etal")) %>% head(3)

iris %>% tbl_df %>% select(matches(".t.")) %>% head(3)

iris %>% tbl_df %>% select(-starts_with("Petal")) %>% head(3)
```

4.4.2 Mutate

Another useful data engineering tool is the mutate() command. This allows you to take an existing dataframe and create a new column in that dataframe. For instance, the code below creates a new variable which is the sum of the petal width and the petal length.

```
iris %>% tbl_df %>%
  mutate(Petal.Width.plus.Length = Petal.Width + Petal.Length) %>%
  select(Species, Petal.Width.plus.Length) %>%
  head(3)

iris %>% tbl_df %>%
  mutate(Petal.Width.plus.Length = Petal.Width + Petal.Length,
         Sepal.Width.plus.Length = Sepal.Width + Sepal.Length) %>%
  select(Species, Petal.Width.plus.Length, Sepal.Width.plus.Length) %>%
  head(3)
```

If you want to create a new dataframe out of the data engineering process, you can use the transmute command. This works like the mutate() command, except it is combined with select() so that the result is a dataframe that only contains the new columns which you make.

```
iris %>% tbl_df %>%
  transmute(Petal.Width.plus.Length = Petal.Width + Petal.Length) %>%
  head(3)
```

4.4.3 Arrange

It is often quite useful to order the data in some ascending or descending fashion according to a particular column. The arrange() command facilitates this. You can choose to order using multiple columns by including more than one column name inside the parentheses.

```
iris %>% tbl_df %>%
  arrange(Sepal.Length) %>%
  head(3)

iris %>% tbl_df %>%
  arrange(desc(Sepal.Length)) %>%
  head(3)
```

4.4.4 Filter

If you want to consider rows which meet certain criteria, the filter function can be very useful. The filter() function will allow you to choose rows based on certain logical expressions. You give the function a predicate, specifying what a row should satisfy to be included. Some examples are included below.

```
iris %>% tbl_df %>%
  filter(Sepal.Length > 5) %>%
  head(3)

iris %>% tbl_df %>%
  filter(Sepal.Length > 5 & Species == "virginica") %>%
  select(Species, Sepal.Length) %>%
  head(3)
```

4.4.5 Group By

Perhaps one of the most useful commands is *group_by*. By itself, it is not particularly useful. However, combined with other commands, it can become an integral part of data analysis. For instance, it can be particularly helpful in calculating summary statistics of interest when you want to segment the dataset.

```
iris %>% tbl_df %>% group_by(Species) %>% head(3)
```

4.4.6 Summarise

We can practice the *group_by* command with summarize. This if we ask summarize to give us the mean, for instance, we could obtain summary statistics by species of iris. First, calculate the mean petal length and the mean sepal length. Then, calculate the mean petal length by species.

```
iris %>% summarise(Mean.Petal.Length = mean(Petal.Length),
                   Mean.Sepal.Length = mean(Sepal.Length))
```

```
iris %>%
  group_by(Species) %>%
  summarise(Mean.Petal.Length = mean(Petal.Length))

iris %>%
  summarise(Observations = n())

iris %>%
  group_by(Species) %>%
  summarise(Number.Of.Species = n())

iris %>%
  group_by(Species) %>%
  summarise(Number.Of.Species = n(),
            Mean.Petal.Length = mean(Petal.Length))
```

5 Basic Graphics

First, call in the graphics package. This should exist already in your version of RStudio, but it is helpful to be mindful of the packages that you are using. The help command included here gives you a list of commands that are available with the graphics package. We will first focus on plot().

```
library(graphics)
library(help = "graphics")
```

In order to plot something we need data to plot. We will plot in two dimensions and want to think about something for the x-axis and the y-axis. The rnorm command takes the specified number of data points (50 in this case) and distributes them normally with mean zero and standard deviation 1.

```
x <- rnorm(50)
y <- rnorm(50)
plot(x, y)
```

If you wanted to use another of the functions in the graphics package, you could try:

```
hist(x)
or
hist(y)
```

The plot() function takes a data argument you can use to plot data from a data frame, but you cannot write code like this to plot the cars data from the datasets package. Instead, you will need to do something like:

```
plot(cars$speed, cars$dist)
```

This gives the title for the x and y-axis as the characters inside the parentheses. We can instead choose the label that appears on the axes in the following manner:

```
plot(cars$speed, cars$dist, xlab = "Speed", ylab = "Stopping
distance")
```

Another option for plotting is to give R a formula command inside the plot() function. For instance, when you want to compare one variable with another, you use the tilde sign:

```
cars %>% plot(dist ~ speed, data = .)
or
plot(cars$dist ~ cars$speed)
```

If you want to use the histogram function here, you could do so with the following code (Note: the %$% operator is found in the magrittr package):

```
cars %$% hist(speed)
which is equivalent to
hist(cars$speed, xlab = "speed")
```

An extension of plotting is the idea of linear regression, which we will cover in more detail in week 2. You will see several plots visualizing the quality of the linear fit.

5.1 Plotting with Time

Often it is desirable to plot a variable with time on the x-axis. Let's use the Longley dataset to illustrate this. Call the data:

```
data(longley)
?longley
```

Using *?longley* will give you a description of the data in R.
The code below will put two lines on a single plot. First, plot the number of unemployed people across time. Then add the line for the number of people in the armed forces distinguishing that line with the color blue.

```
longley %>% plot(Unemployed ~ Year, data = ., type = 'l')
longley %>% lines(Armed.Forces ~ Year, data = ., col = "blue")
```

One difficulty is that the limits on the y-axis are set by the unemployed variable. You can see that the armed forces variable goes lower so we need to set the limits manually. The following code does this:

```
longley %$% plot(Unemployed ~ Year, type = ‘l’, ylim =
range(c(Unemployed, Armed.Forces)))

longley %>% lines(Armed.Forces ~ Year, data = ., col = "blue")
```

If you receive an error message for the pipe command, make sure you run library(magrittr). An extension of this is to fit a line to the data which is defined by some model. The abline() command does this by using a line of the form $y = a + bx$. For instance:

```
cars %>% plot(dist~speed, data = .)
cars %>% lm(dist ~speed, data = .) %>% abline(col = "red")
```

5.2 Plotting by Group

Often it is very nice to visualize plots where types of data are grouped in some way (color or shape, for instance). We can see this particularly well with the iris dataset. First, we can create list which distinguishes among the categories:

```
shape_map <- c("setosa" = 1, versicolor" = 2, "virginica" = 3)
```

Now, we want to plot the petal length and petal width with these three categories grouped by shape. We can do this with the following code:

```
iris %$% plot(Petal.Length ~ Petal.Width, pch =
shape_map[Species])
```

5.3 Plotting with ggplot2

We can go beyond the basic plots and do slightly more sophisticated graphical work with ggplot2. First, run the library command to bring the package into your library. If it is your first time using this package, remember that you will need to run install.packages(“ggplot2”).

```
library(ggplot2)
library(help = "ggplot2")
```

Let’s first review a few plots we have already made:

```
cars %>% qplot(speed, dist, data = .)
```

With the ggplot package, you can easily save plots and assign them to an object in the R environment. For instance:

```
p <- cars %>% qplot(speed, dist, data = .)
p
which is equivalent to
p <- cars %>% qplot(speed, dist, data = .)
print(p)
```

Other variations of plots in the cars dataset that we have seen before:

```
cars %>% qplot(speed, data = ., bins = 10)
```

And, we have the ability to use geoms which allows us to specify certain geometric objects.

```
cars %>% qplot(speed, data = ., geom = "density")
```

5.4 Using Geoms

By stringing together several geometry commands, you can display the same data in different ways. For instance, we might combine scatterplots with smoothed lines. Start with the scatterplot.

```
cars %>% qplot(speed, dist, data = .)
```

Now, we need a ggplot object such as ggplot(cars) and then can add the geometric objects. It is possible to add multiple geoms.

```
ggplot(cars) +
geom_point(aes(x = speed, y = dist))

ggplot(cars) +
geom_point(aes(x = speed, y = dist))+
geom_smooth(aes(x = speed, y = dist))
```

It is also possible to write this specifying the aesthetic inside the ggplot command.

```
ggplot(cars, aes(x = speed, y = dist)) +
geom_point()+
geom_smooth()
```

Earlier, we organized the petal length and petal width by shape. With ggplot, it is quite easy to organize by color with a legend.

```
iris %>% qplot(Petal.Width, Petal.Length, color = Species, data = .)

iris %>% ggplot +
geom_point(aes(x = Petal.Width, y = Petal.Length,
color = Species))

iris %>% ggplot +
geom_point(aes(x = Petal.Width, y = Petal.Length),
color = "red")
```

We can also re-write the qplot code with typical ggplot code. Below gives two examples:

```
cars %>% qplot(speed, data = ., bins = 10)
cars %>% qplot(speed, data = ., geom = "density")

cars %>% ggplot +
geom_histogram(aes(x = speed), bins = 10)

cars %>% ggplot +
geom_density(aes(x = speed))

cars %>% ggplot(aes(x = speed, y = ..count..)) +
geom_histogram(bins = 10) +
geom_density()
```

We can also do more intricate time graphs. The following expands on the Longley data plots we did earlier:

```
longley %>% ggplot(aes(x = Year)) +
geom_line(aes(y = Unemployed)) +
geom_line(aes(y = Armed.Forces), color = "blue")

longley %>% ggplot(aes(x = Year)) +
geom_point(aes(y = Unemployed))+
geom_point(aes(y = Armed.Forces), color = "blue")+
geom_line(aes(y = Unemployed)) +
geom_line(aes(y = Armed.Forces), color = "blue")

longly %>% gather(key, value, Unemployed, Armed.Forces) %>%
 ggplot(aes(x = Year, y = value, color= key)) + geom_line()

longley %>% gather(key, value, Unemployed, Armed.Forces) %>%
 ggplot(aes(x = Year, y = value)) + geom_line() +
 facet_grid(key ~ .)
```

5.5 Faceting

The last code above included a facet command. Faceting is very useful when you want to display data in particular ways that might emphasize subsets of the data. For instance, we can see this with the iris data. We can plot the four measurements for each species in different facets, but they are on slightly different scales, so we will only get a good look at the range of values for the largest range (note: to use gather, you will need to call the tidyr package).

```
iris %>% gather(Measurement, Value, -Species) %>%
ggplot(aes(x = Species, y = Value)) +
geom_boxplot() +
facet_grid(Measurement ~ .)
```

You can see that the scale on the y-axis is uniform across the four facets and might be better with tailored scales. This can be done by letting the y-scale 'range free'.

```
iris %>% gather(Measurement, Value, -Species) %>%
ggplot(aes(x = Species, y = Value)) +
geom_boxplot() +
facet_grid(Measurement ~ ., scale = "free_y")
```

Again, you can see that there are values on the y-axis are not appropriate for the scale. You can shift this with with the free y. Additionally, you can change the labels on the facets using a label map as follows:

```
label_map <- c(Petal.Width = "Petal Width",
Petal.Length = "Petal Length",
Sepal.Width = "Sepal Width",
Sepal.Length = "Sepal Length")

iris %>% gather(Measurement, Value, -Species) %>%
ggplot(aes(x = Species, y = Value)) +
geom_boxplot() +
facet_grid(Measurement ~ ., scale = "free_y",
labeller = labeller(Measurement = label_map))
```

5.6 Scaling

Geometries specify part of how data should be visualized and scales another. The geometries tell ggplot2 how you want your data mapped to visual components, like points or densities, and scales tell ggplot2 how dimensions should be visualized. The simples scales to think about are the x- and y-axes, where values are mapped to positions on the plot as you are familiar with, but scales also apply to visual properties such as colors.

The simplest way to use scales is to put labels on the axes. You can also do this using the xlab() and ylab() functions, and if setting labels were all you were interested in, you could do this. However, in this example, you see a different use of scales. To set the labels in the cars scatterplot, you write:

```
cars %>% ggplot(aes(x = speed, y = dist)) +
geom_point() +
geom_smooth(method = "lm") +
scale_x_continuous("Speed")+
scale_y_continuous("Stopping Distance")
```

In general, you can use the scale x or y continuous() functions to control the axis graphics. For instance, if you wanted to plot the longley data with a tickmark for every year instead of every five years, you could set the breakpoints to every year:

```
longley %>% gather(key, value, Unemployed, Armed.Forces) %>%
ggplot(aes(x = Year, y = value)) +
geom_line() +
scale_x_continuous(breaks = 1947:1962)+
facet_grid(key~.)
```

6 Exercises

For each of these exercises, use the Boston Housing dataset. Recall that you can obtain the Boston Housing data with the data(BostonHousing) command. This dataset lives in the mlbench package, so you will first need to run library(mlbench)

Question 1. Find the mean crime rate by Charles Rivers number. Report these two numbers in your homework.

Question 2. List the 5 tax observations if you arrange the Boston Housing data in descending order by the tax variable.

Question 3. Plot the pupil teacher ratio against the proportion of non-retail business acres per town. Do you see a pattern? Include your plot in your homework submission.

Question 4. Obtain an age distribution histogram. Does this seem like a representative sample? Include your plot in your homework submission.

Question 5. Create a plot using different shapes for the rad variable. Put the average number of rooms on the x-axis and the median home value in thousands on the y-axis. Use only four shapes. Include your plot in your homework submission.

Question 6. Create a scatterplot with ggplot. Put the average number of rooms on the x-axis and the median home value in thousands on the y-axis. Use age for color. What do you notice in this plot that is different from the plots in the worksheet? Include your plot in your homework submission.

Question 7. Create a scatterplot with ggplot and facet on the rad variable. Put the average number of rooms on the x-axis and the median home value on the y-axis. Include your plot in your homework submission.

Question 8. Create a plot of your choice and explain why it creates insight into the Boston Housing dataset. Include your plot in your homework submission.