Introductory Data Munging in R

Ken Oung Yong Quan

23 July 2016

Introductory Data Munging in R

First things first: RStudio

First off, we'll run through the installation of RStudio for those of you who have yet to download and install. You'll be needing R, which you can find here:

Click here to download R

Next, you'll need RStudio, which is a GUI for using R:

Click here to download RStudio

R is a language used mainly for statistical analysis. It is easily extendible through packages that can be downloaded and installed from the R-repository. To access this repository and install pacakges, use the `install.packages("Package Name")` function.

For today, you will be needing the following packages:

```
install.packages(data.table) install.packages(car)
```

Another package that is useful to have is the swirl tutorial package:

```
install.packages(swirl)
```

This package includes tutorials into the different classes and functions of R, and will be a useful reference for getting started with R. However, we will not be covering this, snce you can slowly explore this in your free time. While useful, going through the Swirl tutorials can be time consuming.

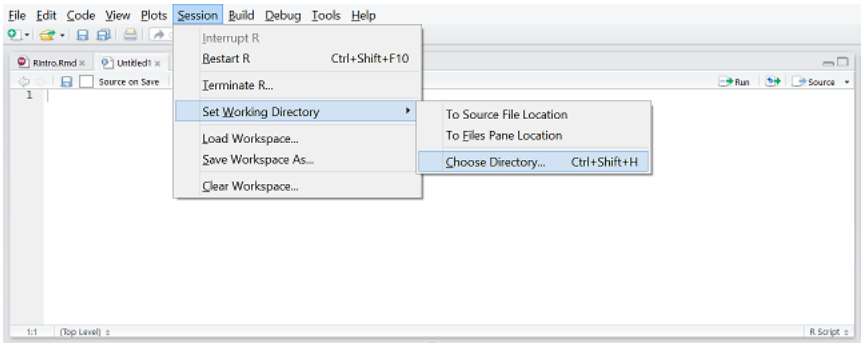
If at any point in time you are confused about a function in R, you can use the command `?<function>` to find out more about the function. For example:

```
?install.packages
```

Importing Data

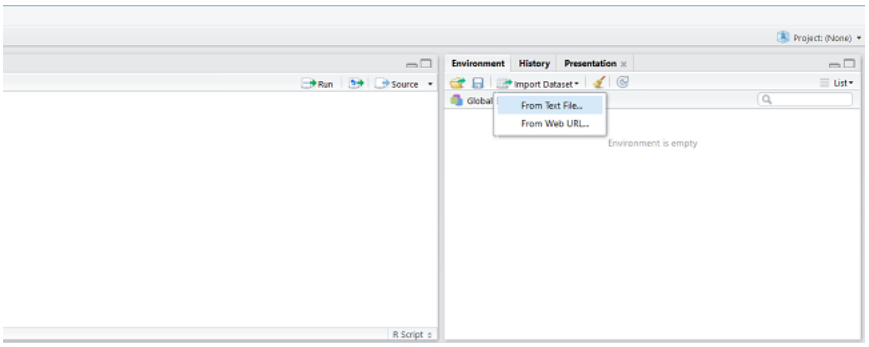
Setting your working directory in RStudio

We'll be using a DMEF dataset, which is mostly online shopping data (some phone and mail transactions too). Let's import the data first. If you're using RStudio, both navigating to the working directory and importing the file can be performed using the GUI, and the code will be generated for you.



Importing a file using RStudio

The read.csv function takes in a csv file and imports it into R. If the file is in your working directory, you will only need the name of the file. If you are trying to import the file from another folder, as shown below, you will need to specify the file path.



The '<-' operator is used to store data in a variable.

```
# Navigating to your working directory (GUI)
setwd("C:/Users/Ken/Dropbox/Academics/2016/Sem2/R Intro/bza-r-intro/code")
#setwd("~/R Intro/bza-r-intro/code")

# Loading the file (GUI)
dmef_dataset <- read.csv("C:/Users/Ken/Dropbox/Academics/2016/Sem2/R Intro/bza-r-intro/data/dmef_dataset
#dmef_dataset <- read.csv("~/R Intro/dmef_dataset.csv")
```

So the code above loads the csv data and saved it to the 'dmef_dataset' variable.

Getting A Feel

You can take a look at the entire dataset using the View command, `View(dmef_dataset)`, get a simple summary using the summary command, `summary(dmef_dataset)`, or see the types of data being stored using the str command, `str(dmef_dataset)`.

Let's just use the summary command to see the descriptive statistics for this dataset.

```
summary(dmef_dataset)
```

```
##      CUSTNO              ZIP           ORDER_LINE
##  Min.   :9.991e+09   10021  :   337   Min.   :    1.000
##  1st Qu.:1.008e+10   10128  :   270   1st Qu.:    1.000
##  Median :1.009e+10   08540  :   258   Median :    1.000
##  Mean   :1.009e+10   10011  :   229   Mean   :    5.107
##  3rd Qu.:1.011e+10   10024  :   224   3rd Qu.:    2.000
##  Max.   :1.013e+10   77024  :   214   Max.   :11012.000
##                      (Other):224597
##     ORDER_NO           PRODUCT_NO       RETN_LINE
##  Min.   :123456799   Min.   :920190   Min.   :   1.00
##  1st Qu.:125942853   1st Qu.:982083   1st Qu.:   1.00
##  Median :128264074   Median :984215   Median :   1.00
##  Mean   :127978114   Mean   :982891   Mean   :  19.72
##  3rd Qu.:129181758   3rd Qu.:987324   3rd Qu.:   1.00
```

```
##  Max.   :132481165   Max.   :993299   Max.   :2008.00
##                                       NA's   :215387
##        BO_DATE           CANCEL_DATE     CANCEL_QUANTITY
##            :208630             :217885   Min.   : 0.00000
##  12/13/2007:   115   5/25/2005 :    71   1st Qu.: 0.00000
##  12/12/2007:   107   12/12/2007:    53   Median : 0.00000
##  12/4/2007 :   106   12/29/2008:    52   Mean   : 0.03931
##  12/10/2007:   101   3/8/2010  :    47   3rd Qu.: 0.00000
##  12/5/2007 :   101   9/10/2008 :    41   Max.   :70.00000
##  (Other)   : 16969   (Other)   :  7980
##  PRODUCT_CATEGORY_ID CHANNEL      DIVISION_ID       OFFER_ID
##  T      :33907       ML:  8112   Min.   :1.000   W08WEB :  9168
##  C      :33871       PH: 90981   1st Qu.:1.000   W07WEB :  9120
##  P      :32683       WE:127036   Median :1.000   W06WEB :  7471
##  E      :31469                   Mean   :2.526   W09WEB :  7044
##  H      :19358                   3rd Qu.:5.000   W05WEB :  6820
##  F      :19346                   Max.   :5.000   W10WEB :  6036
##  (Other):55495                                   (Other):180470
##       ORDER_DATE        EXT_COST         EXT_PRICE         PAY_METHOD
##  4/12/2007 :   622   Min.   :   0.00   Min.   :   0.00   VI     :102103
##  10/12/2007:   610   1st Qu.:  14.03   1st Qu.:  34.95   MC     : 58741
##  7/12/2010 :   609   Median :  24.00   Median :  59.95   AX     : 42159
##  6/12/2005 :   608   Mean   :  39.05   Mean   :  84.21   DI     : 11759
##  4/12/2006 :   604   3rd Qu.:  45.00   3rd Qu.:  99.95   PC     :  6717
##  12/12/2011:   599   Max.   :3100.00   Max.   :5000.00   PY     :  2465
##  (Other)   :222477                                       (Other):  2185
##     QUANTITY           SHIP_DATE      SHIP_QUANTITY
##  Min.   : 1.000             :  8470   Min.   : 0.0000
##  1st Qu.: 1.000   11/12/2007:   697   1st Qu.: 1.0000
##  Median : 1.000   7/12/2011 :   596   Median : 1.0000
##  Mean   : 1.037   17/12/2008:   595   Mean   : 0.9979
##  3rd Qu.: 1.000   16/12/2008:   588   3rd Qu.: 1.0000
##  Max.   :70.000   10/12/2007:   584   Max.   :16.0000
##                   (Other)   :214599
##       RETN_DATE         RETN_QTY        RETN_REVENUE
##  0          :215387   Min.   :0.00000   Min.   :   0.000
##  1/7/2011   :    38   1st Qu.:0.00000   1st Qu.:   0.000
##  12/5/2008  :    34   Median :0.00000   Median :   0.000
##  1/21/2008  :    29   Mean   :0.04833   Mean   :   5.172
##  12/29/2010:    29   3rd Qu.:0.00000   3rd Qu.:   0.000
##  1/6/2010   :    28   Max.   :8.00000   Max.   :4500.000
##  (Other)    : 10584
```

Before we go on...

The data that you have currently imported is in a data.frame format. This is the most primitive and basic class for storing data in R. To check this, you can use the class function:

```
class(dmef_dataset)
```

```
## [1] "data.frame"
```

Personally, I like using the **data.table** library. This is an extension of the data.frame class and allows you to manipulate the dataset more easily. You have other options like the standard in-built **data.frame** object, or **dplyr** which works better with ggplot2, but data.table is the one we'll be working with today.

```
# Firstly, let's load the data.table package that you installed earlier.
library(data.table)

# Making the dataframe a datatable
dmef_dataset <- as.data.table(dmef_dataset)
```

Now, you can check the class of the dataset:

```
class(dmef_dataset)
```

```
## [1] "data.table" "data.frame"
```

As you can see, the data set now has the class "data.table", but it is still a data frame. This is because data table is only an extension of the data frame class. We won't go into too much specifics at this point, so let's continue.

WTF so many columns

If you've taken a look at the full data set, you might think it's a little large to handle. So let's clean up and filter the data. This is where the data table comes in useful. Let's run through some of the basic functions of a data table that might help.

The data table function takes the general form of data.table(x,y,z), where x denotes the subset of rows, y denotes the subset of columns, and z denotes the grouping criteria for the resultant column. Let's say we want to look at rows 3 to 5

Sifting through rows

```
dmef_dataset[3:5,]
```

```
##         CUSTNO   ZIP ORDER_LINE  ORDER_NO PRODUCT_NO RETN_LINE BO_DATE
## 1: 10109980365 76443          3 123456799     989457        NA
## 2: 10120142886 89103          1 123456815     986782        NA
## 3: 10056184374 28655          1 123457056     986620        NA
##    CANCEL_DATE CANCEL_QUANTITY PRODUCT_CATEGORY_ID CHANNEL DIVISION_ID
## 1:                          0                  C      ML           1
## 2:                          0                  E      ML           1
## 3:                          0                  E      ML           1
##    OFFER_ID ORDER_DATE EXT_COST EXT_PRICE PAY_METHOD QUANTITY  SHIP_DATE
## 1:   A10LCB 19/12/2010     28.0     69.95         PC        1 20/12/2010
## 2:   A10LMB 21/12/2010    227.0    349.95         MO        1 21/12/2010
## 3:   A11LSB  18/4/2011    105.8    249.95         MO        1  18/4/2011
##    SHIP_QUANTITY RETN_DATE RETN_QTY RETN_REVENUE
## 1:             1         0        0            0
## 2:             1         0        0            0
## 3:             1         0        0            0
```

But that doesn't really help with data analysis, since you won't know which rows you are interested in, and you definitely do not want to search through all those rows. Luckily, R can also do that for you. Data tables allow for sorting of values based on the row. Let's try that now. For this, we can for example find the rows that contain a certain product...

```
# To keep it short, let's use the head function to look at the first five entries.
head(dmef_dataset[PRODUCT_NO %in% 987668])
```

```
##         CUSTNO   ZIP ORDER_LINE  ORDER_NO PRODUCT_NO RETN_LINE BO_DATE
## 1: 10109980365 76443          1 123456799     987668        NA
## 2: 10109980365 76443          2 123456799     987668        NA
## 3: 10109086080 63146          2 127601151     987668        NA
## 4: 10109086080 63146          2 127617169     987668        NA
## 5: 10109585478 80218          1 127637289     987668         1
## 6: 10110046231 40222          1 127670778     987668        NA
##    CANCEL_DATE CANCEL_QUANTITY PRODUCT_CATEGORY_ID CHANNEL DIVISION_ID
## 1:                          0                  C      ML           1
## 2:                          0                  C      ML           1
## 3:                          0                  C      WE           5
## 4:                          0                  C      WE           5
## 5:                          0                  C      WE           1
## 6:                          0                  C      WE           5
##    OFFER_ID ORDER_DATE EXT_COST EXT_PRICE PAY_METHOD QUANTITY  SHIP_DATE
## 1:   A10LCB 19/12/2010    25.99     79.95         PC        1 20/12/2010
## 2:   A10LCB 19/12/2010    25.99     79.95         PC        1 20/12/2010
## 3:   W09SHP  8/11/2009    15.50     79.95         DI        1  9/11/2009
## 4:   W09SHP 13/11/2009    15.50     79.95         DI        1 16/11/2009
## 5:   A09LMB 19/11/2009    15.50     79.95         AX        1 20/11/2009
## 6:   W09WEB 27/11/2009    15.50     79.95         AX        1 28/11/2009
##    SHIP_QUANTITY RETN_DATE RETN_QTY RETN_REVENUE
## 1:             1         0        0         0.00
## 2:             1         0        0         0.00
## 3:             1         0        0         0.00
## 4:             1         0        0         0.00
## 5:             1 12/4/2009        1        79.95
## 6:             1         0        0         0.00
```

Or maybe we want to look at a few products at a time. To do this, we will create a vector of the products we want using the `c()` function.

```
head(dmef_dataset[PRODUCT_NO %in% c(987668, 989457)])
```

```
##         CUSTNO   ZIP ORDER_LINE  ORDER_NO PRODUCT_NO RETN_LINE BO_DATE
## 1: 10109980365 76443          1 123456799     987668        NA
## 2: 10109980365 76443          2 123456799     987668        NA
## 3: 10109980365 76443          3 123456799     989457        NA
## 4: 10109086080 63146          2 127601151     987668        NA
## 5: 10109086080 63146          2 127617169     987668        NA
## 6: 10109585478 80218          1 127637289     987668         1
##    CANCEL_DATE CANCEL_QUANTITY PRODUCT_CATEGORY_ID CHANNEL DIVISION_ID
## 1:                          0                  C      ML           1
## 2:                          0                  C      ML           1
## 3:                          0                  C      ML           1
```

```
## 4:                           0                  C      WE           5
## 5:                           0                  C      WE           5
## 6:                           0                  C      WE           1
##    OFFER_ID ORDER_DATE EXT_COST EXT_PRICE PAY_METHOD QUANTITY  SHIP_DATE
## 1:   A10LCB 19/12/2010    25.99     79.95         PC        1 20/12/2010
## 2:   A10LCB 19/12/2010    25.99     79.95         PC        1 20/12/2010
## 3:   A10LCB 19/12/2010    28.00     69.95         PC        1 20/12/2010
## 4:   W09SHP  8/11/2009    15.50     79.95         DI        1  9/11/2009
## 5:   W09SHP 13/11/2009    15.50     79.95         DI        1 16/11/2009
## 6:   A09LMB 19/11/2009    15.50     79.95         AX        1 20/11/2009
##    SHIP_QUANTITY RETN_DATE RETN_QTY RETN_REVENUE
## 1:             1         0        0         0.00
## 2:             1         0        0         0.00
## 3:             1         0        0         0.00
## 4:             1         0        0         0.00
## 5:             1         0        0         0.00
## 6:             1 12/4/2009        1        79.95
```

The first variable in the data.table function can take in any argument. We can include filters to check for specific values, as we have done above, but we can also filter out values outside of a certain range. For example:

```
head(dmef_dataset[EXT_PRICE < 30])
```

```
##         CUSTNO   ZIP ORDER_LINE  ORDER_NO PRODUCT_NO RETN_LINE BO_DATE
## 1: 10126696008 05353          2 123457640     989637        NA
## 2: 10128401286 49286          3 123457684     991464        NA
## 3: 10129006933 97327          2 123457798     991464        NA
## 4: 10118306172 77984          2 123457820     989929        NA
## 5: 10131135053 15436          2 123458129     986130        NA
## 6: 10071762171 44216          1 123490760     964576        NA
##    CANCEL_DATE CANCEL_QUANTITY PRODUCT_CATEGORY_ID CHANNEL DIVISION_ID
## 1:                           0                  K      ML           1
## 2:                           0                  C      ML           1
## 3:                           0                  C      ML           1
## 4:                           0                  E      ML           1
## 5:                           0                  H      ML           1
## 6:                           0                  T      PH           5
##    OFFER_ID ORDER_DATE EXT_COST EXT_PRICE PAY_METHOD QUANTITY  SHIP_DATE
## 1:   A11LMB 21/12/2011    13.80     29.95         PC        1 21/12/2011
## 2:   A12WNB  20/1/2012    10.00     29.95         PC        1  24/1/2012
## 3:   A12SPB   2/3/2012    10.00     29.95         PC        1   5/3/2012
## 4:   A11HLB   8/3/2012    12.00     24.95         PC        1   9/3/2012
## 5:   A12OFM   7/8/2012     5.95     14.95         PC        1   8/8/2012
## 6:   W05CAT  14/2/2005     2.00      4.95         VI        1  15/2/2005
##    SHIP_QUANTITY RETN_DATE RETN_QTY RETN_REVENUE
## 1:             1         0        0            0
## 2:             1         0        0            0
## 3:             1         0        0            0
## 4:             1         0        0            0
## 5:             1         0        0            0
## 6:             1         0        0            0
```

We can even throw all of the above together!

```
head(dmef_dataset[EXT_PRICE<70 & PRODUCT_NO %in% c(987669, 989457)])
```

```
##         CUSTNO   ZIP ORDER_LINE  ORDER_NO PRODUCT_NO RETN_LINE BO_DATE
## 1: 10109980365 76443          3 123456799     989457        NA
## 2: 10116452851 49017          1 128146175     989457        NA
## 3: 10116797651 10016          1 128174012     989457        NA
## 4: 10117257911 21014          1 128219233     989457        NA
## 5: 10117257911 21014          2 128219233     989457        NA
## 6: 10117493794 30305          1 128239010     989457        NA
##    CANCEL_DATE CANCEL_QUANTITY PRODUCT_CATEGORY_ID CHANNEL DIVISION_ID
## 1:                          0                   C      ML           1
## 2:                          0                   C      WE           5
## 3:                          0                   C      WE           1
## 4:                          0                   C      WE           1
## 5:                          0                   C      WE           1
## 6:                          0                   C      WE           1
##    OFFER_ID ORDER_DATE EXT_COST EXT_PRICE PAY_METHOD QUANTITY  SHIP_DATE
## 1:   A10LCB 19/12/2010       28     69.95         PC        1 20/12/2010
## 2:   W10CMP  6/10/2010       28     69.95         VI        1  1/11/2010
## 3:   A10GFP 24/10/2010       28     69.95         MC        1  1/11/2010
## 4:   A10GFB  9/11/2010       28     69.95         VI        1 10/11/2010
## 5:   A10GFB  9/11/2010       28     69.95         VI        1 10/11/2010
## 6:   A10GFP 15/11/2010       28     69.95         MC        1 16/11/2010
##    SHIP_QUANTITY RETN_DATE RETN_QTY RETN_REVENUE
## 1:             1         0        0            0
## 2:             1         0        0            0
## 3:             1         0        0            0
## 4:             1         0        0            0
## 5:             1         0        0            0
## 6:             1         0        0            0
```

Sorting columns

data.tables also allow you to filter out columns using their headers. For example, let's take a look at some products and their cost and price.

```
ex1 <- dmef_dataset[,.(PRODUCT_NO, QUANTITY, EXT_PRICE)]
```

You can also generate new columns ased on data from existing columns, similar to what you can normally do in excel, but this time with one line of code. The `EXT_PRICE` column takes into account the total price of all the products. So lets try to find the the unit price of each product.

```
#We start by removing duplicate products. The following code removes duplicates from the dataset using

unique_ex1 <- unique(ex1, by = "PRODUCT_NO")

#Next, we'll create a new column called unit_price, and make it equal to the value of EXT_PRICE divided

unique_ex1 <- unique_ex1[, unit_price := EXT_PRICE/QUANTITY]

#To better see how the code works, let's take a look at the products with quantity greater than 1

head(unique_ex1[QUANTITY > 1])
```

```
##    PRODUCT_NO QUANTITY EXT_PRICE unit_price
## 1:     991734        3    239.85      79.95
## 2:     981222        4    139.80      34.95
## 3:     975483        3     59.85      19.95
## 4:     981856        4     59.80      14.95
## 5:     982117        2     79.90      39.95
## 6:     975592        2     49.90      24.95
```

Sorting by groups

Lastly, let's look at how to sort the data by groups. For example, let's say we want to find out the sum of the prices for each product. Fo this, we designate the product numbers as the groupings. The code should look something like this:

```
head(ex1[,.(total = sum(EXT_PRICE)), by = PRODUCT_NO])
```

```
##    PRODUCT_NO    total
## 1:     987668  5726.40
## 2:     989457  8374.05
## 3:     986782 40994.15
## 4:     986620 16996.60
## 5:     989973  1498.50
## 6:     989049   999.75
```

As some of you may have realized, we could have used this to find the unit price of each product much more easily. Let's try and compare the results.

```
head(ex1[,.(unit = sum(EXT_PRICE)/sum(QUANTITY)), by = PRODUCT_NO])
```

```
##    PRODUCT_NO      unit
## 1:     987668  79.53333
## 2:     989457  70.37017
## 3:     986782 350.37735
## 4:     986620 249.95000
## 5:     989973  49.95000
## 6:     989049 199.95000
```

```
head(unique_ex1)
```

```
##    PRODUCT_NO QUANTITY EXT_PRICE unit_price
## 1:     987668        1     79.95      79.95
## 2:     989457        1     69.95      69.95
## 3:     986782        1    349.95     349.95
## 4:     986620        1    249.95     249.95
## 5:     989973        1     49.95      49.95
## 6:     989049        1    199.95     199.95
```

Looks like there's a bit of discrepancy. While the reason may be obvious based on the different ways we approached this, let's try and confirm it with some code:

```
ex1[PRODUCT_NO == 987668, .N, by = EXT_PRICE/QUANTITY]
```

```
##    EXT_PRICE  N
## 1:     79.95 69
## 2:     59.95  2
## 3:     89.95  1
```

The above code sorted product number 987668 into different categories based on the unit price calculated from each individual transacion. As you can see, there are 69 transactions at 79.95 each, 2 transactions at 59.95 each, and one poor soul who bought the product for 89.95 each. For our first method, we removed any duplicates before calculating anything, and the duplicates were calculated based on product number. Hence, the transactions with different prices were removed, and not counted in calculating the unit price. However, when we calculated it by grouping the product numbers and calculating the overall average, these different prices were taken into account.

The important lesson here is to ensure that whichever method you use to extract data from a dataset, you should always besure that you are getting the correct values. It always helps to find out alternative methods of extracting the same data, just to check if they correlate.

Moving on, finally

Now that you know the basic functions for managing data in a data.table, let's carry on with our analysis. We'll start by removing the columns we don't care about. The `dmef_small` dataset we get will be used for the rest of this exercise.

```
dmef_small <- dmef_dataset[, .(ORDER_NO, ZIP, ORDER_LINE, PRODUCT_CATEGORY_ID,
                               CHANNEL, PRODUCT_NO, EXT_PRICE, QUANTITY,
                               RETN_QTY, RETN_REVENUE)]
```

Weird prices?

We can start doing some simple histograms to better understand the distribution of the data.

For example, let's make a histogram for price.

```
# First I create a new col for price. EXT_PRICE = PRICE * QTY so to get price..
dmef_small[, PRICE := EXT_PRICE/QUANTITY]

# You should see a new column called PRICE
head(dmef_small)
```

```
##     ORDER_NO   ZIP ORDER_LINE PRODUCT_CATEGORY_ID CHANNEL PRODUCT_NO
## 1: 123456799 76443          1                   C      ML     987668
## 2: 123456799 76443          2                   C      ML     987668
## 3: 123456799 76443          3                   C      ML     989457
## 4: 123456815 89103          1                   E      ML     986782
## 5: 123457056 28655          1                   E      ML     986620
## 6: 123457080 03743          1                   E      ML     989973
##    EXT_PRICE QUANTITY RETN_QTY RETN_REVENUE  PRICE
## 1:     79.95        1        0            0  79.95
## 2:     79.95        1        0            0  79.95
```

```
## 3:     69.95        1        0            0  69.95
## 4:    349.95        1        0            0 349.95
## 5:    249.95        1        0            0 249.95
## 6:     49.95        1        0            0  49.95
```

```
# We want to see the distribution of prices for diff objs
# But in this case, each observation is a transaction
# This means there could be super popular objects that mess things up for us
# So first let's just get prices for unique products
object.price <- dmef_small[, .(PRODUCT_NO, PRODUCT_CATEGORY_ID, PRICE)]
```

Remember the mistake we made earlier when using the unique function, in leaving out different price groups for a product? Let's try to resolve that while using the unique function. We can do this by adding another variable to the unique function, so that it searches for rows that are unique both in price and product number.

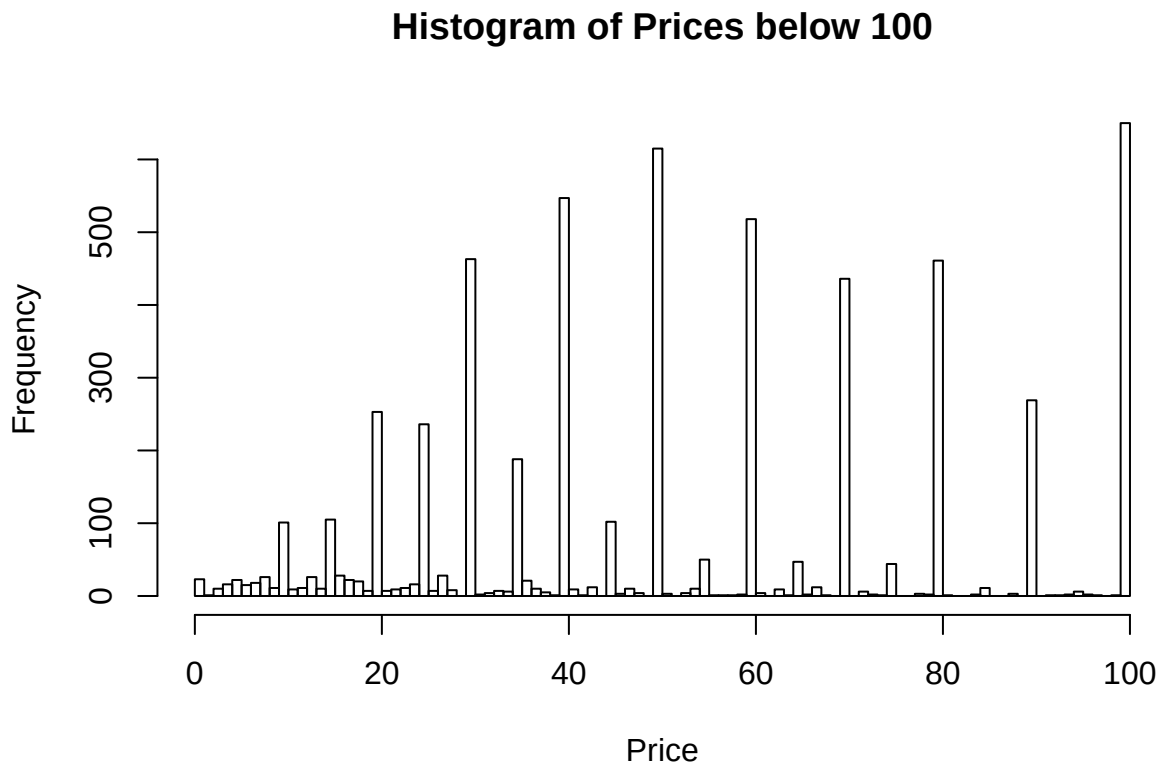
```
uniq.object.price <- unique(object.price, by = c("PRODUCT_NO", "PRICE"))
```

Now we're left with 8909 unique products with unique prices to look at. Next, let's plot them on a histogram. The histogram function is simply `hist(data table)`. There are additional parameters you can define, and you can find out more by typing `?hist()` into your console. For now, we will be using `main` to determine the name of the histogram, and `xlab` to label the x-axis.

```
# Plot prices on a histogram
hist(uniq.object.price[, PRICE],
     main = "Histogram of all prices",
     xlab = "Price")
```

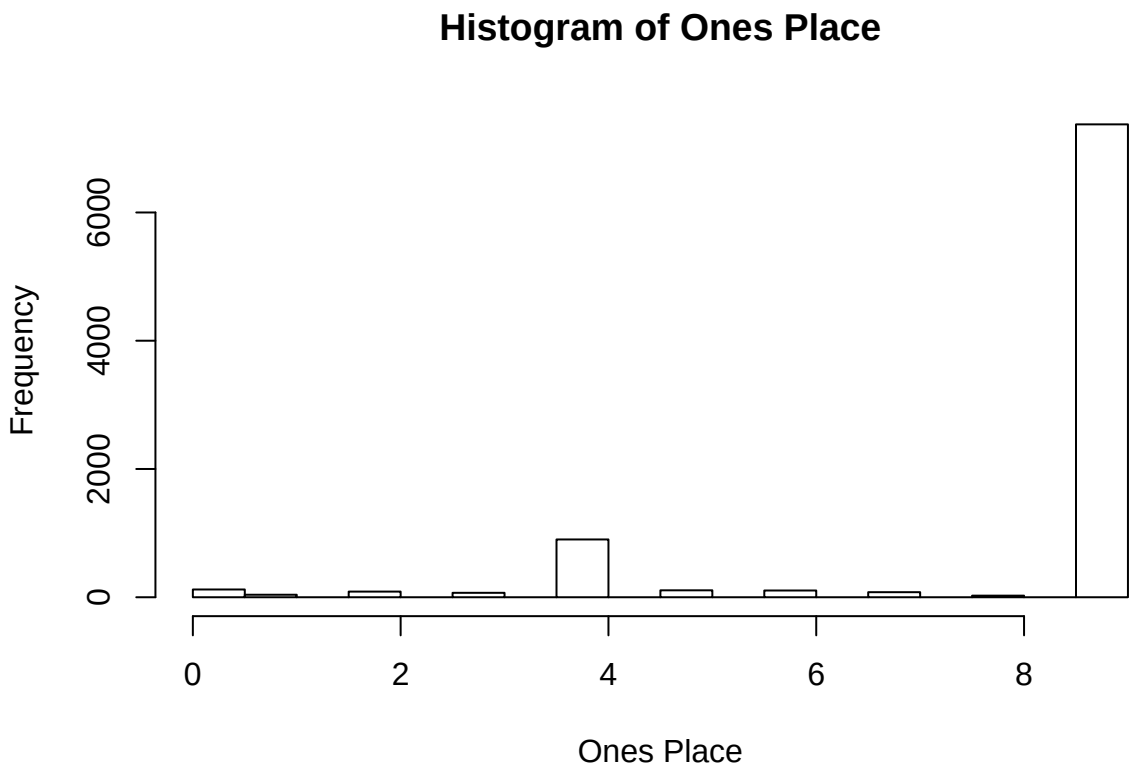
Can't really see much. . . let's filter to those cheaper than 100. Then, to ensure that we can see the prices more easily, we'll use `break` to specify for the histogram to show a hundred bars. This way, each bar will represent a dollar across the x-axis.

```
hist(uniq.object.price[PRICE < 100, PRICE], breaks=100,
     main = "Histogram of Prices below 100",
     xlab = "Price")
```

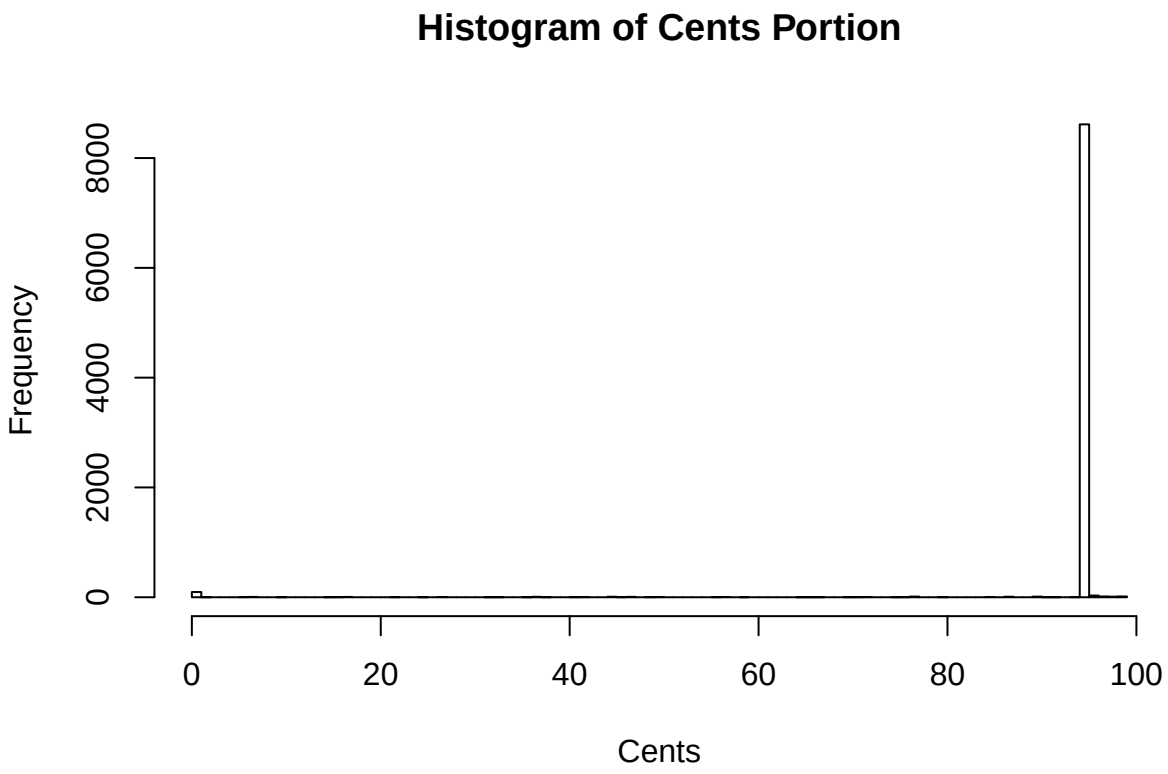


Looks like the prices bunch around certain values. Let's try to break this down, and look at the the ending number for the price. To do that, we define a function in r, which removes the decimal place, and then find the remainder when divided by 10, giving the ones place digit. After that, we'll add a new column for the ones place, and plot the histogram.

```
LastDigit <- function(x) { (x - x%%1) %% 10 }

# Add a new col ONES.PLACE

uniq.object.price[, ONES.PLACE := LastDigit(PRICE)]

hist(uniq.object.price[, ONES.PLACE],
     main = "Histogram of Ones Place",
     xlab = "Ones Place")
```




```
#Now, let's look at the cents portion. Again, we'll define a function for the cents part.

Cents <- function(x) { round(100 * (x %% 1)) }

#Creating a column for the cents.
uniq.object.price[, CENTS := Cents(PRICE)]

#We'll plot a Histogram with the same paramenters as before
hist(uniq.object.price[, CENTS], breaks=100,
     main = "Histogram of Cents Portion",
     xlab = "Cents")
```




```
# Wow okay looks like there's not much variation
# How about the exact numbers?
table(uniq.object.price[, CENTS])
```

```
##
##    0    2    6    7   10   15   16   17   22   25   27   32   33   36   37
##   95    1    3    6    1    1    2    6    2    1    4    1    1    1    8
##   38   41   42   45   46   47   49   50   56   57   59   65   66   67   70
##    1    2    3    9    2    7    2    4    2    4    1    1    1    1    2
##   71   72   75   76   77   80   85   87   90   91   92   94   95   96   97
##    2    4    1    2   12    2    4    9   11    1    1    2 8614   31   14
##   98   99
##   10   14
```

Lol is this even helpful?

I mean, the prices thing sounds interesting and all, but can we derive any cool business insights using R?

Actually, that last part was just a warm-up. Retail businesses can sometimes have problems with high return rates. Wouldn't it be useful if we could find out more about why products were being returned? Let's try finding out the proportion of returns for each product category, as well as the distribution of returned products.

Return vs Non-return

First let's split the dataset.

```
# Split into two groups
returned <- dmef_small[RETN_QTY > 0]
not.returned <- dmef_small[RETN_QTY == 0]

# Checks to make sure we're right
table(dmef_small[, RETN_QTY==0])
```

```
##
##  FALSE   TRUE
##  10742 215387
```

```
# This should match up with our number of rows
nrow(returned)
```

```
## [1] 10742
```

```
nrow(not.returned)
```

```
## [1] 215387
```

```
# Yup looks like it worked, and about 4.75% of all goods were returned?
nrow(returned)/nrow(dmef_small) * 100
```

```
## [1] 4.750386
```

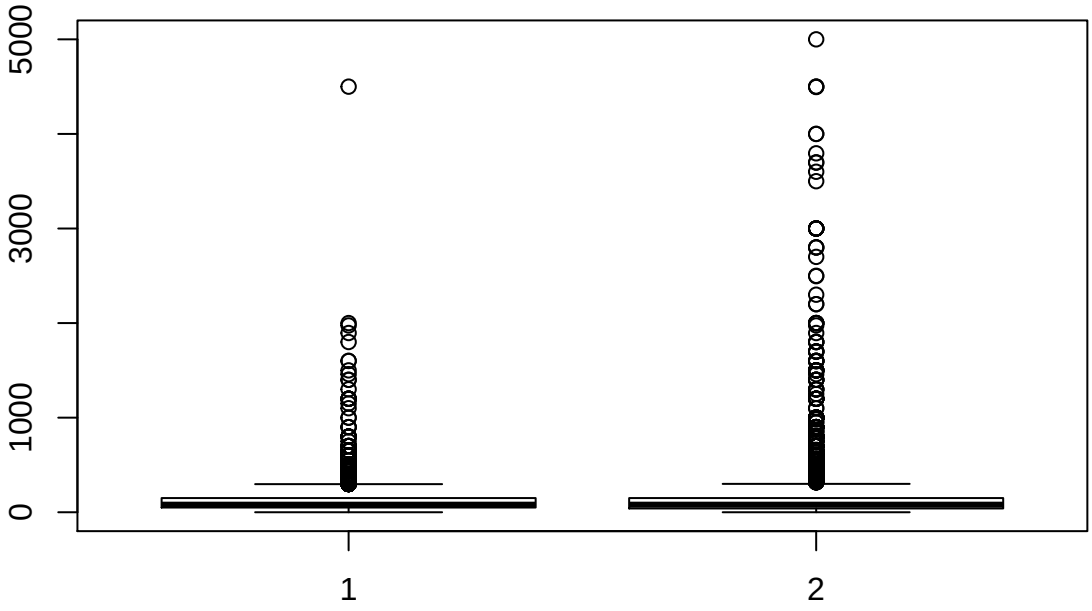
Price

Since we've been looking at price so much, was there a difference in the prices of products that were returned vs not returned? Remember we did a quick price sorting earlier, under `unique.object.price`. We'll use the `returned` and `not.returned` dataset to filter out the products from the

```
returned.object.price <- uniq.object.price[PRODUCT_NO %in% returned[, PRODUCT_NO]]
not.returned.object.price <- uniq.object.price[PRODUCT_NO %in% not.returned[, PRODUCT_NO]]
```

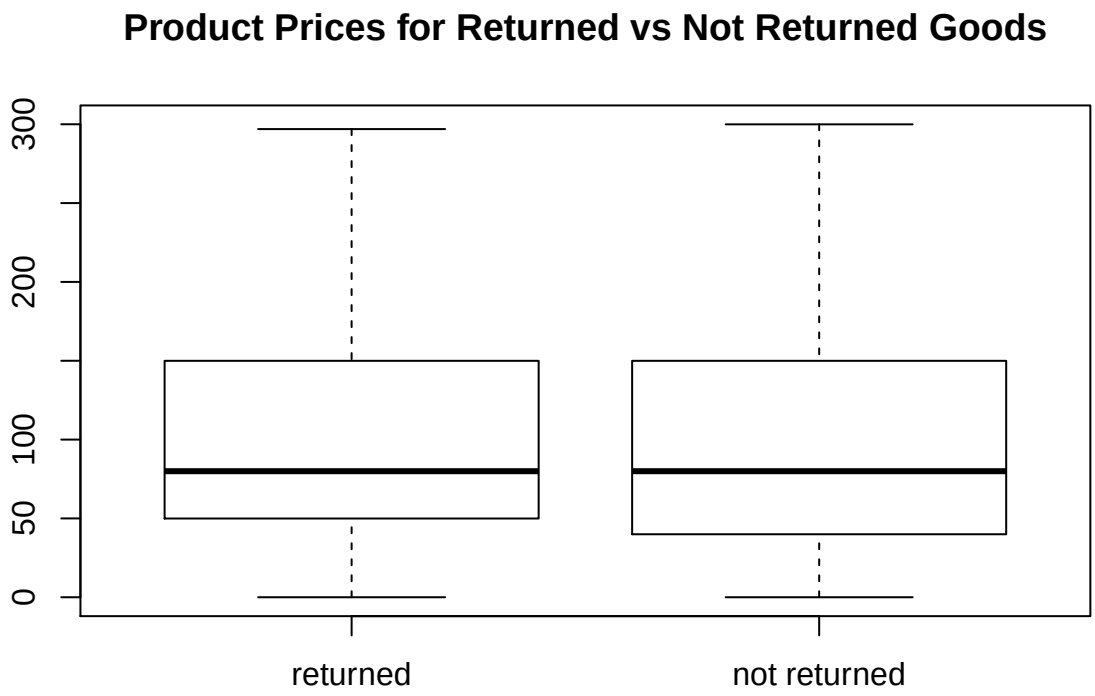
Now, we'll introduce you to another type of plot you'll encounter quite often: the box plot. The box plot shows the distribution of the dataset, with the center line representing the mean, the top and bottom of the box representing the 25th and 75th percentile respectively. Additional points outside the box represent outliers. Since a box plot is used to compare distributions, it takes in two data vectors. As usual, to find out more you can type `?boxplot` into your console.

```
# Any difference in price distribution?
boxplot(returned.object.price[, PRICE],not.returned.object.price[, PRICE])
```



Well clearly there are too many outliers. Let's ignore the outliers.

```
boxplot(returned.object.price[, PRICE],not.returned.object.price[, PRICE],
        main = "Product Prices for Returned vs Not Returned Goods",
        names = c("returned", "not returned"),
        outline=F)
```

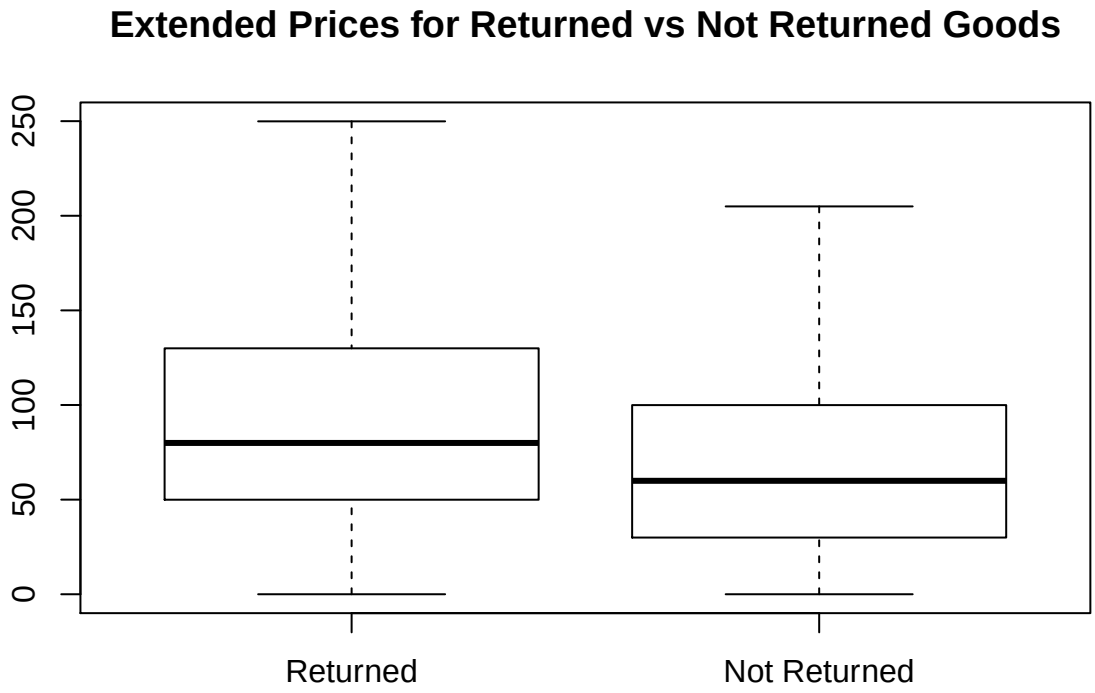


Well for the prices, it appears that median price is about the same for both, but products that are returned don't go as expensive as those that are returned. You can tell from a lower 75th percentile, and lower max (excluding outliers). Of course, it's also possible that you observe because of a greater variance. There are after all about 2.7x the number of unique products in the not returned list vs the returned list.

Total Order Size

Instead of just looking at object price, how about the total spent in that transaction, `(EXT_PRICE)`?

```
boxplot(returned[, EXT_PRICE], not.returned[, EXT_PRICE], outline=F,
        main = "Extended Prices for Returned vs Not Returned Goods",
        names = c("Returned", "Not Returned"))
```

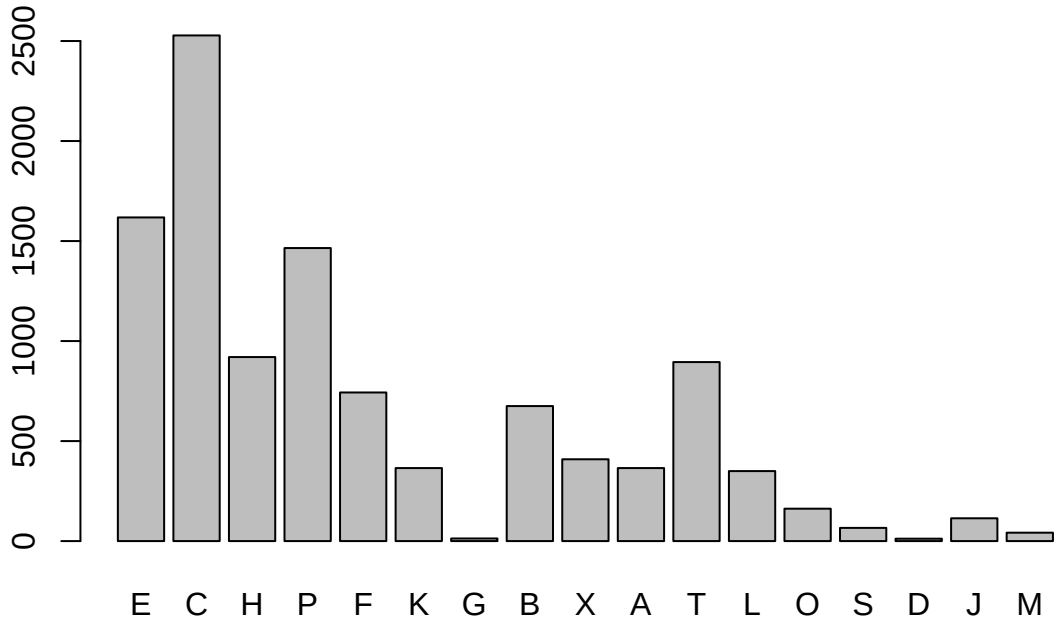


Interestingly, after accounting for outliers, it appears that returned goods tend to be in larger orders than non-returned goods. This might indicate that customers who make larger purchases tend to return goods more, or that customers who like to return goods tend to buy more.

Product Categories

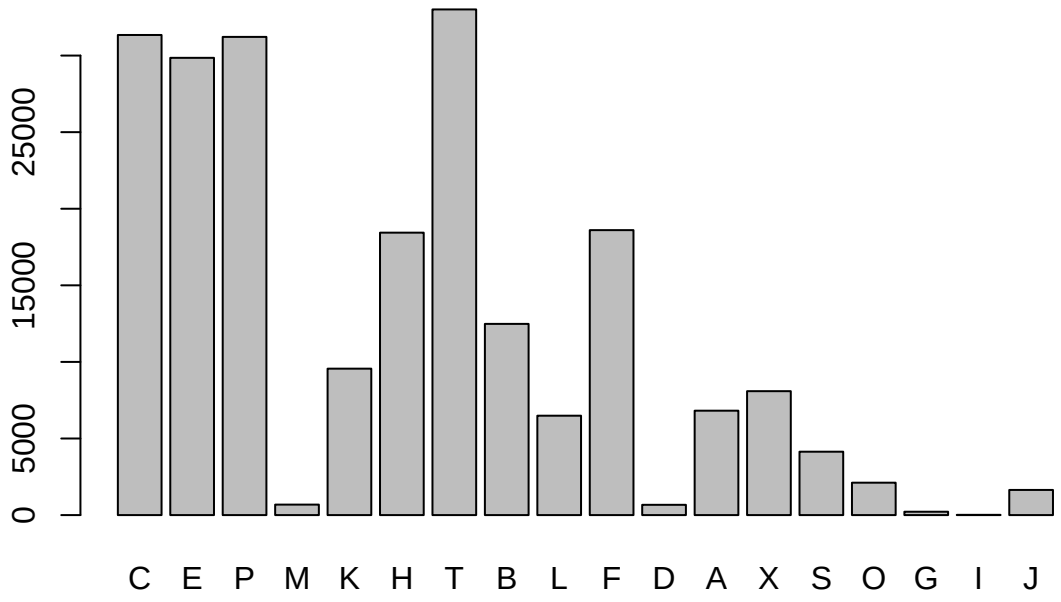
Now, let's take a look at which product categories tend to have the highest rate of returns. Firstly, we'll use a bar plot to show the proportion of returned goods that came from that product category. Let's sort the data by getting the count of returned goods per product category.

```
bar1 <- returned[, .N, by = PRODUCT_CATEGORY_ID]

#Now, let's plot the bar chart using the product counts.

barplot(bar1[,N], names = bar1[,PRODUCT_CATEGORY_ID])
```



Next, we will do the same for non-returned goods.

```
#We'll do the same conversion as we did with the returned goods.
bar2 <- not.returned[, .N, by = PRODUCT_CATEGORY_ID]

#The same goes for the bar plot.
barplot(bar2[,N], names = bar2[,PRODUCT_CATEGORY_ID])
```



When trying to compare data using bar plots, it helps if we can do an overlay to see their distributions with respect to each other, especially when the order of the bars are sorted. Let's try that.

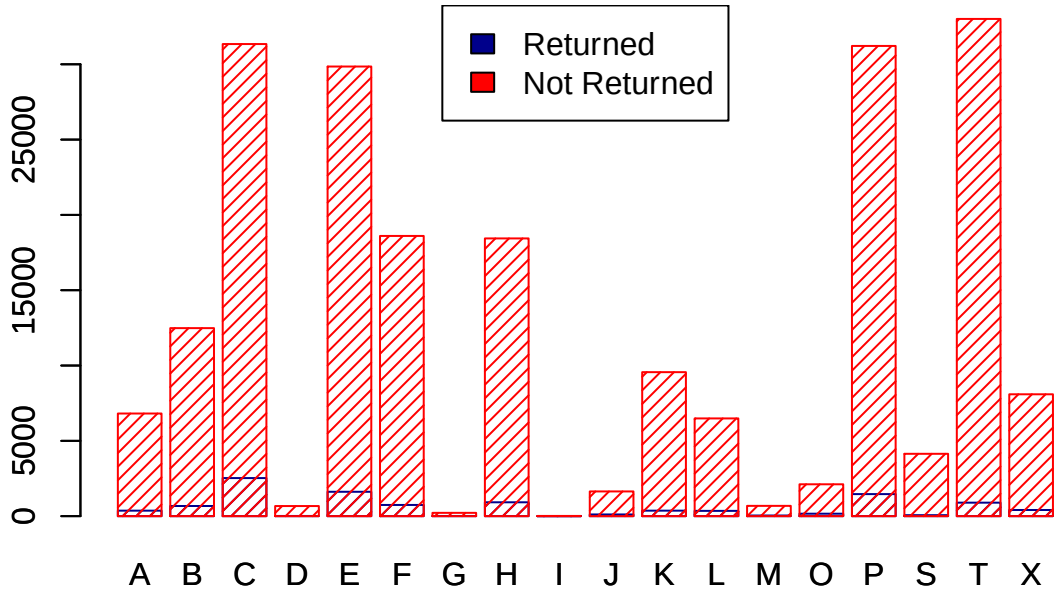
Firstly, you'll realize that product category I has never been returned, and hence does not show up in the original. We'll just fix that by adding in an empty row for I, using the `rbind` function. Next, we'll sort both tables based on their product category in alphabetical order.

```
#We'll start by sorting both the datasets based on the product category
bar1 <- rbind(bar1, data.table(PRODUCT_CATEGORY_ID = "I", N = 0))
bar1 <- bar1[order(PRODUCT_CATEGORY_ID)]
bar2 <- bar2[order(PRODUCT_CATEGORY_ID)]
```

The `order()` function sorts the rows in a data table based on the specified column. In this case, the table is now sorted in ascending alphabetical order based on the product category ID. To sort it in descending order, you can simply use `order(-PRODUCT_CATEGORY_ID)` instead

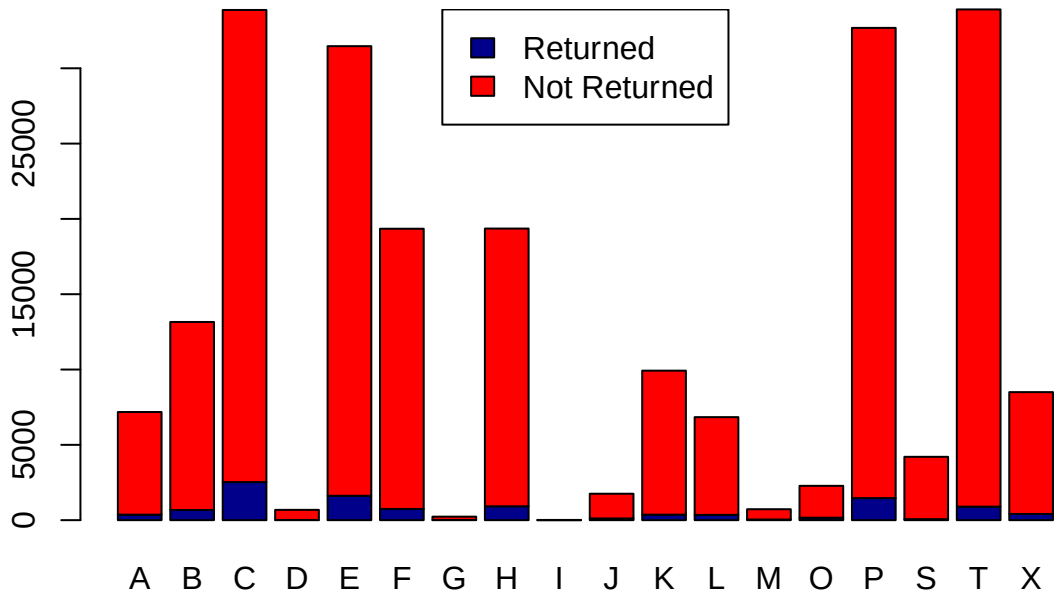
Next we'll plot the bar charts in an overlay. We'll also add in a legend for clarity.

```
barplot(bar1[,N], names = bar1[,PRODUCT_CATEGORY_ID], border = "darkblue", col = "darkblue",  density =
        ylim = c(0, max(dmef_small[, .N, by = PRODUCT_CATEGORY_ID][,N])))
par(new=TRUE)
barplot(bar2[,N], names = bar2[,PRODUCT_CATEGORY_ID], border = "red", col = "red", density = 20,
        ylim = c(0,max(dmef_small[, .N, by = PRODUCT_CATEGORY_ID][,N])))
legend("top", legend = c("Returned", "Not Returned"), fill = c("darkblue", "red"))
```



Perhaps the numbers are to large to actually get a better picture of proportion. Let's try a stacked bar plot, which also happens to be easier. We'll start by merging the datasets

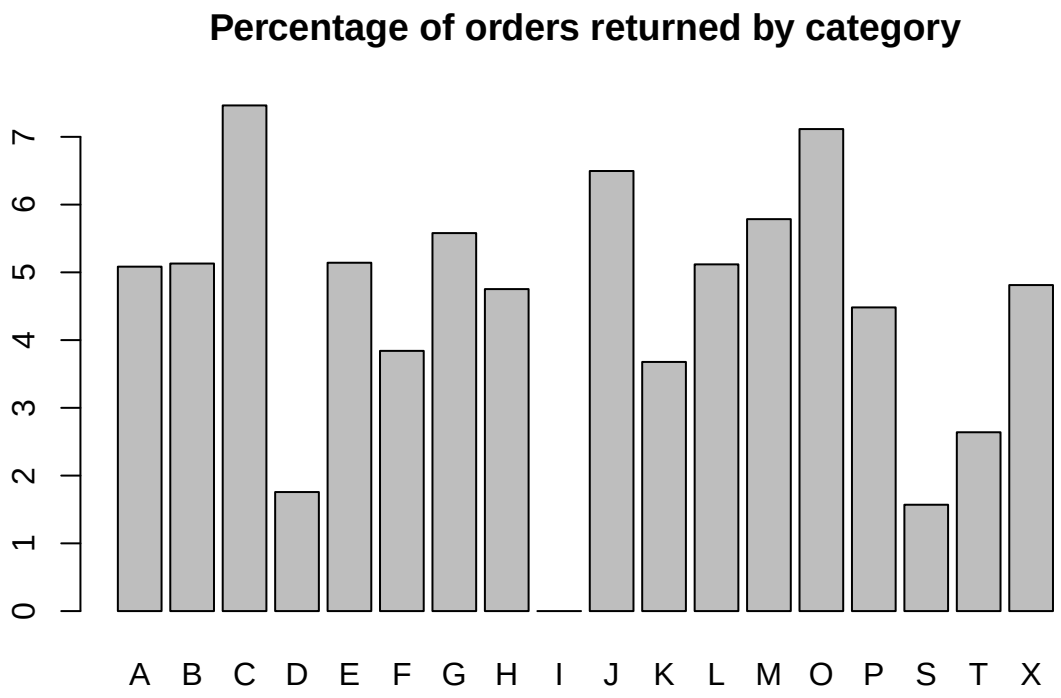
```
bar <- bar1[,.(bar1[,N], bar2[,N])]

#Next, we'll convert it to a matrix and invert it, and just insert it into the barplot function

bar <- as.matrix(t(bar))
barplot(bar,
        names = bar2[,as.character(PRODUCT_CATEGORY_ID)],
        col = c("darkblue","red"),
        ylim = c(0, max(dmef_small[, .N, by = PRODUCT_CATEGORY_ID][,N])))
legend("top", legend = c("Returned", "Not Returned"), fill = c("darkblue", "red"))
```



This all looks nice and stuff, but let's try to think what this chart means. Each barplot represents the distribution of returned and non-returned goods respectively. For example, we can see that products C, E, P and T were by far the most popular products based on volume of sales. However, take note of the small blue bars representing the number of returned goods for that product category. Although T has a marginally higher sales volume, C has a larger number of returned goods!

Let's break that information further into a more easily understandable format, and find out the percentage of goods returned for each product category.

```
# Percentage of good returned in each product category
perc.returned.by.cat <- dmef_small[order(PRODUCT_CATEGORY_ID), .(Percentage.Returned=.SD[RETN_QTY>0, .N]

#Plotting
barplot(perc.returned.by.cat[, Percentage.Returned],
        names=perc.returned.by.cat[, PRODUCT_CATEGORY_ID],
        main = "Percentage of orders returned by category")
```



Not surprisingly based on the previous graph, we can see that product category C has the highest percentage of returns. However, another category we may have missed earlier is O, which is almost as high as C! Looking back, this would be because the overall sales volume of O was so low.

Regression

For this part, we'll be using a new dataset called mtcars. It's available in R's built-in base package. Retrieve it using the `data(mtcars)` and convert it to a `data.table`.

```
# Importing the data
data(mtcars)

library(data.table)
mtcars <- as.data.table(mtcars)
```

```
# Find out more about this dataset
?mtcars
```

Using a multiple linear regression model, we want to investigate the fuel efficiency of cars by looking at their mpg (miles per gallon) rating. Let's see how each factor affects the mpg using the built in regression function in R.

```
# Fit the model
fit <- lm(mpg ~ ., data = mtcars)
summary(fit)
```

```
##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4506 -1.6044 -0.1196  1.2193  4.6271
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.30337   18.71788   0.657   0.5181
## cyl         -0.11144    1.04502  -0.107   0.9161
## disp         0.01334    0.01786   0.747   0.4635
## hp          -0.02148    0.02177  -0.987   0.3350
## drat         0.78711    1.63537   0.481   0.6353
## wt          -3.71530    1.89441  -1.961   0.0633 .
## qsec         0.82104    0.73084   1.123   0.2739
## vs           0.31776    2.10451   0.151   0.8814
## am           2.52023    2.05665   1.225   0.2340
## gear         0.65541    1.49326   0.439   0.6652
## carb        -0.19942    0.82875  -0.241   0.8122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.65 on 21 degrees of freedom
## Multiple R-squared:  0.869,  Adjusted R-squared:  0.8066
## F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

By including all the variables, we get a decent model with an adjusted R-squared of 0.8066. However, we note that the p-values are all rather high. This indicates that the model can be improved by removing some of these regressors. We start by removing cyl which has the highest p-value.

```
# Removing the regressor with the highest p-value
fit_no_cyl <- lm(mpg ~ . - cyl, data = mtcars)
summary(fit_no_cyl)
```

```
##
## Call:
## lm(formula = mpg ~ . - cyl, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4286 -1.5908 -0.0412  1.2120  4.5961
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.96007   13.53030   0.810   0.4266
## disp         0.01283    0.01682   0.763   0.4538
## hp          -0.02191    0.02091  -1.048   0.3062
## drat         0.83520    1.53625   0.544   0.5921
## wt          -3.69251    1.83954  -2.007   0.0572 .
## qsec         0.84244    0.68678   1.227   0.2329
## vs           0.38975    1.94800   0.200   0.8433
```

```
## am           2.57743    1.94035   1.328   0.1977
## gear         0.71155    1.36562   0.521   0.6075
## carb        -0.21958    0.78856  -0.278   0.7833
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.59 on 22 degrees of freedom
## Multiple R-squared:  0.8689, Adjusted R-squared:  0.8153
## F-statistic: 16.21 on 9 and 22 DF,  p-value: 9.031e-08
```

Note the improvement in adjusted R-squared. We repeat this until all our regressors have p-values below 0.05.

```
fit_improved <- lm(mpg ~ wt + qsec + am, data = mtcars)
summary(fit_improved)
```

```
##
## Call:
## lm(formula = mpg ~ wt + qsec + am, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4811 -1.5555 -0.7257  1.4110  4.6610
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   9.6178     6.9596   1.382 0.177915
## wt           -3.9165     0.7112  -5.507 6.95e-06 ***
## qsec          1.2259     0.2887   4.247 0.000216 ***
## am            2.9358     1.4109   2.081 0.046716 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.459 on 28 degrees of freedom
## Multiple R-squared:  0.8497, Adjusted R-squared:  0.8336
## F-statistic: 52.75 on 3 and 28 DF,  p-value: 1.21e-11
```

This process can actually be automated using the stepAIC function in the MASS package.

```
library(MASS)
stepAIC(fit)
```

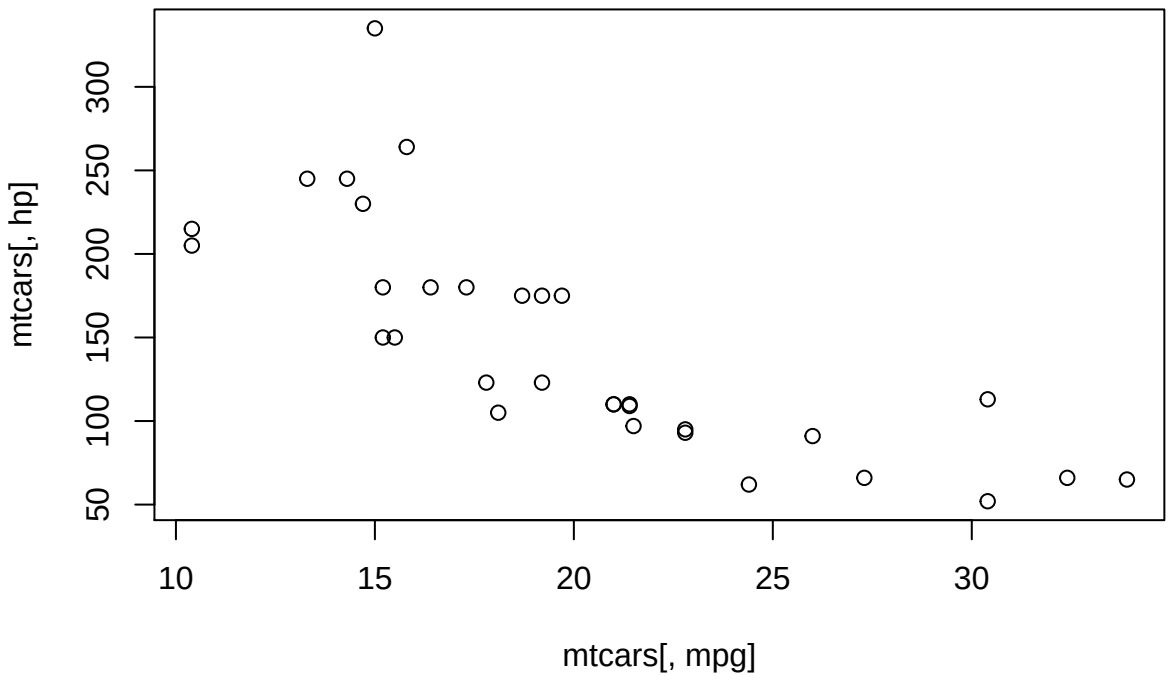
```
## Start:  AIC=70.9
## mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am + gear + carb
##
##        Df Sum of Sq    RSS    AIC
## - cyl   1    0.0799 147.57 68.915
## - vs    1    0.1601 147.66 68.932
## - carb  1    0.4067 147.90 68.986
## - gear  1    1.3531 148.85 69.190
## - drat  1    1.6270 149.12 69.249
## - disp  1    3.9167 151.41 69.736
## - hp    1    6.8399 154.33 70.348
```

```
## - qsec  1    8.8641 156.36 70.765
## <none>              147.49 70.898
## - am    1   10.5467 158.04 71.108
## - wt    1   27.0144 174.51 74.280
##
## Step:  AIC=68.92
## mpg ~ disp + hp + drat + wt + qsec + vs + am + gear + carb
##
##        Df Sum of Sq    RSS    AIC
## - vs    1    0.2685 147.84 66.973
## - carb  1    0.5201 148.09 67.028
## - gear  1    1.8211 149.40 67.308
## - drat  1    1.9826 149.56 67.342
## - disp  1    3.9009 151.47 67.750
## - hp    1    7.3632 154.94 68.473
## <none>              147.57 68.915
## - qsec  1   10.0933 157.67 69.032
## - am    1   11.8359 159.41 69.384
## - wt    1   27.0280 174.60 72.297
##
## Step:  AIC=66.97
## mpg ~ disp + hp + drat + wt + qsec + am + gear + carb
##
##        Df Sum of Sq    RSS    AIC
## - carb  1    0.6855 148.53 65.121
## - gear  1    2.1437 149.99 65.434
## - drat  1    2.2139 150.06 65.449
## - disp  1    3.6467 151.49 65.753
## - hp    1    7.1060 154.95 66.475
## <none>              147.84 66.973
## - am    1   11.5694 159.41 67.384
## - qsec  1   15.6830 163.53 68.200
## - wt    1   27.3799 175.22 70.410
##
## Step:  AIC=65.12
## mpg ~ disp + hp + drat + wt + qsec + am + gear
##
##        Df Sum of Sq    RSS    AIC
## - gear  1     1.565 150.09 63.457
## - drat  1     1.932 150.46 63.535
## <none>              148.53 65.121
## - disp  1    10.110 158.64 65.229
## - am    1    12.323 160.85 65.672
## - hp    1    14.826 163.35 66.166
## - qsec  1    26.408 174.94 68.358
## - wt    1    69.127 217.66 75.350
##
## Step:  AIC=63.46
## mpg ~ disp + hp + drat + wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## - drat  1     3.345 153.44 62.162
## - disp  1     8.545 158.64 63.229
## <none>              150.09 63.457
```

```
## - hp    1    13.285 163.38 64.171
## - am    1    20.036 170.13 65.466
## - qsec  1    25.574 175.67 66.491
## - wt    1    67.572 217.66 73.351
##
## Step:  AIC=62.16
## mpg ~ disp + hp + wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## - disp  1     6.629 160.07 61.515
## <none>              153.44 62.162
## - hp    1    12.572 166.01 62.682
## - qsec  1    26.470 179.91 65.255
## - am    1    32.198 185.63 66.258
## - wt    1    69.043 222.48 72.051
##
## Step:  AIC=61.52
## mpg ~ hp + wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## - hp    1     9.219 169.29 61.307
## <none>              160.07 61.515
## - qsec  1    20.225 180.29 63.323
## - am    1    25.993 186.06 64.331
## - wt    1    78.494 238.56 72.284
##
## Step:  AIC=61.31
## mpg ~ wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## <none>              169.29 61.307
## - am    1    26.178 195.46 63.908
## - qsec  1   109.034 278.32 75.217
## - wt    1   183.347 352.63 82.790


##
## Call:
## lm(formula = mpg ~ wt + qsec + am, data = mtcars)
##
## Coefficients:
## (Intercept)           wt         qsec           am
##       9.618       -3.917        1.226        2.936
```

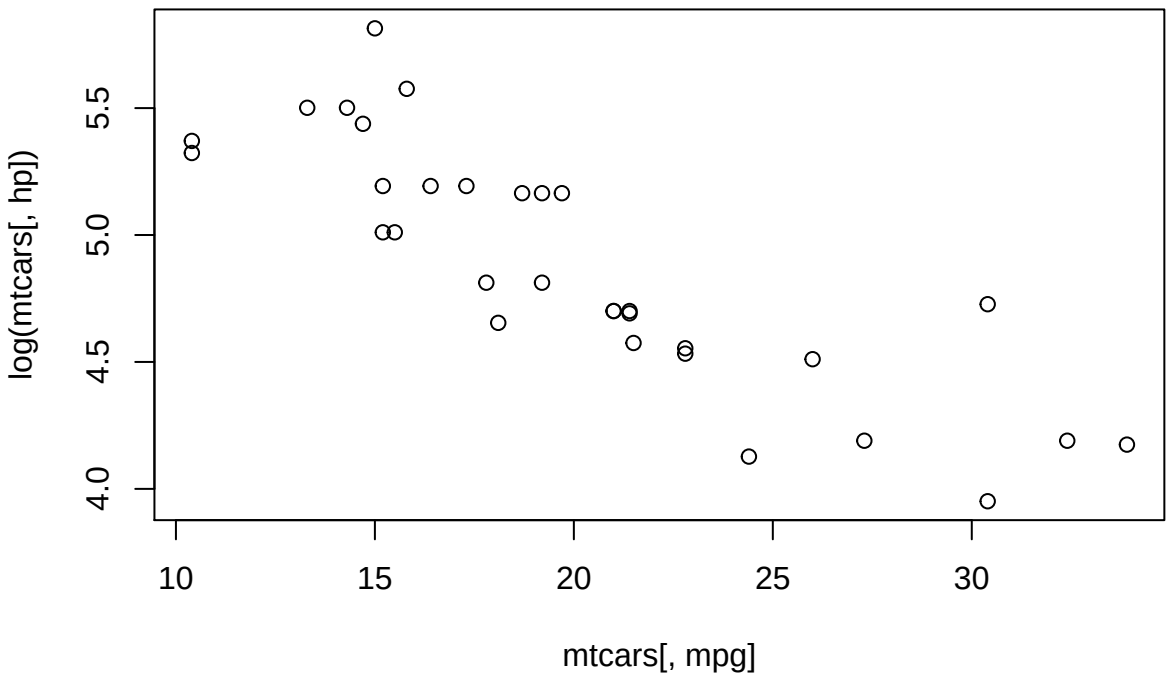
However, there are actually better models that you can obtain by applying some simple transformations. For example, let's take a look at the relationship between mpg and hp.

```
plot(mtcars[, mpg], mtcars[, hp])
```



We can see there isn't a very distinct linear relationship, which can hurt our linear regression model. So let's apply a basic log transformation.

```
plot(mtcars[, mpg], log(mtcars[, hp]))
```



And that looks significantly better. Let's try this new adjusted hp in this regression model. Using the same steps as before:

```
fit_transformed <- lm(mpg ~ . - hp + log(hp), data=mtcars)
stepAIC(fit_transformed)
```

```
## Start:  AIC=67.15
## mpg ~ (cyl + disp + hp + drat + wt + qsec + vs + am + gear +
##     carb) - hp + log(hp)
##
##           Df Sum of Sq    RSS    AIC
## - drat     1    0.0231 131.21 65.154
## - cyl      1    0.0508 131.24 65.161
## - vs       1    0.1177 131.31 65.177
## - carb     1    0.3123 131.50 65.225
## - gear     1    3.4982 134.69 65.991
## - qsec     1    4.0112 135.20 66.112
## - disp     1    4.9119 136.10 66.325
## - am       1    6.4492 137.64 66.684
## <none>                 131.19 67.149
## - log(hp)  1   23.1462 154.33 70.348
## - wt       1   24.1001 155.29 70.546
##
## Step:  AIC=65.15
## mpg ~ cyl + disp + wt + qsec + vs + am + gear + carb + log(hp)
##
```

```
##           Df Sum of Sq    RSS    AIC
## - cyl      1    0.0385 131.25 63.164
## - vs       1    0.1225 131.33 63.184
## - carb     1    0.2893 131.50 63.225
## - gear     1    3.6153 134.83 64.024
## - qsec     1    3.9891 135.20 64.113
## - disp     1    5.1573 136.37 64.388
## - am       1    6.6248 137.84 64.731
## <none>                 131.21 65.154
## - wt       1   24.9628 156.17 68.728
## - log(hp)  1   25.3991 156.61 68.817
##
## Step:  AIC=63.16
## mpg ~ disp + wt + qsec + vs + am + gear + carb + log(hp)
##
##           Df Sum of Sq    RSS    AIC
## - vs       1    0.0916 131.34 61.186
## - carb     1    0.2555 131.50 61.226
## - gear     1    3.9854 135.24 62.121
## - qsec     1    4.0148 135.26 62.128
## - disp     1    6.0243 137.27 62.600
## - am       1    6.8252 138.07 62.786
## <none>                 131.25 63.164
## - wt       1   25.5756 156.82 66.861
## - log(hp)  1   27.0102 158.26 67.152
##
## Step:  AIC=61.19
## mpg ~ disp + wt + qsec + am + gear + carb + log(hp)
##
##           Df Sum of Sq    RSS    AIC
## - carb     1    0.2755 131.62 59.253
## - gear     1    4.4333 135.78 60.248
## - disp     1    5.9362 137.28 60.601
## - qsec     1    5.9797 137.32 60.611
## - am       1    6.7341 138.08 60.786
## <none>                 131.34 61.186
## - wt       1   26.4325 157.77 65.054
## - log(hp)  1   26.9244 158.27 65.153
##
## Step:  AIC=59.25
## mpg ~ disp + wt + qsec + am + gear + log(hp)
##
##           Df Sum of Sq    RSS    AIC
## - gear     1     4.371 135.99 58.299
## - am       1     6.812 138.43 58.868
## <none>                 131.62 59.253
## - qsec     1     9.184 140.80 59.411
## - disp     1    10.665 142.28 59.746
## - log(hp)  1    34.290 165.91 64.662
## - wt       1    53.234 184.85 68.122
##
## Step:  AIC=58.3
## mpg ~ disp + wt + qsec + am + log(hp)
##
```

```
##           Df Sum of Sq    RSS    AIC
## - disp     1     6.975 142.96 57.899
## - qsec     1     8.633 144.62 58.268
## <none>                 135.99 58.299
## - am       1    20.830 156.82 60.859
## - log(hp)  1    30.021 166.01 62.682
## - wt       1    50.604 186.59 66.422
##
## Step:  AIC=57.9
## mpg ~ wt + qsec + am + log(hp)
##
##           Df Sum of Sq    RSS    AIC
## - qsec     1     4.350 147.31 56.859
## <none>                 142.96 57.899
## - am       1    15.536 158.50 59.201
## - log(hp)  1    26.322 169.29 61.307
## - wt       1    51.845 194.81 65.801
##
## Step:  AIC=56.86
## mpg ~ wt + am + log(hp)
##
##           Df Sum of Sq    RSS    AIC
## <none>                 147.31 56.859
## - am       1    11.297 158.61 57.223
## - wt       1    47.496 194.81 63.801
## - log(hp)  1   131.006 278.32 75.217

##
## Call:
## lm(formula = mpg ~ wt + am + log(hp), data = mtcars)
##
## Coefficients:
## (Intercept)           wt           am      log(hp)
##      58.996       -2.468        1.757       -6.489
```

```
fit_final <- lm(mpg ~ wt + am + log(hp), data=mtcars)
summary(fit_final)
```

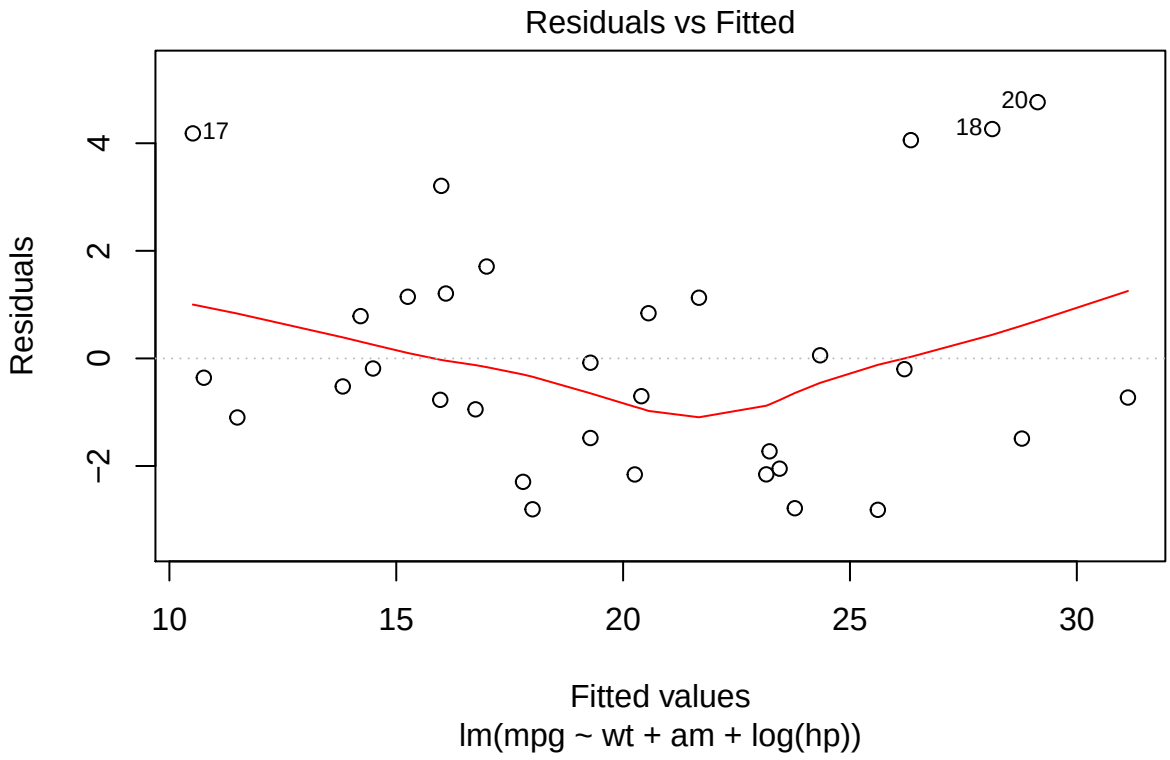
```
##
## Call:
## lm(formula = mpg ~ wt + am + log(hp), data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.8148 -1.5495 -0.4402  1.1327  4.7638
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  58.9957     4.8970  12.047 1.36e-12 ***
## wt           -2.4681     0.8215  -3.005  0.00555 **
## am            1.7565     1.1987   1.465  0.15397
## log(hp)      -6.4888     1.3004  -4.990 2.85e-05 ***
## ---
```

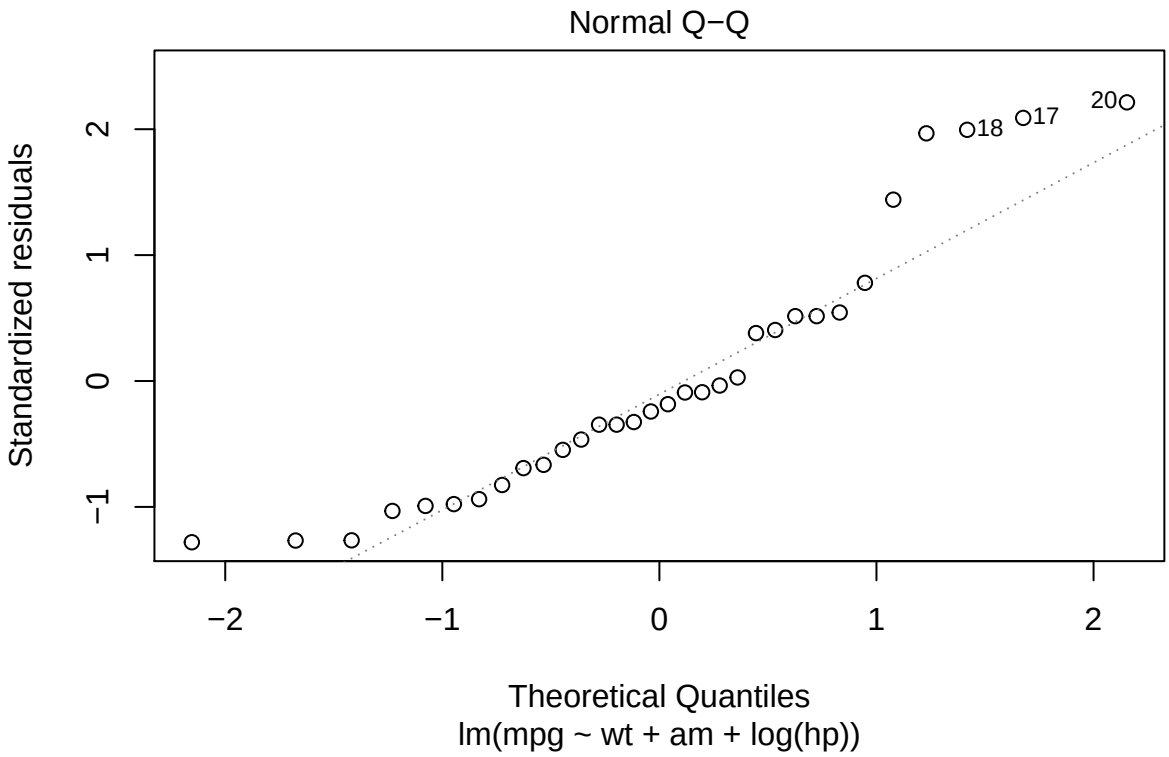
```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.294 on 28 degrees of freedom
## Multiple R-squared:  0.8692, Adjusted R-squared:  0.8552
## F-statistic: 62.01 on 3 and 28 DF,  p-value: 1.747e-12
```

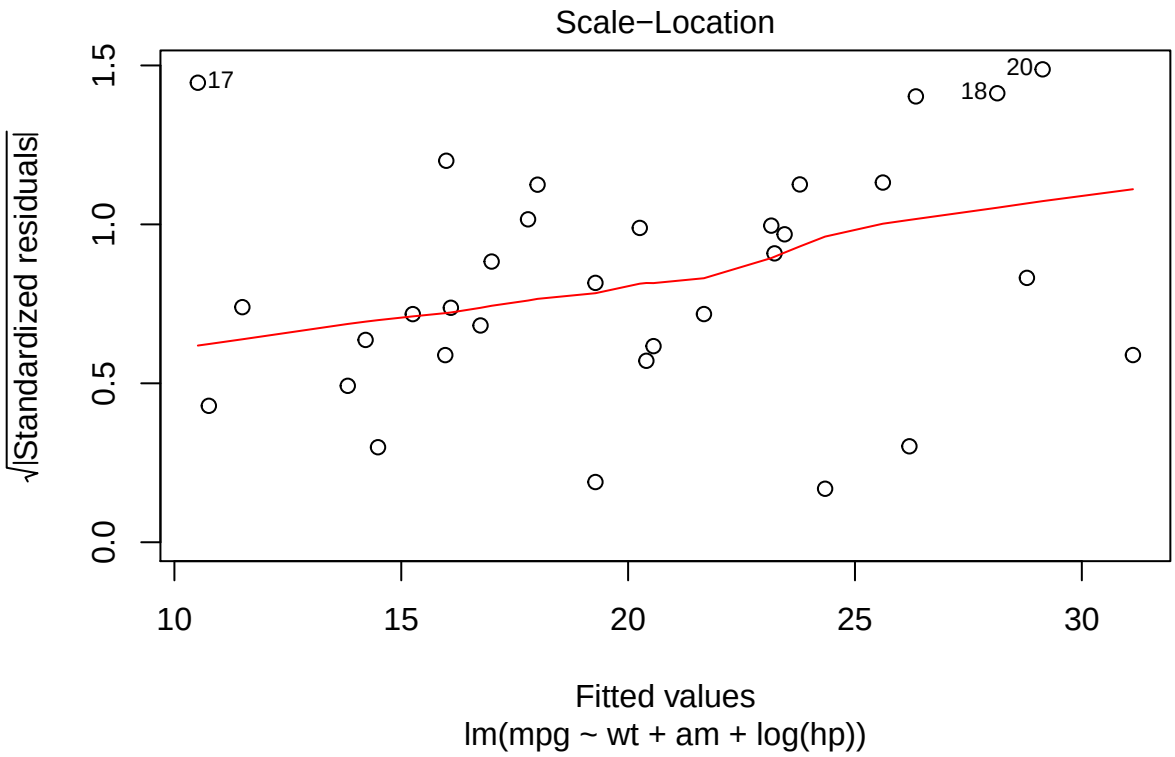
However, having a good fit for the regression model is not all that matters. You need to check if your four regression assumptions are met. The four assumptions include:

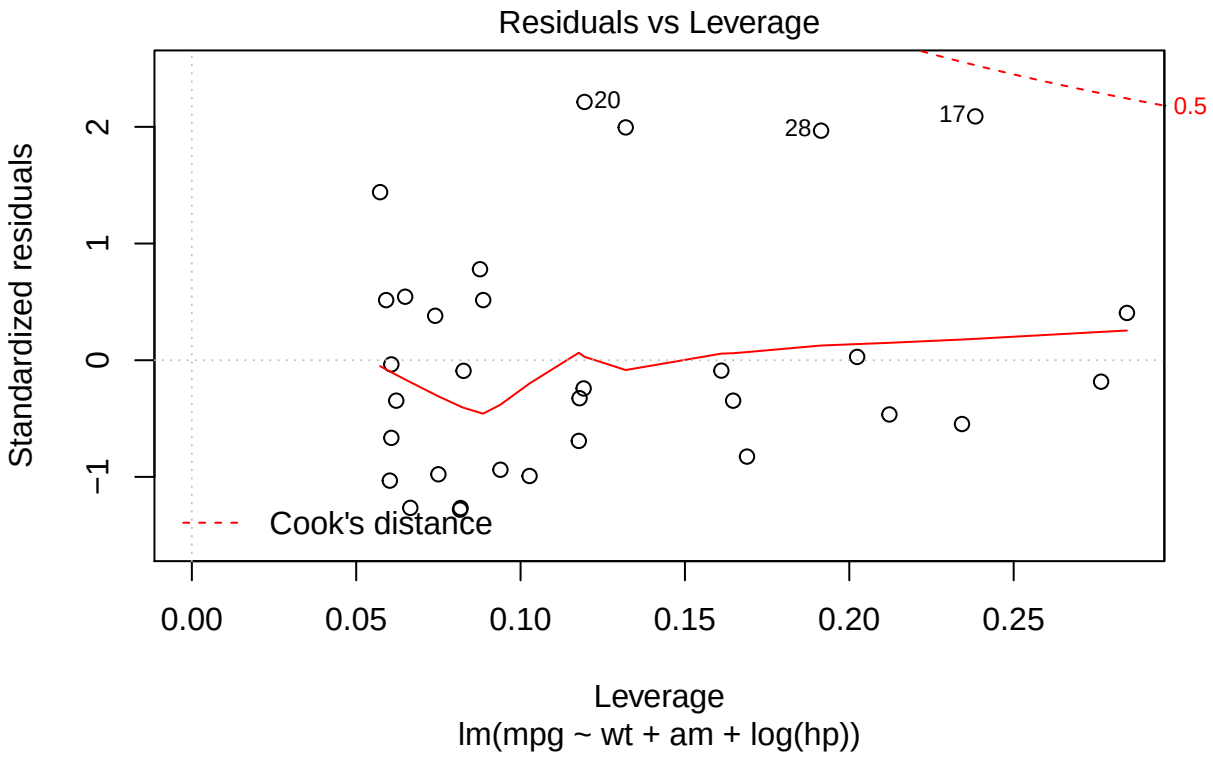
- normality
- homoscedasticity
- linearity
- independence of errors (generally applies to time series data)

```
# Regression Assumptions
plot(fit_final)
```




Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(mpg ~ wt + am + log(hp))
18
17
20
−2
−1
0
1
2


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
17
18
20
10
15
20
25
30
Fitted values
lm(mpg ~ wt + am + log(hp))



The first plot tests your assumptions for linearity and homoscedasticity. Firstly, if the red line is straight, then the assumption for linearity has been cleared. In this case, although the red line is not straight, we can see that this is due to a few outliers in the top right corner, while the bulk of the data points are roughly linear, and we can accept this. Secondly, we need to ensure that the variance between the data points are constant. Looking at this dataset, there data points appear evenly distributed and there is no distinct change in variance. Hence, the assumption for homoscedasticity is considered valid.

The second plot is used to test for normality. The regression assumes that the data points follow a normal distribution and if this is the case the data points on the QQ plot should closely align to the line on the chart. In this plot we can see that this is so and hence we can hold the assumption for normality.

The last chart measures Cook's distance, which is the acceptable range for outliers. the red dotted line at the corner measures limit for outliers, and since none of our data points reside outside that line, we can accept this regression model.

```
library(car)
durbinWatsonTest(fit_final)
```

```
##  lag Autocorrelation D-W Statistic p-value
##    1      0.06134832      1.796144   0.354
##  Alternative hypothesis: rho != 0
```

The last test is the test for independence of errors. This tends to be a significant problem with time series data, where yesterday's stock price has a high chance of affecting today's price. To test for this, we run the Durbin Watson test. As long as the Durbin Watson Statistic is between 1.5 and 2.5, the assumption for independence of errors is valid.

```r
# Check for multi-collinearity
# Anything below 10 is okay
library(car)
vif(fit_final)
```

```
##       wt       am  log(hp)
## 3.806564 2.108117 2.250186
```

Lastly, if two variables in your regression model are highly correlated, it can affect the outcome of your regression. Hence, we run the Variance Inflation Factor test to ensure that the variables used in your regression model are not correlated. As long as all of the vif values are less than 10, the regression model will hold.