Basic analysis & Other topics

Hrvoje Stojic

September 15, 2017

Introduction

The last handout gave an overview on importing data, the essentials of R plots, and handling text and dates. All these represent the first step in an project. In this handout we will cover the actual data analysis, but prior to performing any data analysis we need to go over few other concepts.

The apply family of functions

In R it is usually advisable to vectorize your operations, i.e. performing operations to all elements of a vector (or other more general objects) instead of performing inefficient loop structures. Most R functions vectorize by default, for example the algebraic operations.

```
c(1, 2, 3, 4, 5) + 5
exp(c(1, 2, 3, 4, 5))
```

When programming your routines you too should think on applying a single operation to all elements at the same time (and not on element-by-element terms). The apply() functions are a natural way of using this philosophy, specially when manipulating data frames:

```
# Summarizing columns
apply(mtcars, MARGIN = 2, FUN = mean)
apply(mtcars, MARGIN = 2, FUN = range)

# Summarizing rows
apply(mtcars, MARGIN = 1, FUN = mean)
apply(mtcars, MARGIN = 1, FUN = range)

# Custom functions 1: coefficient of variation
apply(mtcars, MARGIN = 2, FUN = function(x) sd(x)/mean(x))

# Custom functions 2: range
apply(mtcars, MARGIN = 2, FUN = function(x) c(min(x),
    max(x)))
```

```
##        mpg        cyl       disp         hp
##  20.090625   6.187500 230.721875 146.687500
```

```
##       drat         wt       qsec         vs
##   3.596563   3.217250  17.848750   0.437500
##         am       gear       carb
##   0.406250   3.687500   2.812500
##       mpg cyl  disp  hp drat    wt qsec vs
## [1,] 10.4   4  71.1  52 2.76 1.513 14.5  0
## [2,] 33.9   8 472.0 335 4.93 5.424 22.9  1
##      am gear carb
## [1,]  0    3    1
## [2,]  1    5    8
##           Mazda RX4       Mazda RX4 Wag
##            29.90727            29.98136
##          Datsun 710      Hornet 4 Drive
##            23.59818            38.73955
##   Hornet Sportabout             Valiant
##            53.66455            35.04909
##          Duster 360           Merc 240D
##            59.72000            24.63455
##            Merc 230            Merc 280
##            27.23364            31.86000
##           Merc 280C          Merc 450SE
##            31.78727            46.43091
##          Merc 450SL         Merc 450SLC
##            46.50000            46.35000
##  Cadillac Fleetwood Lincoln Continental
##            66.23273            66.05855
##   Chrysler Imperial            Fiat 128
##            65.97227            19.44091
##         Honda Civic      Toyota Corolla
##            17.74227            18.81409
##       Toyota Corona    Dodge Challenger
##            24.88864            47.24091
##         AMC Javelin          Camaro Z28
##            46.00773            58.75273
##    Pontiac Firebird           Fiat X1-9
##            57.37955            18.92864
##       Porsche 914-2        Lotus Europa
##            24.77909            24.88027
##      Ford Pantera L        Ferrari Dino
##            60.97182            34.50818
##       Maserati Bora          Volvo 142E
##            63.15545            26.26273
##      Mazda RX4 Mazda RX4 Wag Datsun 710
## [1,]         0             0          1
```

```
## [2,]       160           160        108
##      Hornet 4 Drive Hornet Sportabout
## [1,]              0                 0
## [2,]            258               360
##      Valiant Duster 360 Merc 240D Merc 230
## [1,]       0          0       0.0      0.0
## [2,]     225        360     146.7    140.8
##      Merc 280 Merc 280C Merc 450SE
## [1,]      0.0       0.0        0.0
## [2,]    167.6     167.6      275.8
##      Merc 450SL Merc 450SLC
## [1,]        0.0         0.0
## [2,]      275.8       275.8
##      Cadillac Fleetwood Lincoln Continental
## [1,]                  0                   0
## [2,]                472                 460
##      Chrysler Imperial Fiat 128 Honda Civic
## [1,]                 0      1.0         1.0
## [2,]               440     78.7        75.7
##      Toyota Corolla Toyota Corona
## [1,]            1.0           0.0
## [2,]           71.1         120.1
##      Dodge Challenger AMC Javelin Camaro Z28
## [1,]                0           0          0
## [2,]              318         304        350
##      Pontiac Firebird Fiat X1-9
## [1,]                0         1
## [2,]              400        79
##      Porsche 914-2 Lotus Europa
## [1,]           0.0            1
## [2,]         120.3          113
##      Ford Pantera L Ferrari Dino
## [1,]              0            0
## [2,]            351          175
##      Maserati Bora Volvo 142E
## [1,]             0          1
## [2,]           335        121
##       mpg       cyl      disp        hp
## 0.2999881 0.2886338 0.5371779 0.4674077
##      drat        wt      qsec        vs
## 0.1486638 0.3041285 0.1001159 1.1520369
##        am      gear      carb
## 1.2282853 0.2000825 0.5742933
##       mpg cyl  disp  hp drat    wt qsec vs
```

```
## [1,] 10.4   4  71.1  52 2.76 1.513 14.5  0
## [2,] 33.9   8 472.0 335 4.93 5.424 22.9  1
##      am gear carb
## [1,]  0    3    1
## [2,]  1    5    8
```

There are several variants to the `apply` function.

```
# Summaryzing grouping by factors
tapply(mtcars$hp, mtcars$cyl, mean)

# Summarizing lists
list <- list(a = 1:5, b = 6:20, c = 21:99)
sapply(list, mean)
lapply(list, mean)

# Replicate vector operations
replicate(5, rnorm(10))
```

```
##         4         6         8
##  82.63636 122.28571 209.21429
##  a  b  c
##  3 13 60
## $a
## [1] 3
##
## $b
## [1] 13
##
## $c
## [1] 60
##
##             [,1]          [,2]       [,3]
##  [1,]  0.5407623  0.1712850911 -1.2069464
##  [2,]  1.7199221  0.0005249014  0.7860011
##  [3,] -0.6868320  0.3142832197 -2.4226577
##  [4,]  0.8859037 -1.3883728743  0.9009070
##  [5,]  0.6117014 -0.9630541388 -0.7102741
##  [6,]  0.5742805 -0.2033726137 -0.0669236
##  [7,] -0.6356455  1.4444301006  0.2936719
##  [8,]  1.1902311 -1.0656405797 -1.1392490
##  [9,]  0.4676337  2.0951250174  1.2484690
## [10,] -0.8744605 -1.8932764765  1.0422503
##             [,4]       [,5]
##  [1,] -0.2927180  1.7513480
```

```
##  [2,] -0.1458198 -0.9163917
##  [3,]  1.1381003  0.7739408
##  [4,] -0.6519375 -0.7253593
##  [5,]  1.2137893  1.2513494
##  [6,] -1.0967150 -0.3119181
##  [7,]  0.7074462  0.1765839
##  [8,] -0.3096904 -0.6377105
##  [9,]  0.2887965  0.1142709
## [10,] -0.3133365 -0.7259293
```

Practice `apply` functions. Apply the `fivenum` function to 1) all the columns of `mtcars` , customizing the output row names; 2) the weight, grouping by number of carburetors.

Basic statistics

Making sense out of large amounts of data is the everyday task of data scientists. You will often be confronted with the need to summarize thousands of observations into a simpler form a human brain understands, be it a table with just a few numbers or a pretty picture.

You may need to do this for exploratory reasons: understanding the characteristics of your data is always key for success in subsequent analysis. You may need this for pedagogical reasons, say if you are to present your results to a manager or client. In this topic we review some of the most common techniques data scientists use to summarize their data and how to apply them in R. We also dip our toes into the waters of statistical inference by reviewing linear regression.

Graphics

We have already seen graphics with `base` R, but I favor `ggplot2` package over `base` graphics that comes by defualt in R. In my opinion this package is easier to learn and you can generate stunning graphics with much fewer steps than with `base` R.

```
# install.packages('ggplot2')
library(ggplot2)
```

As we go along showing how to generate various summary statistics we will illustrate the same data in graphical format with `ggplot2`.

The most basic plotting function in this package is `qplot`, designed to be familiar if you are used to plot from the base package. Look online for a more extensive description by `ggplot2` creator Hadley Wickham. We will be just touching the surface of what you can do with ggplot2. Do yourself a favor and dig deep into the capabilities of this library.

Data

You will need to install the package hflights, which contains data for all flights in 2011 departing from major Houston airports, IAH (George Bush Intercontinental) and HOU (Houston Hobby). Load the data by calling the package using the library function. Once you load the data you can use the head function to inspect the table.

```
# install.packages('hflights')
library(hflights)
head(hflights)
```

```
##      Year Month DayofMonth DayOfWeek DepTime
## 5424 2011     1          1         6    1400
## 5425 2011     1          2         7    1401
## 5426 2011     1          3         1    1352
## 5427 2011     1          4         2    1403
## 5428 2011     1          5         3    1405
## 5429 2011     1          6         4    1359
##      ArrTime UniqueCarrier FlightNum TailNum
## 5424    1500            AA       428  N576AA
## 5425    1501            AA       428  N557AA
## 5426    1502            AA       428  N541AA
## 5427    1513            AA       428  N403AA
## 5428    1507            AA       428  N492AA
## 5429    1503            AA       428  N262AA
##      ActualElapsedTime AirTime ArrDelay
## 5424                60      40      -10
## 5425                60      45       -9
## 5426                70      48       -8
## 5427                70      39        3
## 5428                62      44       -3
## 5429                64      45       -7
##      DepDelay Origin Dest Distance TaxiIn
## 5424        0    IAH  DFW      224      7
## 5425        1    IAH  DFW      224      6
## 5426       -8    IAH  DFW      224      5
## 5427        3    IAH  DFW      224      9
## 5428        5    IAH  DFW      224      9
## 5429       -1    IAH  DFW      224      6
##      TaxiOut Cancelled CancellationCode
## 5424      13         0
```

```
## 5425       9         0
## 5426      17         0
## 5427      22         0
## 5428       9         0
## 5429      13         0
##      Diverted
## 5424        0
## 5425        0
## 5426        0
## 5427        0
## 5428        0
## 5429        0
```

Frequency counts

Counting how often an observation occurs is one of the most common operations you will need to perform. This is not only useful for descriptive purposes but very often you will need to save your results as a new table to operate on.

The most straightforward way to do frequency counts in R is using the `table` function. Suppose we want to know how many flights departed from each airport:

```
table(hflights$Origin)
```

```
##
##    HOU    IAH
##  52299 175197
```

You can also include other variables to obtain cross-tables. Suppose you want to find out how many cancellations occurred at each airport.

```
table(hflights$Origin, hflights$Cancelled)
```

```
##
##           0      1
##   HOU  51431    868
##   IAH 173092   2105
```

You can combine the table function with other R functions, for example to find out how many missing values there are for the variable `ActualElapsedTime`, which contains information about flight duration.

```
table(is.na(hflights$ActualElapsedTime))
```

```
##
##  FALSE   TRUE
## 223874   3622
```

Say you need to create a table with frequency counts for each airline by airport and save it as a data frame called carrier_freq. You could use the function as.data.frame in combination with table.

```
carrier_table <- as.data.frame(table(Origin = hflights$Origin,
    UniqueCarrier = hflights$UniqueCarrier))
head(carrier_table)
```

```
##   Origin UniqueCarrier Freq
## 1    HOU            AA    0
## 2    IAH            AA 3244
## 3    HOU            AS    0
## 4    IAH            AS  365
## 5    HOU            B6  695
## 6    IAH            B6    0
```

There are many alternative ways to do frequency counts in R; `xtabs` is very similar to `table` but incorporates the formula notation.

```
carrier_xtabs <- as.data.frame(xtabs(~Origin +
    UniqueCarrier, data = hflights))
```

Bar plots

Bar plots are an excellent way to display frequency counts and other statistics calculated by groups. There are two ways to plot things in `ggplot2` - at the moment we will focus on `qplot` function that resembles more the base R graphics which should be more familiar to you. To produce a bar plot with `qplot` we use the option `geom = "bar"`. `geom` specifies a layer that should be plotted and there can actually be more than one layer, as we will see later on.

```
qplot(
    data = hflights,     # data source
    x = UniqueCarrier,   # column labels in x axis
    fill = Origin,       # bar fill color
    geom = "bar",        # bar plot!
    position = "dodge"   # draw bars side-by-side
)
```

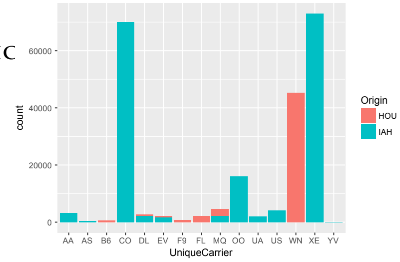

Figure 1: Carrier frequency by airport of origin.

Practice with `qplot`. Create a table that contains a ranking of carriers by percentage of flights with arrival delays (from lowest to highest), then draw a bar plot with these percentages. Can you order the bars from lowest to highest? Can you make the bars horizontal?

Centrality, extreme values, and dispersion measures

Centrality measures, such as the mean, are another way to summarize numeric data. Conveniently, R provides a `mean` function that will let you quickly find the mean of any numeric variable, such as flight duration.

```
mean(hflights$ActualElapsedTime)
```

```
## [1] NA
```

Yikes! Something went wrong. Remember this variable has some missing values so the `mean` function returns a missing value too. We have to tell mean to ignore missing values using the `na.rm` option.

```
mean(hflights$ActualElapsedTime, na.rm = TRUE)
```

```
## [1] 129.3237
```

The mean can often be affected by extreme observations (outliers), so you may want to use a more robust measure such as the median.

```
median(hflights$ActualElapsedTime, na.rm = TRUE)
```

```
## [1] 128
```

Recall the median is the value separating the higher half of your observations from the lower half. But often, you want to find out the value that defines the top quartile or the top 1% of your data. The quantile function will be helpful in such cases.

```
quantile(hflights$ActualElapsedTime, c(0.01, 0.25,
    0.5, 0.75, 0.99), na.rm = TRUE)
```

```
##  1% 25% 50% 75% 99%
##  44  77 128 165 277
```

Sometimes you want to perform the reverse operation. Say, what percentage of all flights departing from IAH are shorter than 60 minutes? The empirical conditional distribution function, `ecdf` is what you need. This function creates another function you can use to find out the answer (remember closures?).

```
fdur_ecdf <- ecdf(hflights$ActualElapsedTime)
fdur_ecdf(60)
```

```
## [1] 0.1395249
```

Quite often you want to find out what are the maximum and minimum values in your data. This is easily achieved with the max and min functions.

```
max(hflights$ActualElapsedTime, na.rm = TRUE)
min(hflights$ActualElapsedTime, na.rm = TRUE)
```

```
## [1] 575
## [1] 34
```

But what if you want to have a more complete picture of a variable without calling six or seven functions every time? The summary function is a handy way to get the most important statistics all at once.

```
summary(hflights$ActualElapsedTime)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.
##    34.0    77.0   128.0   129.3   165.0
##    Max.    NA's
##   575.0    3622
```

Finally, you may also want to find out what is the most common value in your data, the mode.[1] For some reason base R does not have a mode function but you can easily program your own.

[1] There is a `mode` function, but it does something else than you would expect.

```
Mode <- function(x) {
    ux <- unique(x[!is.na(x)])
    ux[which.max(tabulate(match(x, ux)))]
}
Mode(hflights$ActualElapsedTime)
```

```
## [1] 54
```

The mode will work with both numeric and categorical data.

```
Mode(hflights$UniqueCarrier)
```

```
## [1] "XE"
```

Box-and-whisker plots

Box-and-whisker plots summarize many of the statistics discussed above into a single figure. The upper and lower hinges of the box correspond to the first and third quartiles (the 25th and 75th percentiles). The upper whisker extends from the hinge to the highest value that is within 1.5 * IQR of the hinge. The lower whisker extends from the hinge to the lowest value within 1.5 * IQR of the hinge. Data beyond the end of the whiskers are considered outliers and plotted as points. To produce a box-and-whisker plot with `qplot` we use the option `geom = "boxplot"`.

```
qplot(
    data = hflights,         # data source
    x = Origin,              # x axis variable
    y = ActualElapsedTime,   # y axis variable
    geom = "boxplot"         # make a box plot!
)
```

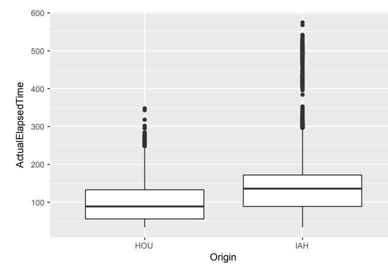

Figure 2: Flight duration by airport of origin.

Dispersion

Besides centrality, you often want to have an idea of how much variability is in your data; that's what dispersion measures are for. The most common of these are the variance and its squared root, the standard deviation, which you obtain in R with the `var` and `sd` functions.

```
var(hflights$ActualElapsedTime, na.rm = TRUE)
sd(hflights$ActualElapsedTime, na.rm = TRUE)
```

```
## [1] 3514.811
## [1] 59.28584
```

Another common measure of dispersion is the interquartile range, `IQR`, which gives you the distance between the 25% and 75% quantiles of your variable.

```
IQR(hflights$ActualElapsedTime, na.rm = TRUE)
```

```
## [1] 88
```

Practice. Compare carriers by centrality and dispersion of flight duration using boxplots.

By-group processing

Describing segments of your data and comparing them is another common task you will encounter. This is called by-group processing.

Segments should be defined by one or more of the variables in your dataset (not ones you wish to describe).

Of course you could just apply the functions discussed above (or any others) to subsets of your data that you select manually, but that is not very practical or elegant. Say you wanted to calculate average flight duration for each carrier. There are 15 carriers in the data; you don't want to call the mean function 15 times! Writing a loop that does this would also be terribly inefficient.

R has options for efficient by-group processing, for instance using `tapply`.

```
tapply(hflights$ActualElapsedTime, hflights$UniqueCarrier,
    FUN = mean, na.rm = TRUE)
```

```
##        AA        AS        B6        CO
##  92.76872 276.96978 205.44279 170.92406
##        DL        EV        F9        FL
## 126.49054 127.30693 143.17308 111.99432
##        MQ        OO        UA        US
## 116.23268 137.08054 183.44958 158.00620
##        WN        XE        YV
## 100.04592 103.86030 151.38462
```

You can use more than one variable to define your groups. Say you want to know the average departure delay by airport and day of week.

```
tapply(hflights$DepDelay, list(hflights$Origin,
    hflights$DayOfWeek), FUN = mean, na.rm = TRUE)
```

```
##             1         2         3        4
## HOU 13.184221 10.415046 12.354447 17.71489
## IAH  9.093206  6.651868  6.677232 10.83805
##             5        6         7
## HOU 13.502046 9.859795 11.889595
## IAH  8.807676 7.228887  9.205315
```

There are other, even more powerful ways to do this in R, especially when dealing with bigger data. But we leave those for the next topic.

Histograms

A histogram is a useful graphic technique to visualize the distribution of a variable. In the following figure we compare the dispersion in taxi-out time (from gate to take-off) by airport.

```
hist_data <- hflights[hflights$TaxiOut <= 30 &
                      !is.na(hflights$TaxiOut),]

qplot(
    data = hist_data,     # data source
    x = TaxiOut,          # x variable
    facets = Origin ~ ., # define grid y ~ x
    geom = "histogram",
    binwidth = 1          # bin width
)
```

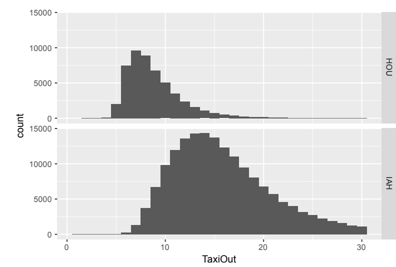

Figure 3: Taxi-out time by airport

To produce a histogram with `qplot` we use the option `geom = "histogram"`. Note the use of the facets option to create a grid of figures corresponding to each group and the binwidth option to control the width of each bar. Both the mean and dispersion appear to be much greater at IAH than HOU.

Practice. Create a table with percentage of canceled flights by month for each carrier. Paint the results using a point-and-line plot using different colors for each carrier. Do you see any patterns? Paint each airport of origin in a separate panel. Can you explain the outliers in May and August?

Correlation and linear regression

The relationship between two or more variables is our final topic for this session. Scatter plots are a useful to illustrate relationships between continuous variables. For instance, you would think that departure delays should affect arrival delays. Let's plot both variables together to see if we're right.

```
hflights_jan <- hflights[hflights$Month == 1,]
qplot(
    data = hflights_jan,     # data source
    x = DepDelay,            # x var
    y = ArrDelay,            # y var
    geom = "point"           # make scatterplot!
)
```

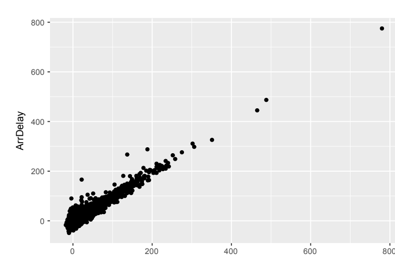

Figure 4: Relationship between departure and arrival delays during January

To produce a scatter plot with `qplot` we use the option `geom = "point"`. It looks like our intuition was right, longer delays in departure generally produce longer delays in arrival (surprise!).

Correlation lets us quantify the degree of association between two variables like in the picture above. It takes values between -1 (perfect negative correlation) and 1 (perfect positive correlation). A value of 0 means no correlation between the variables. With the function `cor` in R we can easily compute this measure.

```
cor(hflights$DepDelay, hflights$ArrDelay, use = "complete.obs")
```

```
## [1] 0.9292181
```

Say we want to uncover a relationship between two variables x and y of the form y = a + bx. Linear regression refers to a statistical technique that estimates coefficients a and b from a data sample such that the sum of squared differences between predicted values and real values of y is minimized 11 . The function lm in R allows us to estimate linear regression mod- els. Its main input is an R formula of the form y ~ x .

```
mod <- lm(ArrDelay ~ DepDelay, data = hflights)
mod
```

```
##
## Call:
## lm(formula = ArrDelay ~ DepDelay, data = hflights)
##
## Coefficients:
## (Intercept)     DepDelay
##     -2.2528       0.9928
```

Instead of using the `qplot` function for plotting we will see how to create plots through a chain of plot components. This syntax is much clearer, it is easier to debug and allows for more detailed specification of each component. `ggplot` has an amazing documentation with many example, check it out![2]

[2] docs.ggplot2.org/current/

```
ggplot(data = hflights_jan, aes(x = DepDelay,
    y = ArrDelay, color = factor(Origin))) +
geom_point(size = 2, alpha = 0.2) +
geom_smooth(method = "lm", size = 0.7, color = "darkred") +

scale_x_continuous("Departure delay", limits = c(-50,
    400), breaks = seq(-50, 400, 50)) +
scale_y_continuous("Arrival delay", limits = c(-50,
    400), breaks = seq(-50, 400, 50)) +
scale_colour_manual(name = "Airport in\nHouston",
    values = c("forestgreen", "black")) +
annotate("text", x = 50, y = 350, size = 5, label = paste0("Intercept =",
    round(mod$coefficients[1], 2), "\nSlope =",
```

```
    round(mod$coefficients[2], 2))) +
theme(panel.background = element_blank(), legend.key = element_rect(fill = "#FFFFFF"),
    legend.title = element_text(size = 12), legend.text = element_text(size = 12),
    axis.line = element_line(colour = "darkgrey",
        size = 0.5), axis.text = element_text(size = 12),
    axis.ticks = element_line(size = 0.5), axis.title.y = element_text(vjust = 1.8,
        size = 16), axis.title.x = element_text(vjust = -0.8,
        size = 16))
```

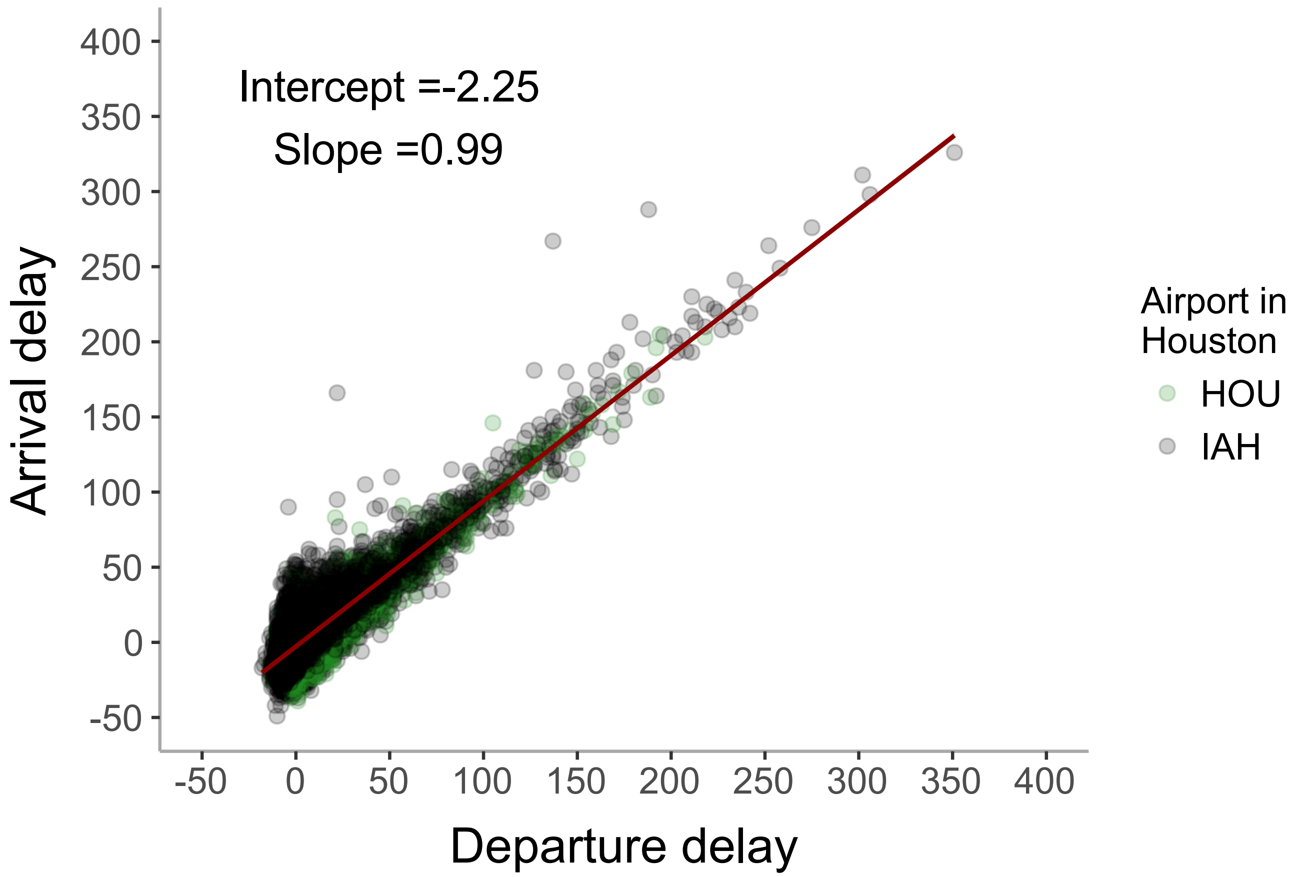

Figure 5: A bit more customized ggplot.

The result is a straight line with intercept `a = -2.25` and slope `b = 0.99`. Note how we can combine the scatter plot with the regression line by passing a vector with the options point and smooth to the geom argument.

Regression between two variables is called univariate regression. More often than not, however, we are interested in relationships between multiple variables: multivariate regression. We can easily introduce new variables by adding terms to the formula in our call to the `lm` function. For example, we can add `Origin` to test if delays can be airport-related. We can test as well if delays are affected by `Distance`, since it is likely that airlines can more easily make-up for delayed departures on longer flights. To obtain more details we use the `summary` function as follows. Note how the summary function changes the output depending on the class of the input, in this case the class of `delay_mod` is `lm`.

```
delay_mod <- lm(ArrDelay ~ DepDelay + Origin +
    Distance, data = hflights)
class(delay_mod)
summary(delay_mod)
```

```
## [1] "lm"
##
## Call:
## lm(formula = ArrDelay ~ DepDelay + Origin + Distance, data = hflights)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -68.742  -6.358  -0.867   4.983 154.212
##
## Coefficients:
##               Estimate Std. Error t value
## (Intercept) -3.372e+00  5.970e-02  -56.49
## DepDelay     9.986e-01  8.222e-04 1214.43
## OriginIAH    4.568e+00  5.751e-02   79.43
## Distance    -3.110e-03  5.317e-05  -58.48
##             Pr(>|t|)
## (Intercept)   <2e-16 ***
## DepDelay      <2e-16 ***
## OriginIAH     <2e-16 ***
## Distance      <2e-16 ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 11.15 on 223870 degrees of freedom
##   (3622 observations deleted due to missingness)
## Multiple R-squared:  0.8682, Adjusted R-squared:  0.8682
## F-statistic: 4.916e+05 on 3 and 223870 DF,  p-value: < 2.2e-16
```

Indeed, it seems that flying from IAH adds on average 4.56 minutes of delay, while airlines can reduce delays by 0.003 minutes per trip mile. The column with p-values (Pr(>|t|)) indicates that all three variables are highly statistically significant. The R-squared statistic indicates that our model explains about 86% of the variation in arrival delay times. Hoorray!

Practice. Find out if there is a statistically significant relationship between flight distance and aircraft speed. Do planes move faster or slower in long-distance flights? Do certain airlines seem to use faster planes? Use regression as well as graphical methods to justify your answer.

Data wrangling with `dplyr` and `tidyr`

If you think data scientists spend most of their time running regressions and machine learning algorithms or generating stunning data visualizations, think again. That part usually comes only at the end of a laborious process of cleaning, transforming, and combining tables needed to get data ready for analysis, a process often referred to as data wrangling or data janitoring. Some people claim this kind of work can take up to 50-80% of a data scientist's time.[3]

[3] nyti.ms/1t8IzfE

It is essential then to have the tools that make this janitor work as efficient as possible. For this session we will introduce you to some of these cutting edge tools, which will hopefully save you time writing code as well as processing it.

We will again use the data of all flights departing from major Houston airports in 2011.

The dplyr package

In my experience using R, there is a time before and after the discovery of this library developed by Hadley Wickham.

```
library(dplyr)
```

Find our more about `dplyr`:

```
vignette("introduction", package = "dplyr")
```

Getting quick summaries, broken according to certain groups perhaps, is surprisingly laborious in R. Moreover, syntax is such that there is a lot of clutter and execution is actually rather slow. `dplyr` syntax is extremely clean and more importantly, lots of code was written in low level languages to speed up the execution. As a result, you can do things with fewer lines of code than you would normally write in R, and the performance gains in sorting, subsetting, merging, and by-group processing are huge. Also, it connects really well with other useful packages developed by the same team, such as `tidyr`, `reshape` or `ggplot2`.[4]

[4] An alternative package that has similar performance, but more difficult syntax (it corresponds less with the usual R syntax), is `data.table`.

The tbl_df wrapper. This optional but convenient wrapper for data frames will keep you from accidentally printing lots of data to the screen. In most cases when you use `dplyr` functions, your data frame will be first passed through this wrapper. If you really want to print the whole data frame, you can use the function `print.data.frame`.

```
hflights_df <- tbl_df(hflights)
hflights_df
```

```
## # A tibble: 227,496 x 21
##     Year Month DayofMonth DayOfWeek DepTime
##   * <int> <int>      <int>     <int>   <int>
##   1  2011     1          1         6    1400
##   2  2011     1          2         7    1401
##   3  2011     1          3         1    1352
##   4  2011     1          4         2    1403
##   5  2011     1          5         3    1405
##   6  2011     1          6         4    1359
##   7  2011     1          7         5    1359
##   8  2011     1          8         6    1355
##   9  2011     1          9         7    1443
## 10  2011     1         10         1    1443
## # ... with 227,486 more rows, and 16 more
## #   variables: ArrTime <int>,
## #   UniqueCarrier <chr>, FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>,
## #   AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>,
## #   Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>,
## #   Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

Sort and subset data

`dplyr` functions correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code. To select certain rows from your table use the function `filter`.

```
system.time(dptest <- filter(hflights_df, Dest ==
    "BPT"))
dptest
print.data.frame(dptest)
```

```
##    user  system elapsed
##   0.002   0.001   0.004
## # A tibble: 3 x 21
##    Year Month DayofMonth DayOfWeek DepTime
##   <int> <int>      <int>     <int>   <int>
## 1  2011     2         23         3     947
## 2  2011     3          1         2     921
## 3  2011     3          2         3     905
## # ... with 16 more variables: ArrTime <int>,
## #   UniqueCarrier <chr>, FlightNum <int>,
## #   TailNum <chr>, ActualElapsedTime <int>,
## #   AirTime <int>, ArrDelay <int>,
## #   DepDelay <int>, Origin <chr>,
## #   Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>,
## #   Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
##   Year Month DayofMonth DayOfWeek DepTime
## 1 2011     2         23         3     947
## 2 2011     3          1         2     921
## 3 2011     3          2         3     905
##   ArrTime UniqueCarrier FlightNum TailNum
## 1    1030            XE      2204  N17928
## 2    1015            XE      2204  N14940
## 3    1001            XE      2204  N14943
##   ActualElapsedTime AirTime ArrDelay
## 1                43      22       29
## 2                54      30       14
## 3                56      29        0
##   DepDelay Origin Dest Distance TaxiIn
## 1       32    IAH  BPT       79      5
## 2        6    IAH  BPT       79      5
## 3      -10    IAH  BPT       79      5
##   TaxiOut Cancelled CancellationCode
## 1      16         0
## 2      19         0
## 3      22         0
##   Diverted
## 1        0
## 2        0
## 3        0
```

Compare it with the traditional R function and syntax.

```
system.time(dptest2 <- hflights[hflights$Dest ==
    "BPT", ])
dptest2

##    user  system elapsed
##   0.003   0.000   0.004
##         Year Month DayofMonth DayOfWeek
## 683596  2011     2         23         3
## 1441066 2011     3          1         2
## 1441067 2011     3          2         3
##         DepTime ArrTime UniqueCarrier
## 683596      947    1030            XE
## 1441066     921    1015            XE
## 1441067     905    1001            XE
##         FlightNum TailNum ActualElapsedTime
## 683596       2204  N17928                43
## 1441066      2204  N14940                54
## 1441067      2204  N14943                56
##         AirTime ArrDelay DepDelay Origin
## 683596       22       29       32    IAH
## 1441066      30       14        6    IAH
## 1441067      29        0      -10    IAH
##         Dest Distance TaxiIn TaxiOut
## 683596   BPT       79      5      16
## 1441066  BPT       79      5      19
## 1441067  BPT       79      5      22
##         Cancelled CancellationCode Diverted
## 683596          0                         0
## 1441066         0                         0
## 1441067         0                         0
```

To select certain columns use the function `select`.

```
dptest <- select(dptest, UniqueCarrier, Origin,
    Dest, Year:DayofMonth)
dptest

## # A tibble: 3 x 6
##   UniqueCarrier Origin  Dest  Year Month
##           <chr>  <chr> <chr> <int> <int>
## 1            XE    IAH   BPT  2011     2
## 2            XE    IAH   BPT  2011     3
## 3            XE    IAH   BPT  2011     3
## # ... with 1 more variables:
## #   DayofMonth <int>
```

To sort your table according to the values of a variable use the function `arrange` . Use `desc` to sort in descending order.

```
arrange(dptest, desc(Month), desc(DayofMonth))
```

```
## # A tibble: 3 x 6
##   UniqueCarrier Origin  Dest  Year Month
##           <chr>  <chr> <chr> <int> <int>
## 1            XE    IAH   BPT  2011     3
## 2            XE    IAH   BPT  2011     3
## 3            XE    IAH   BPT  2011     2
## # ... with 1 more variables:
## #   DayofMonth <int>
```

Modify and create new columns

To add new columns that are functions of existing columns, use `mutate`.

```
mutate(dptest, newVar = (Month + DayofMonth)/2)
```

```
## # A tibble: 3 x 7
##   UniqueCarrier Origin  Dest  Year Month
##           <chr>  <chr> <chr> <int> <int>
## 1            XE    IAH   BPT  2011     2
## 2            XE    IAH   BPT  2011     3
## 3            XE    IAH   BPT  2011     3
## # ... with 2 more variables:
## #   DayofMonth <int>, newVar <dbl>
```

The same function also works to modify existing columns.

```
mutate(dptest, Origin = tolower(Origin))
```

```
## # A tibble: 3 x 6
##   UniqueCarrier Origin  Dest  Year Month
##           <chr>  <chr> <chr> <int> <int>
## 1            XE    iah   BPT  2011     2
## 2            XE    iah   BPT  2011     3
## 3            XE    iah   BPT  2011     3
## # ... with 1 more variables:
## #   DayofMonth <int>
```

Summarizing data

One of the most useful functions is `summarize` that collapses a data frame to a single row.

```
summarize(hflights_df, dep_delay = mean(DepDelay,
    na.rm = TRUE), arr_delay = mean(ArrDelay,
    na.rm = TRUE))

## # A tibble: 1 x 2
##   dep_delay arr_delay
##       <dbl>     <dbl>
## 1  9.444951  7.094334
```

In dplyr, you use the group_by function to describe how to break a dataset down into groups of rows. You can then proceed to operate by group.

```
origin_day <- group_by(hflights_df, Origin, DayOfWeek)
delays <- summarize(origin_day, dep_delay = mean(DepDelay,
    na.rm = TRUE), arr_delay = mean(ArrDelay,
    na.rm = TRUE))
delays

## Source: local data frame [14 x 4]
## Groups: Origin [?]
##
## # A tibble: 14 x 4
##    Origin DayOfWeek dep_delay arr_delay
##     <chr>     <int>     <dbl>     <dbl>
##  1    HOU         1 13.184221  8.118392
##  2    HOU         2 10.415046  5.526221
##  3    HOU         3 12.354447  7.304571
##  4    HOU         4 17.714889 12.722579
##  5    HOU         5 13.502046  8.185185
##  6    HOU         6  9.859795  3.555811
##  7    HOU         7 11.889595  5.687796
##  8    IAH         1  9.093206  8.296455
##  9    IAH         2  6.651868  5.560299
## 10    IAH         3  6.677232  4.949363
## 11    IAH         4 10.838052  8.934377
## 12    IAH         5  8.807676  7.027071
## 13    IAH         6  7.228887  6.318096
## 14    IAH         7  9.205315  7.292854
```

Chaining operations

Probably my favorite feature of the package is the %>% operator. Remeber piping operation from shell? This oeprator does exactly the same - it sends the result of one operation as the first argument to the next. This lets you perform multiple operations at once and the resulting code is very readable.[5]

For example, say we want to get a ranking of average departure delay times by carrier flying from IAH. You could do:

```
exmpl <- hflights_df %>%  # data source
    filter(Origin == "IAH") %>%  # subset rows
    select(UniqueCarrier, DepDelay) %>%  # subset columns
    group_by(UniqueCarrier) %>%  # group by carrier
    summarize(dep_delay =
        round(mean(DepDelay, na.rm = TRUE),2)) %>%
    arrange(dep_delay)

head(exmpl)
```

```
## # A tibble: 6 x 2
##   UniqueCarrier dep_delay
##           <chr>     <dbl>
## 1            YV      1.54
## 2            US      1.62
## 3            AS      3.71
## 4            AA      6.39
## 5            XE      7.71
## 6            OO      8.89
```

[5] The package `magrittr` that developed it actually implements this operator for any R function. Using it wisely may decrease development time and improve readability and maintainability of code.

Practice

Create a table that contains the following information by carrier:

- Percentage of flights with arrival delays.
- Percentage of cancelled flights.
- Average speed for short (< 3 hours) and medium-long haul flights (>= 3 hours).

Try to do it with base R and do some performance comparisons that demonstrate the speed of the package. You can use `system.time` function, but also check `microbenchmark` package.

The tidyr package

Reshaping data is another common data janitoring task. Currently, the best tools in R for this task come from Hadley Wickham's recent `tidyr` package. It follows a concept of **tidy data**, advocated by Wickham, which consists of a set of rules that should be followed when creating data sets. Having datasets in this format would facilitate data analysis and plotting, and decrease the amount of time spent on data wrangling.

Learn more about the `tidyr` package and tidy data with the vignette. It is well worth reading the paper on tidy data by Hadley Wickham.[6]

[6] Link to the paper: jstatsoft.org/v59/i10/paper

```
vignette("tidy-data", "tidyr")
```

This package focuses on reshaping **data frames**.[7] To illustrate how it works, let's first create a table that has total flying time per month and tail number (aircraft ID) from January to March.

[7] For reshaping other classes of R objects see R base's `reshape` function or Wickham's previous reshaping packages `reshape` and `reshape2`.

```
library(tidyr)
last_month <- 3
fdur <- hflights_df %>% filter(AirTime > 0 & Month %in%
    1:last_month) %>% group_by(Month, TailNum) %>%
    summarize(total_airtime = round(sum(AirTime)))

print(fdur)

## Source: local data frame [5,762 x 3]
## Groups: Month [?]
##
## # A tibble: 5,762 x 3
##    Month TailNum total_airtime
##    <int>   <chr>         <dbl>
## 1      1  N0EGMQ           370
## 2      1  N10156          3542
## 3      1  N11106          1323
## 4      1  N11107          1312
## 5      1  N11109          2245
## 6      1  N11113          1789
## 7      1  N11119          3200
## 8      1  N11121          3280
## 9      1  N11127          3050
## 10     1  N11137          2837
## # ... with 5,752 more rows
```

`fdur` has 5762 observations and 3 columns.

Wider data

We say you make your data **wider** when you increase the number of columns and decrease the number of rows, while keeping information constant. Use the `spread` function to make data wider.

```
fdur_wide <- spread(
    fdur,          # table to make wider
    Month,         # key defining columns
    total_airtime # values to fill columns
)
names(fdur_wide)[2:4] <-
    paste0(month.abb[1:last_month],"-2011")

print(fdur_wide)

## # A tibble: 2,515 x 4
##    TailNum 'Jan-2011' 'Feb-2011' 'Mar-2011'
##  *   <chr>      <dbl>      <dbl>      <dbl>
##  1  N0EGMQ        370        697         NA
##  2  N10156       3542       2279       2196
##  3  N11106       1323       1371       3345
##  4  N11107       1312       2530       1108
##  5  N11109       2245       1975       3867
##  6  N11113       1789       2228       3991
##  7  N11119       3200        975         NA
##  8  N11121       3280       2435       3772
##  9  N11127       3050       1694       4053
## 10  N11137       2837       2612       2353
## # ... with 2,505 more rows
```

`spread` has created a column for each month, the new table has 2515 observations and 4 columns. Note that if an aircraft has no records for a month, `spread` produces a missing value (NA).

Longer data

We say you make your data **longer** whenever you reduce the number of columns and increase the number of rows in your data, while keeping the same amount of information. Use the `gather` function to make data longer.

```
fdur_long <- gather(
    fdur_wide,      # table to make longer
    Month,          # new column of observation ids
    total_airtime, # new column of observation values
    -TailNum,       # exclude variable from gathering
    na.rm = TRUE    # remove missing values
)
print(fdur_long)
```

```
## # A tibble: 5,762 x 3
##    TailNum    Month total_airtime
##  *   <chr>    <chr>         <dbl>
##  1  N0EGMQ Jan-2011           370
##  2  N10156 Jan-2011          3542
##  3  N11106 Jan-2011          1323
##  4  N11107 Jan-2011          1312
##  5  N11109 Jan-2011          2245
##  6  N11113 Jan-2011          1789
##  7  N11119 Jan-2011          3200
##  8  N11121 Jan-2011          3280
##  9  N11127 Jan-2011          3050
## 10  N11137 Jan-2011          2837
## # ... with 5,752 more rows
```

`fdur_long` has the same number of records (5762) and columns (3) as the original table, `fdur`.

Separating

This package has one additional function to split columns that contain two variables, a problem often encountered by data scientists. For instance, the column Month in the `fdur_long` table we just created actually has information for month and year and maybe we would like to separate this information into two different variables.

```
separate(
    fdur_long,  # data
    Month,      # column to separate
    into = c("Month", "Year"), # new columns
    sep = "-"   # separating expression (regex)
)
```

```
## # A tibble: 5,762 x 4
##    TailNum Month  Year total_airtime
##  *   <chr> <chr> <chr>         <dbl>
##  1  N0EGMQ   Jan  2011           370
##  2  N10156   Jan  2011          3542
##  3  N11106   Jan  2011          1323
##  4  N11107   Jan  2011          1312
##  5  N11109   Jan  2011          2245
##  6  N11113   Jan  2011          1789
##  7  N11119   Jan  2011          3200
##  8  N11121   Jan  2011          3280
##  9  N11127   Jan  2011          3050
```

```
## 10  N11137   Jan  2011         2837
## # ... with 5,752 more rows
```

Chaining operations

If you load the `dplyr` package, then you can also use the `%>%` operator with `tidyr` and combine operations of both packages, for example

```
fdur %>% filter(TailNum == "N0EGMQ") %>% spread(Month,
    total_airtime)
```

```
## # A tibble: 1 x 3
##   TailNum   '1'   '2'
## *   <chr> <dbl> <dbl>
## 1  N0EGMQ   370   697
```

Reproducibility and R-markdown

R Markdown is a format that lets you easily create dynamic documents (like this one!) combining easy-to-write plain text format (minimally marked style called markdown[8]) with chunks of R code that get re-run each time you compile the document. You can choose to print the code in these chunks, the output of your code, or both. It is powered by `knitr` and `rmarkdown` packages in R, and `Pandoc` command line tools.

R markdown makes it really easy to produce documents using this format and turn them into **html**, **pdf**, or **word** documents. You can also make some really cool presentations.

The RStudio folks have created a great website that will have you learn the basics in no-time so you can start creating reports and doing your homework using R Markdown.[9] You can also start by examining the `.Rmd` files that I have used for creating these handouts.

[8] Markdown was originally developed to support simple writing that is easy to convert to HTML and nowadays is a web standard. Check the creators website, it has nice overview of the markdown features - at daringfireball.net/projects/markdown/. Another important piece of the puzzle is a command line tool called Pandoc. Exactly because of markdown simplicity, it is possible to convert it to many other formats, HTML, Word and pdf being only few of many. Check excellent Pandoc's website at pandoc.org/.

[9] rmarkdown.rstudio.com/

Why you should be a fan of R markdown?

- **Reproducibility**:
  - It makes your data analysis more reproducible. The R code describes exactly the steps from the raw data to the final report. This makes it perfect for sharing reports with your colleagues.
  - It is written with almost no formatting at all (markdown), which makes it easier to convert to any other format, from nicely looking PDFs to the all-present MS docx and complete HTML documents (fancy a blog?).
- **Efficiency**:

- Statistical output from figures to tables is automatically placed in your report. No more copy-pasting and reformatting the output from your statistical analysis program into your report.
- You want to use a slightly different subset of the data? You want to drop that outlier observation? No problem, you can update your report with a single click instead of updating every table and figure.
- Whoever has done some copy-pasting knows how easy is to overlook one number or one figure. This type of document significantly reduces the chance of such errors.

- **Education & Communication**:
  - Excellent for teaching as one can check how exactly is some analysis done from the report.
  - Do not disregard this aspect, look at Github and Stackoverflow stars who get job offers on this account!

A quick start

RStudio has a great support for authoring documents in R markdown, with very neat collection of buttons that give you quick access to various outputs. Clicking on File > New File > R Markdown also provides a basic template.

```
---
title: "Untitled"
output:
  html_document:
    fig_caption: yes
    highlight: pygments
    keep_md: yes
    number_sections: yes
    theme: cosmo
    toc: yes
---

# A title

This is an R Markdown document. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS Word
documents.

You can embed an R code chunk within the text like this:
some result =2, and a chunk that would
produce a standalone result like this:
```

```
```r
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00
```

You can also embed plots, for example:

\includegraphics{/home/hstojic/Teaching/BGSE_DS_ITC_2017/handouts/handout_Rstats_files/figure-latex/unname

Note that the `echo = FALSE` parameter was added to
the code chunk to prevent printing of the R code that
generated the plot.
```

How to render files if you are not using RStudio? Navigate to a folder with `.Rmd` file and

```
rmarkdown::render("input.Rmd")
rmarkdown::render("input.Rmd", "pdf_document")
```

Debugging

Identifying and correcting code bugs is quite common, even for experienced programmers. Debugging a series of separated code chunks may be easy - just execute each chunk and identify the line that has the error. But debugging functions and loops may be difficult, so we need some tools to make it easier.

Old-school debugging

The most immediate way of debugging a function or loop is printing out to the console some information of the process. The `print` function outputs into the console once a process is executed. As an alternative to `print` you can use `cat` function, however `cat` cannot easily handle some types of outputs, such as matrices etc.

```
cumSum <- 0
for (iteration in 1:10) {
    cumSum <- cumSum + iteration
    print(paste("Iteration ", iteration,
          ". Cumulative sum: ", cumSum),
          sep = "")
}
```

Built-in R and RStudio utilities

R has a built-in facility for single stepping through R functions, and for examining variables during execution.[10]

RStudio includes a visual debugger that can help you understand the code and find bugs.[11] The debugger includes the following features:

- editor breakpoints, both inside and outside functions;
- code and local environment visualization during debugging;
- debug stepping tools (next, continue, etc.);
- deep integration with traditional R debugging tools, such as `browser` and `debug`.

[10] Debugging in R using `debug` , `browser` and `trace` functions `http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR/`

[11] Debugging with RStudio: `https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio`

Accessing R from terminal

When running computations in R on servers, your scripts have to be standalone code, i.e. they should not require any user input. Once you transfer you scripts to the server you have to execute them there from the command line. Recall that you should use `screen` tool as well, to prevent termination of the computation due to termination of SSH connection[12].

There are two ways to execute an R script from the command line.

1. R CMD BATCH - experts advise against using this command
2. Rscript - recommended way

By default Rscript[13] command will not produce a file with the output, instead it will print it in the terminal, but you can instruct it to redirect it to a file. If you add shebang, `#!/usr/bin/Rscript`, at the top of your script, then you can also run it as you would run a shell script.

[12] There are alternatives to `screen`, see for example tmux

[13] There are many options that you could additionally use, like `--vanilla` to use basic R, check the help docs from the command line, `Rscript --help`.

```
Rscript script.R
Rscript script.R > log.txt

# Check the output
cat log.txt

# if shebang is added
./script.R
```

R CMD BATCH command will automatically create a new file called script.Rout with the output.

```
R CMD BATCH script.R

# Check the output
cat script.Rout
```

Note that the output refers to the output that you would see in normal R console were you to run the same code interactively. The data, results or figures that you specifically save to the disk in that script should be on the disk, if the code ran successfully (no error occurred).

Passing in arguments from terminal

Sometimes you want you R script to be able to accept an argument when it is run from terminal. This is often the case if you have a shell script that describes your whole pipeline. For example, in a step before calling the script you might transform the dataset in some ways directly from shell with `awk` and the new dataset needs and potentially some variables have to be passed to the script as inputs.

There are some useful posts on StackOverflow and for more details see `help(commandArgs)`.

Let us create a small shell script that will illustrate this, `shellScript.sh`.

```
#!/bin/bash

# shell creates some variable
date=`date --date=-1day +%Y-%m-%d`

# then we run the R script and pass in the variables
Rscript script.R $date 10 40 training > log.txt
```

Now we are only missing an example of an R script that accepts inputs from shell, `script.R`.

```r
# tell R to receive arguments passed by the shell
options(echo=TRUE)
args <- commandArgs(trailingOnly = TRUE)

# lets see what we have passed
print(args)

date <- as.Date(args[1])
par1 <- as.numeric(args[2])
par2 <- as.numeric(args[3])
name <- args[4]

# save some processing
x <- rnorm(1000, mean=par1, sd=par2)
pdf(paste(name,".pdf", sep=""))
plot(x)
title(main=as.character(date))
dev.off()

summary(x)
```

Using `> log.txt` in the `shellScript.sh` file saves the output in an external file called `log.txt`. If you leave it out, it will simply print it on the terminal. Before running the shell script by calling `./shellScript.sh` remember to make it executable with `chmod +x shellScript.sh`.

littler

Sometimes you would want a simple command line interface to R, so that you can execute R commands in an easy fashion, perhaps use in piping etc. For these purposes you can use `littler`[14] To install it on Linux machines, type in your terminal

[14] See littler website.

```bash
sudo apt-get install littler
```

Some example

```bash
echo 'cat(pi^2,"\n")' | r
```

it is possible to call R from shell and send this input to it, however it will start a whole R session and produce a lot of irrelevant output in the terminal. Compare the output from the command above with this one:

```
R -e 'cat(pi^2,"\n")'
```

Interactive vizualization with Shiny

This sections focuses on dynamic visualization with the help R package Shiny. Figures are nice, but we can go further, we can illustrate our results in interactive web applications.

1. If you intend to work as a data scientist, as a rule you will have to report to others.
2. Our visual sensory system has the largest "bandwidth" of all our senses. You should leverage it to convey your messages better. With interactive visualization you can transmit much more information, but you can also influence to which aspects of your visualization the audience should pay attention to. It is much more than simple engineering.[15]
3. Besides improving communication with our audience, interactive visualizations can improve our own understanding of the problem and the methods.
4. It's fun and eye catching!

[15] see excellent books by Edward Tufte on how to present visual information!

Why Shiny? - You need only R!

- For interactive visualizations nowadays you would use HTML and Javascript - Shiny hides almost all of it (although a bit of HTML/JS knowledge can come handy)!

Hello World! example

There is a nice gallery of examples on the shiny website and they are a good starting point. At the moment you can get some slots on Shinyapps server server for free but with some contraints, so that is not a long term solution if you would use it a lot. You can install Shinyapps on your server, but you will need to learn a bit about servers etc.

First make sure you have the following packages installed: `shiny`. If you want to deploy apps on RStudio's Shinyapps server, you will need some additional packages.

```
install.packages("shiny")
```

Web applications in shiny usually consist of two files: `ui.R` and `server.R`. It can be done in a single file, which can come handy if you want to make a Shiny object a part of your `.Rmd` file.

Best way to learn it is to take a look at an example. These are the ui.R and server.R from the "K means" example.

ui.R

```
pageWithSidebar(

    headerPanel('Iris k-means clustering'),

    sidebarPanel(
        selectInput('xcol', 'X Variable', names(iris)),
        selectInput('ycol', 'Y Variable', names(iris),
                    selected=names(iris)[[2]]),
        numericInput('clusters', 'Cluster count', 3,
                     min = 1, max = 9)
    ),

    mainPanel(
        plotOutput('plot1')
    )

)
```

- `pageWithSidebar()` is one of the types of user interface layouts, there are many more of them, check the Shiny reference page. A lot of these actually build on Boostrap designed HTML classes.
- user interface will usually consist of a part where a user provides an input (in this user layout it is sidebarPanel()), and a part where output produced by server.R is shown (here mainPanel())
- names of inputs and outputs have to be unique (here we have "xcol", "ycol", "clusters" and "plot1") - names are actually "id" tags in HTML, they serve as unique identifiers of crucial HTML elements

server.R

```
function(input, output, session) {

    # Combine the selected variables into a new data frame
    selectedData <- reactive({
        iris[, c(input$xcol, input$ycol)]
    })

    clusters <- reactive({
        kmeans(selectedData(), input$clusters)
    })

    output$plot1 <- renderPlot({
```

```
        palette(c("#E41A1C", "#377EB8", "#4DAF4A", "#984EA3",
          "#FF7F00", "#FFFF33", "#A65628", "#F781BF", "#999999"))

        par(mar = c(5.1, 4.1, 0, 1))
        plot(selectedData(),
             col = clusters()$cluster,
             pch = 20, cex = 3)
        points(clusters()$centers, pch = 4, cex = 4, lwd = 4)
    })

}
```

- function `function(input, output, session) {...}` defines the computations done on the server side, which produce the output shown on the user interface, it has to be included in the `server.R` file.
- `input` argument to the function is a list by which all the user inputs are forwarded to the server part, e.g. user specified number of clusters, which is captured by `clusters` variable in the `ui.R`, so the server can access this value through `input$clusters`.

Running locally

Within R we can launch an app locally by navigating first to the directory where our application is located and then running the following command: `shiny::runApp()`. This will open the application in a browser or simply give you an URL (you should see something like 127.0.0.1:1234, this is the URL pointing to your own computer). In RStudio you get a nice button called "run App" once you open server.R or ui.R, and it does everything automatically.

After opening it in a browser try opening the "developer tools" or "inspect element" in your browser and examine the HTML itself. You will notice that Shiny heavily relies on Boostrap. Boostrap is a Javascript library aimed at cross-platform compatibility and adjustment to various devices that are nowadays used for accessing the internet - desktops, tablets, mobile phones.

Deploying publicly

To deploy it publicly, you will need to set up an account on Shinyapps or install Shiny Server on your own server. I'll let you explore this topic on your own.

"Alternatives" to Shiny

1. plotly - Language independent data visualization, together with hosting and data. Aimed at achieving the ideal of reproducibility.
2. D3 - Powerful Javascript library for interactive graphics in HTML. It is quite versatile, but there is a rather steep learning curve to get to the basic level. There are some R packages that create basic type of interactive graphics in D3, take a look at the R2D3.
3. Google Charts - famous Hans Rosling TED talk few years ago featured moving charts that nicely illustrated evolution of some indicators over time (e.g. infant mortality and GDP in the world over time). Finally, Google bought the visualization libraries and improved them. There is an R package that facilitated creating such charts with R - googleVis, and it can be used within Shiny as well (albeit not without issues).