Ulrich Matter

Big Data Analytics

To the students who have motivated and inspired this work.

Contents

List of Tables

List of Figures

Preface

This textbook introduces the reader to the concept of Big Data in the context of empirical economic research. The book covers the computational constraints underlying Big Data Analytics and how to handle them in the statistical computing environment R (local and in the cloud). Revisiting basic statistical/econometric concepts, the book looks at each step of dealing with large data sets in empirical economic research (storage/import, transformation, visualization, aggregation).

FIGURE 1: Creative Commons License

Ulrich Matter St. Gallen, Switzerland

[1] http://creativecommons.org/licenses/by-nc-sa/4.0/

1

Introduction

1.1 Example of Computation Time and Memory Allocation

1.1.1 Preparation

We first read the `economics` data set into R and extend it by duplicating its rows in order to get a slightly larger data set (this step can easily be adapted to create a very large data set).

```
# read dataset into R
economics <- read.csv("data/economics.csv")
# have a look at the data
head(economics, 2)
```

```
##         date   pce    pop psavert uempmed unemploy
## 1 1967-07-01 507.4 198712    12.5     4.5     2944
## 2 1967-08-01 510.5 198911    12.5     4.7     2945
```

```
# create a 'large' dataset out of this
for (i in 1:3) {
     economics <- rbind(economics, economics)
}
dim(economics)
```

```
## [1] 4592    6
```

1.1.2 Naïve Approach (ignorant of R)

The goal of this code example is to compute the real personal consumption expenditures, assuming that `pce` in the `economics` data set

provides the nominal personal consumption expenditures. Thus, we divide each value in the vector `pce` by a deflator `1.05`.

The first approach we take is based on a simple `for`-loop. In each iteration one element in `pce` is divided by the `deflator` and the resulting value is stored as a new element in the vector `pce_real`.

```
# Naïve approach (ignorant of R)
deflator <- 1.05 # define deflator
# iterate through each observation
pce_real <- c()
n_obs <- length(economics$pce)
for (i in 1:n_obs) {
  pce_real <- c(pce_real, economics$pce[i]/deflator)
}

# look at the result
head(pce_real, 2)
```

```
## [1] 483.2 486.2
```

How long does it take?

```
# Naïve approach (ignorant of R)
deflator <- 1.05 # define deflator
# iterate through each observation
pce_real <- list()
n_obs <- length(economics$pce)
time_elapsed <-
     system.time(
         for (i in 1:n_obs) {
              pce_real <- c(pce_real, economics$pce[i]/deflator)
})

time_elapsed
```

```
##    user  system elapsed
##   0.079   0.012   0.091
```

Assuming a linear time algorithm ($O(n)$), we need that much time for one additional row of data:

```
time_per_row <- time_elapsed[3]/n_obs
time_per_row
```

```
##   elapsed
## 1.982e-05
```

If we deal with big data, say 100 million rows, that is

```
# in seconds
(time_per_row*100^4)
```

```
## elapsed
##    1982
```

```
# in minutes
(time_per_row*100^4)/60
```

```
## elapsed
##   33.03
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
## elapsed
##  0.5505
```

Can we improve this?

1.1.3 Improvement 1: Pre-allocation of memory

In the naïve approach taken above, each iteration of the loop causes R to re-allocate memory because the number of elements in vector `pce_element` is changing. In simple terms, this means that R needs to execute more steps in each iteration. We can improve this with a sim-

ple trick by initiating the vector in the right size to begin with (filled with NA values).

```
# Improve memory allocation (still somewhat ignorant of R)
deflator <- 1.05 # define deflator
n_obs <- length(economics$pce)
# allocate memory beforehand
# Initiate the vector in the right size
pce_real <- rep(NA, n_obs)
# iterate through each observation
time_elapsed <-
     system.time(
         for (i in 1:n_obs) {
              pce_real[i] <- economics$pce[i]/deflator
})
```

Let's see if this helped to make the code faster.

```
time_per_row <- time_elapsed[3]/n_obs
time_per_row
```

```
##   elapsed
## 1.089e-06
```

Again, we can extrapolate (approximately) the computation time, assuming the data set had millions of rows.

```
# in seconds
(time_per_row*100^4)
```

```
## elapsed
##   108.9
```

```
# in minutes
(time_per_row*100^4)/60
```

```
## elapsed
```

```
##   1.815
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
## elapsed
## 0.03025
```

This looks much better, but we can do even better.

1.1.4 Improvement 2: Exploit vectorization

In this approach, we exploit the fact that in R 'everything is a vector' and that many of the basic R functions (such as math operators) are *vectorized*. In simple terms, this means that a vectorized operation is implemented in such a way that it can take advantage of the similarity of each of the vector's elements. That is, R only has to figure out once how to apply a given function to a vector element in order to apply it to all elements of the vector. In a simple loop, R has to go through the same 'preparatory' steps again and again in each iteration, this is time-intensive.

In this example, we specifically exploit that the division operator `/` is actually a vectorized function. Thus, the division by our `deflator` is applied to each element of `economics$pce`.

```
# Do it 'the R way'
deflator <- 1.05 # define deflator
# Exploit R's vectorization!
time_elapsed <-
     system.time(
     pce_real <- economics$pce/deflator
          )
# same result
head(pce_real, 2)
```

```
## [1] 483.2 486.2
```

Now this is much faster. In fact, `system.time()` is not precise enough to capture the time elapsed...

```
time_per_row <- time_elapsed[3]/n_obs

# in seconds
(time_per_row*100^4)
```

```
## elapsed
##       0
```

```
# in minutes
(time_per_row*100^4)/60
```

```
## elapsed
##       0
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
## elapsed
##       0
```

In order to measure the improvement, we use `microbenchmark::microbenchmark()` to measure the elapsed time in microseconds (millionth of a second).

```
library(microbenchmark)
# measure elapsed time in microseconds (avg.)
time_elapsed <-
  summary(microbenchmark(pce_real <- economics$pce/deflator))$mean

# per row (in sec)
time_per_row <- (time_elapsed/n_obs)/10^6
```

Now we get a more precise picture regarding the improvement due to vectorization (again, assuming 100 million rows):

```
# in seconds
(time_per_row*100^4)
```

```
## [1] 0.1911
```

```
# in minutes
(time_per_row*100^4)/60
```

```
## [1] 0.003185
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
## [1] 5.308e-05
```

1.1.4.1 What do we learn from this?

1. How R allocates and deallocates memory can have a substantial effect on computation time.
    - (Particularly, if we deal with a large data set!)
2. In what way the computation is implemented can matter a lot for the time elapsed.
    - (For example, loops vs. vectorization/apply)

2

Computing Resources

2.1 Components of a standard computing environment

Figure 2.1 illustrates the key components of a standard computing environment to process digital data. In our case, these components serve the purpose of computing a statistic, given a large data set as input.

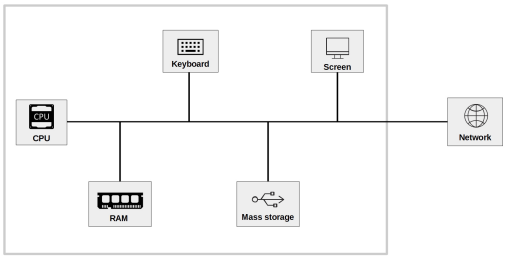


FIGURE 2.1: Basic components of a standard computing environment.

- The component actually *processing* data is the Central Processing Unit (CPU). When using R to process data, R commands are translated into complex combinations of a small set of basic operations which the *CPU* then executes.
- In order to work with data (e.g., in R), it first has to be loaded into the *memory* of our computer. More specifically, into the Random Access Memory (*RAM*). Typically, data is only loaded in the RAM as long as we work with it.
- *Mass Storage* refers to the type of computer memory we use to store data in the long run. This is what we call the *hard drive* or *hard disk*. In

these days, the relevant hard disk is actually often not the one physically built into our computer but a hard disk 'in the cloud' (built into a server to which we connect over the Internet).

Very simply put, the difference between 'data analytics' and 'Big Data analytics' is that in the latter case, the standard usage of one or several of these components fails or works very inefficiently because the amount of data overwhelms its normal capacity.

From this hardware-perspective, there are two basic strategies to cope with the situation that one of these components is overwhelmed by the amount of data:

- *Scale up ('horizontal scaling')*: Extend the physical capacity of the affected component by building a system with large RAM shared between applications. This sounds like a trivial solution ('if RAM is too small, buy more RAM…'), but in practice it can be very expensive.
- *Scale out ('vertical scaling')*: Distribute the workload over several computers (or separate components of a system).

From a software-perspective, there are many (context-specific) strategies that can help us to use the resources available more efficiently in order to process large amounts of data. In the context of computing statistics based on big data, this can involve:

- Implementing the computation of a given statistical procedure in a more efficient way (make better use of a given programming language or choose another programming language).
- Choosing/implementing a more efficient statistical procedure/algorithm (see, e.g., the *Uluru* algorithm).
- At a lower level, improving how the system allocates resources.

2.2 Units of information/data storage

The smallest unit of information in computing/digital data is called a *bit* (from *bi*nary dig*it*; abbrev. 'b') and can take one of two (symbolic) values, either a `0` or a `1` ("off" or "on"). Consider, for example, the decimal number `139`. Written in the binary system, `139` corresponds to the

binary number `10001011`. In order to store this number on a hard disk, we require a capacity of 8 bits, or one *byte* (1 byte = 8 bits; abbrev. 'B'). Historically, one byte encoded a single character of text (i.e., in the ASCII character encoding system). 4 bytes (or 32 bits) are called a *word*. When thinking of a given data set in its raw/binary representation, we can simply think of it as a row of `0`s and `1`s, as illustrated in the following figure.

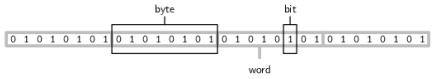


FIGURE 2.2: Writing data stored in RAM to a Mass Storage device (hard drive). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

Bigger units for storage capacity usually build on bytes:

- 1 kilobyte (KB) $= 1000^1 \approx 2^{10}$ bytes
- 1 megabyte (MB) $= 1000^2 \approx 2^{20}$ bytes
- 1 gigabyte (GB) $= 1000^3 \approx 2^{30}$ bytes
- 1 terabyte (TB) $= 1000^4 \approx 2^{40}$ bytes
- 1 petabyte (PB) $= 1000^5 \approx 2^{50}$ bytes
- 1 exabyte (EB) $= 1000^6 \approx 2^{60}$ bytes
- 1 zettabyte (ZB) $= 1000^7 \approx 2^{70}$ bytes

$1ZB = 1000000000000000000000$ bytes $=$ 1 billion terabytes $=$ 1 trillion gigabytes.

2.2.1 Example in R: Data types and information storage

Given the fact that computers only understand `0`s and `1`s, different approaches are taken to map these digital values to other symbols or images (text, decimal numbers, pictures, etc.) that we humans can more easily make sense of. Regarding text and numbers, these mappings involve *character encodings* (in which combinations of `0`s and `1`s represent a character in a specific alphabet) and *data types*.

Let's illustrate the main concepts with the simple numerical example from above. When we see the decimal number `139` written somewhere, we know that it means 'one-hundred-and-thirty-nine'. The fact that

our computer is able to print `139` on the screen means that our computer can somehow map a sequence of `0`s and `1`s to the symbols `1`, `3`, and `9`. Depending on what we want to do with the data value `139` on our computer, there are different ways of how the computer can represent this value internally. Inter alia, we could load it into RAM as a *string* ('text'/'character') or as an *integer* ('natural number') or *double* (numeric, floating point number). All of them can be printed on screen but only the latter two can be used for arithmetic computations. This concept can easily be illustrated in R.

We initiate a new variable with the value `139`. By using this syntax, R by default initiates the variable as an object of type `double`. We then can use this variable in arithmetic operations.

```
my_number <- 139
# check the class
typeof(my_number)
```

```
## [1] "double"
```

```
# arithmetic
my_number*2
```

```
## [1] 278
```

When we change the *data type* to 'character' (string) such operations are not possible.

```
# change and check type/class
my_number_string <- as.character(my_number)
typeof(my_number_string)
```

```
## [1] "character"
```

```
# try to multiply
my_number_string*2
```

```
## Error in my_number_string * 2: non-
numeric argument to binary operator
```

If we change the variable to type `integer`, we can still use math operators.

```
# change and check type/class
my_number_int <- as.integer(my_number)
typeof(my_number_int)
```

```
## [1] "integer"
```

```
# arithmetics
my_number_int*2
```

```
## [1] 278
```

Having all variables in the right type is relevant for data analytics with all kind of sample sizes. However, given the fact that different data types have to be represented differently internally, different types might take up more or less memory and therefore substantially affect the performance when dealing with massive amounts of data.

We can illustrate this point with `object.size()`:

```
object.size("139")
```

```
## 112 bytes
```

```
object.size(139)
```

```
## 56 bytes
```

2.3 Big Data econometrics

When thinking about how to approach a data analytics task based on large amounts of data, it is helpful to consider two key aspects concerning the computational burden involved. (i) how is the statistic we have in mind computed. That is, what is the formal definition of the statistic (and how computationally demanding is it)? And (ii), given a definition of the statistic, how does the program/software/language implement the computation thereof?

Regarding the former, we might realize that there is an alternative statistical procedure that would provide essentially the same output but that happens to be more efficient (here: computationally efficient in contrast to statistically efficient). The latter point is a question of how to efficiently implement the given statistical procedure in your computing environment (taking into consideration the computer's available resources: CPU, RAM, etc.).

Below, we look at both of these two aspects by means of illustrative examples.

2.4 Example: Fast least squares regression

As an illustration of how an alternative statistical procedure can speed up our analysis, we look at one such procedure that has recently been developed to estimate linear models when the classical OLS estimator is too computationally intense for very large samples: The *Uluru* algorithm (Dhillon et al., 2013).

2.4.1 OLS as a point of reference

Recall the OLS estimator in matrix notation, given the linear model $\mathbf{y} = \mathbf{X}\beta + \epsilon$:

$$\hat{\beta}_{OLS} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

In order to compute $\hat{\beta}_{OLS}$, we have to compute $(\mathbf{X}^\top\mathbf{X})^{-1}$, which implies a computationally expensive matrix inversion.[1] If our data set is large, $\mathbf{X}$ is large and the inversion can take up a lot of computation time. Moreover, the inversion and matrix multiplication to get $\hat{\beta}_{OLS}$ needs a lot of memory. In practice, it might well be that the estimation of a linear model via OLS with the standard approach in R (`lm()`) brings a computer to its knees, as there is not enough RAM available.

To further illustrate the point, we implement the OLS estimator in R.

```
beta_ols <-
     function(X, y) {

          # compute cross products and inverse
          XXi <- solve(crossprod(X,X))
          Xy <- crossprod(X, y)

          return( XXi  %*% Xy )
     }
```

Now, we will test our OLS estimator function with a few (pseudo) random numbers in a Monte Carlo study. First, we set the sample size parameters `n` (how many observations shall our pseudo sample have?) and `p` (how many variables shall describe these observations?) and initiate the data set `X`.

```
# set parameter values
n <- 10000000
p <- 4

# Generate sample based on Monte Carlo
# generate a design matrix (~ our 'dataset') with four variables and 10000 observations
X <- matrix(rnorm(n*p, mean = 10), ncol = p)
```

[1] The computational complexity of this is larger than $O(n^2)$. That is, for an input of size n, the time needed to compute (or the number of operations needed) is n^2.

```
# add column for intercept
X <- cbind(rep(1, n), X)
```

Now we define how the real linear model looks like that we have in mind and compute the output `y` of this model, given the input `X`.[2]

```
# MC model
y <- 2 + 1.5*X[,2] + 4*X[,3] - 3.5*X[,4] + 0.5*X[,5] + rnorm(n)
```

Finally, we test our `beta_ols` function.

```
# apply the ols estimator
beta_ols(X, y)
```

```
##         [,1]
## [1,]  1.9897
## [2,]  1.5006
## [3,]  4.0005
## [4,] -3.4999
## [5,]  0.4998
```

2.4.2 The Uluru algorithm as an alternative to OLS

Following Dhillon et al. (2013), we implement a procedure to compute $\hat{\beta}_{Uluru}$:

$$\hat{\beta}_{Uluru} = \hat{\beta}_{FS} + \hat{\beta}_{correct}$$

, where

$$\hat{\beta}_{FS} = (\mathbf{X}_{subs}^\top \mathbf{X}_{subs})^{-1} \mathbf{X}_{subs}^\top \mathbf{y}_{subs}$$

, and

$$\hat{\beta}_{correct} = \frac{n_{subs}}{n_{rem}} \cdot (\mathbf{X}_{subs}^\top \mathbf{X}_{subs})^{-1} \mathbf{X}_{rem}^\top \mathbf{R}_{rem}$$

[2] In reality we would not know this, of course. Acting as if we knew the real model is exactly the point of Monte Carlo studies. It allows us to analyze the properties of estimators by simulation.

, and

$$\mathbf{R}_{rem} = \mathbf{Y}_{rem} - \mathbf{X}_{rem} \cdot \hat{\beta}_{FS}$$

.

The key idea behind this is that the computational bottleneck of the OLS estimator, the cross product and matrix inversion, $(\mathbf{X}^\top \mathbf{X})^{-1}$, is only computed on a sub-sample (X_{subs}, etc.), not the entire data set. However, the remainder of the data set is also taken into consideration (in order to correct a bias arising from the sub-sampling). Again, we implement the estimator in R to further illustrate this point.

```
beta_uluru <-
     function(X_subs, y_subs, X_rem, y_rem) {

          # compute beta_fs (this is simply OLS applied to the subsample)
          XXi_subs <- solve(crossprod(X_subs, X_subs))
          Xy_subs <- crossprod(X_subs, y_subs)
          b_fs <- XXi_subs  %*% Xy_subs

          # compute \mathbf{R}_{rem}
          R_rem <- y_rem - X_rem %*% b_fs

          # compute \hat{\beta}_{correct}
          b_correct <- (nrow(X_subs)/(nrow(X_rem))) * XXi_subs %*% crossprod(X_rem, R_rem)

          # beta uluru
          return(b_fs + b_correct)
     }
```

Test it with the same input as above:

```
# set size of subsample
n_subs <- 1000
# select subsample and remainder
n_obs <- nrow(X)
X_subs <- X[1L:n_subs,]
y_subs <- y[1L:n_subs]
```

```
X_rem <- X[(n_subs+1L):n_obs,]
y_rem <- y[(n_subs+1L):n_obs]

# apply the uluru estimator
beta_uluru(X_subs, y_subs, X_rem, y_rem)
```

```
##         [,1]
## [1,]  1.9244
## [2,]  1.5015
## [3,]  4.0059
## [4,] -3.5000
## [5,]  0.5004
```

This looks quite good already. Let's have a closer look with a little Monte Carlo study. The aim of the simulation study is to visualize the difference between the classical OLS approach and the *Uluru* algorithm with regard to bias and time complexity if we increase the subsample size in *Uluru*. For simplicity, we only look at the first estimated coefficient β_1.

```
# define subsamples
n_subs_sizes <- seq(from = 1000, to = 500000, by=10000)
n_runs <- length(n_subs_sizes)
# compute uluru result, stop time
mc_results <- rep(NA, n_runs)
mc_times <- rep(NA, n_runs)
for (i in 1:n_runs) {
     # set size of subsample
     n_subs <- n_subs_sizes[i]
     # select subsample and remainder
     n_obs <- nrow(X)
     X_subs <- X[1L:n_subs,]
     y_subs <- y[1L:n_subs]
     X_rem <- X[(n_subs+1L):n_obs,]
     y_rem <- y[(n_subs+1L):n_obs]
```

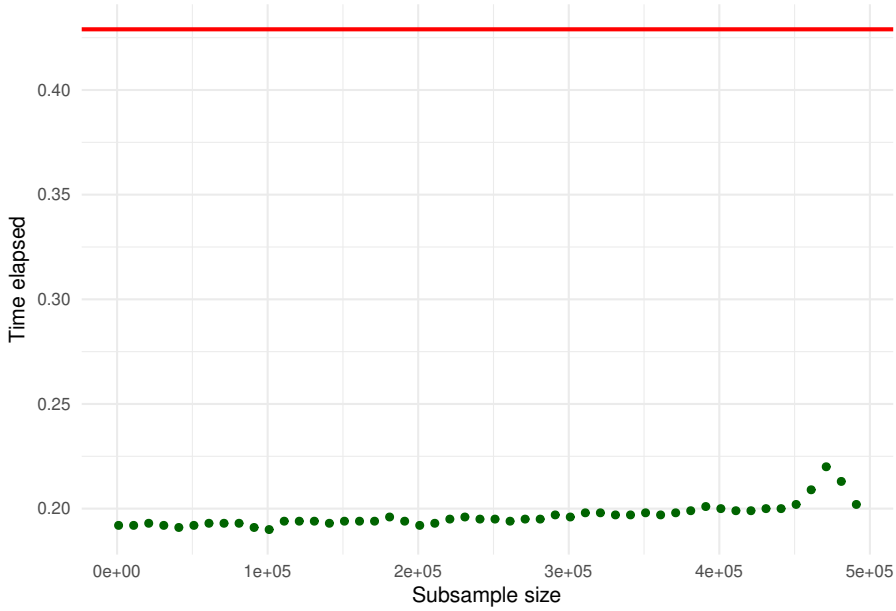
```
    mc_results[i] <- beta_uluru(X_subs, y_subs, X_rem, y_rem)[2] # the first element is the i
    mc_times[i] <- system.time(beta_uluru(X_subs, y_subs, X_rem, y_rem))[3]

}

# compute ols results and ols time
ols_time <- system.time(beta_ols(X, y))
ols_res <- beta_ols(X, y)[2]
```

Let's visualize the comparison with OLS.

```
# load packages
library(ggplot2)

# prepare data to plot
plotdata <- data.frame(beta1 = mc_results,
                       time_elapsed = mc_times,
                       subs_size = n_subs_sizes)
```

First, let's look at the time used estimate the linear model.

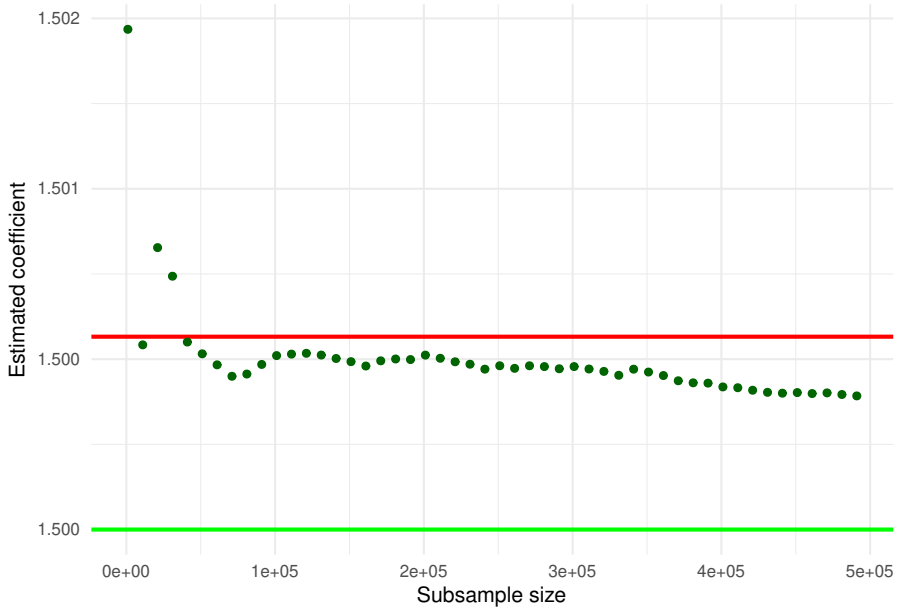
```
ggplot(plotdata, aes(x = subs_size, y = time_elapsed)) +
    geom_point(color="darkgreen") +
    geom_hline(yintercept = ols_time[3],
               color = "red",
               size = 1) +
    theme_minimal() +
    ylab("Time elapsed") +
    xlab("Subsample size")
```



The horizontal red line indicates the computation time for estimation via OLS, the green points indicate the computation time for the estimation via the Ulruru algorithm. Note that even for large sub-samples, the computation time is substantially lower than for OLS.

Finally, let's have a look at how close the results are to OLS.

```
ggplot(plotdata, aes(x = subs_size, y = beta1)) +
     geom_hline(yintercept = ols_res,
                color = "red",
                size = 1) +
       geom_hline(yintercept = 1.5,
                color = "green",
                size = 1) +
     geom_point(color="darkgreen") +

     theme_minimal() +
     ylab("Estimated coefficient") +
     xlab("Subsample size")
```



The horizontal red line indicates the size of the estimated coefficient, when using OLS. The horizontal green line indicates the size of the actual coefficient. The green points indicate the size of the same coefficient estimated by the Uluru algorithm for different sub-sample sizes. Note that even relatively small sub-samples already deliver estimates very close to the OLS estimates.

3

Resource allocation

When optimizing the performance of an analytics program processing large amounts of data, it is useful to differentiate between the efficient allocation of computational (CPU) power, and the allocation of RAM (and mass storage).[1] Below, we will look at both aspects in turn.

3.1 Case study: Parallel processing

In this example, we estimate a simple regression model that aims to assess racial discrimination in the context of police stops.[2] The example is based on the 'Minneapolis Police Department 2017 Stop Dataset', containing data on nearly all stops made by the Minneapolis Police Department for the year 2017.

We start with importing the data into R.

```
url <- "https://vincentarelbundock.github.io/Rdatasets/csv/carData/MplsStops.csv"
stopdata <- data.table::fread(url)
```

We specify a simple linear probability model that aims to test whether a person identified as 'white' is less likely to have her vehicle searched when stopped by the police. In order to take into account

[1] In many data analysis tasks the two are, of course, intertwined. However, keeping both aspects in mind when optimizing an analytics program helps to choose the right tools.

[2] Note that this example aims to illustrate a point about computation in an applied econometrics context. It does not make any argument about identification or the broader research question whatsoever.

level-differences between different police precincts, we add precinct-indicators to the regression specification

First, let's remove observations with missing entries (NA) and code our main explanatory variable and the dependent variable.

```
# remove incomplete obs
stopdata <- na.omit(stopdata)
# code dependent var
stopdata$vsearch <- 0
stopdata$vsearch[stopdata$vehicleSearch=="YES"] <- 1
# code explanatory var
stopdata$white <- 0
stopdata$white[stopdata$race=="White"] <- 1
```

We specify our baseline model as follows.

```
model <- vsearch ~ white + factor(policePrecinct)
```

And estimate the linear probability model via OLS (the `lm` function).

```
fit <- lm(model, stopdata)
summary(fit)
```

```
##
## Call:
## lm(formula = model, data = stopdata)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.1394 -0.0633 -0.0547 -0.0423  0.9773
##
## Coefficients:
##                           Estimate Std. Error t value
## (Intercept)                0.05473    0.00515   10.62
## white                     -0.01955    0.00446   -4.38
## factor(policePrecinct)2    0.00856    0.00676    1.27
```

```
## factor(policePrecinct)3  0.00341    0.00648    0.53
## factor(policePrecinct)4  0.08464    0.00623   13.58
## factor(policePrecinct)5 -0.01246    0.00637   -1.96
##                          Pr(>|t|)
## (Intercept)               < 2e-16 ***
## white                     1.2e-05 ***
## factor(policePrecinct)2     0.21
## factor(policePrecinct)3     0.60
## factor(policePrecinct)4   < 2e-16 ***
## factor(policePrecinct)5     0.05 .
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.254 on 19078 degrees of freedom
## Multiple R-squared:  0.025,  Adjusted R-squared:  0.0248
## F-statistic: 97.9 on 5 and 19078 DF,  p-value: <2e-16
```

A potential problem with this approach (and there might be many more in this simple example) is that observations stemming from different police precincts might be correlated over time. If that is the case, we likely underestimate the coefficient's standard errors. There is a standard approach to compute estimates for so-called *cluster-robust* standard errors which would take the problem of correlation over time within clusters into consideration (and deliver a more conservative estimate of the SEs). However, this approach only works well if the number of clusters in the data is roughly 50 or more. Here we only have 5.

The alternative approach is to compute bootstrapped clustered standard errors. That is, we apply the bootstrap resampling procedure[3] at the cluster level. Specifically, we draw B samples (with replacement), estimate and record for each bootstrap-sample the coefficient vector, and then estimate SE_{boot} based on the standard deviation of all respective estimated coefficient values.

[3] https://en.wikipedia.org/wiki/Bootstrapping_(statistics)

```
# load packages
library(data.table)
# set the 'seed' for random numbers (makes the example reproducible)
set.seed(2)

# set number of bootstrap iterations
B <- 10
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)
# draw bootstrap samples, estimate model for each sample
for (i in 1:B) {

     # draw sample of precincts (cluster level)
     precincts_i <- sample(precincts, size = 5, replace = TRUE)
     # get observations
     bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
     bs_i <- rbindlist(bs_i)

     # estimate model and record coefficients
     boot_coefs[i,] <- coef(lm(model, bs_i))[1:2] # ignore FE-coefficients
}
```

Finally, let's compute SE_{boot}.

```
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

```
## [1] 0.004043 0.004690
```

Note that even with a very small B, computing SE_{boot} takes up some time to compute. When setting B to over 500, computation time will be substantial. Also note that running this code does hardly use up more memory than the very simple approach without bootstrapping

(after all, in each bootstrap iteration the data set used to estimate the model is approximately the same size as the original data set). There is little we can do to improve the script's performance regarding memory. However we can tell R how to allocate CPU resources more efficiently to handle that many regression estimates.

Particularly, we can make use of the fact that most modern computing environments (such as a laptop) have CPUs with several *cores*. We can exploit this fact by instructing the computer to run the computations *in parallel* (simultaneously computing on several cores). The following code is a parallel implementation of our bootstrap procedure which does exactly that.

```
# install.packages("doSNOW", "parallel")
# load packages for parallel processing
library(doSNOW)

# get the number of cores available
ncores <- parallel::detectCores()
# set cores for parallel processing
ctemp <- makeCluster(ncores) #
registerDoSNOW(ctemp)


# set number of bootstrap iterations
B <- 10
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)

# bootstrapping in parallel
boot_coefs <-
     foreach(i = 1:B, .combine = rbind, .packages="data.table") %dopar% {

          # draw sample of precincts (cluster level)
          precincts_i <- sample(precincts, size = 5, replace = TRUE)
```

```
          # get observations
          bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
          bs_i <- rbindlist(bs_i)

          # estimate model and record coefficients
          coef(lm(model, bs_i))[1:2] # ignore FE-coefficients

     }


# be a good citizen and stop the snow clusters
stopCluster(cl = ctemp)
```

As a last step, we compute again SE_{boot}.

```
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

```
## (Intercept)       white
##    0.004193    0.004334
```

3.2 Case study: Memory allocation

Consider the first steps of a data pipeline in R. The first part of our script to import and clean the data looks as follows.

```
#############################################################
# Big Data Statistics: Flights data import and preparation
#
# U. Matter, January 2019
#############################################################
```

```
# SET UP -----------------

# fix variables
DATA_PATH <- "materials/data/flights.csv"

# DATA IMPORT ----------------
flights <- read.csv(DATA_PATH)

# DATA PREPARATION --------
flights <- flights[,-1:-3]
```

When running this script, we notice that some of the steps need a noticable amount of time to process. Moreover, while none of these steps obviously involves a lot of computation (such as a matrix inversion or numerical optimization), it quite likely involves memory allocation. We first read data into RAM (allocated to R by our operating system). It turns out that there are different ways to allocate RAM when reading data from a CSV file. Depending on the amount of data to be read in, one or the other approach might be faster. We first investigate the RAM allocation in R with `mem_change()` and `mem_used()`.

```
# SET UP -----------------

# fix variables
DATA_PATH <- "data/flights.csv"
# load packages
library(pryr)
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp

##
## Attaching package: 'pryr'

## The following object is masked from 'package:data.table':
```

```
##
##     address
```

```
# check how much memory is used by R (overall)
mem_used()
```

```
## 1.04 GB
```

```
# check the change in memory due to each step

# DATA IMPORT ----------------
mem_change(flights <- read.csv(DATA_PATH))
```

```
## 33.2 MB
```

```
# DATA PREPARATION --------
flights <- flights[,-1:-3]

# check how much memory is used by R now
mem_used()
```

```
## 1.07 GB
```

The last result is kind of interesting. The object `flights` must have been larger right after importing it than at the end of the script. We have thrown out several variables, after all. Why does R still use that much memory? R does by default not 'clean up' memory unless it is really necessary (meaning no more memory is available). In this case, R has still way more memory available from the operating system, thus there is no need to 'collect the garbage' yet. However, we can force R to collect the garbage on the spot with `gc()`. This can be helpful to better keep track of the memory needed by an analytics script.

```
gc()
```

```
##             used  (Mb) gc trigger   (Mb)  max used
## Ncells   1080420  57.8    2100091  112.2   1889052
```

```
## Vcells 126631854 966.2  213476124 1628.7 211148494
##          (Mb)
## Ncells  100.9
## Vcells 1611.0
```

Now, let's see how we can improve the performance of this script with regard to memory allocation. Most memory is allocated when importing the file. Obviously, any improvement of the script must still result in importing all the data. However, there are different ways to read data into RAM. `read.csv()` reads all lines of a csv file consecutively. In contrast, `data.table::fread()` first 'maps' the data file into memory and only then actually reads it in line by line. This involves an additional initial step, but the larger the file, the less relevant is this first step with regard to the total time needed to read all the data into memory. By switching on the `verbose` option, we can actually see what `fread` is doing.

```
# load packages
library(data.table)

# DATA IMPORT ----------------
flights <- fread(DATA_PATH, verbose = TRUE)
```

```
##   OpenMP version (_OPENMP)       201511
##   omp_get_num_procs()            12
##   R_DATATABLE_NUM_PROCS_PERCENT  unset (default 50)
##   R_DATATABLE_NUM_THREADS        unset
##   R_DATATABLE_THROTTLE           unset (default 1024)
##   omp_get_thread_limit()         2147483647
##   omp_get_max_threads()          12
##   OMP_THREAD_LIMIT               unset
##   OMP_NUM_THREADS                unset
##   RestoreAfterFork               true
##  data.table is using 6 threads with throttle==1024. See ?setDTthreads.
## freadR.c has been passed a filename: data/flights.csv
## [01] Check arguments
##   Using 6 threads (omp_get_max_threads()=12, nth=6)
```

```
##   NAstrings = [<<NA>>]
##   None of the NAstrings look like numbers.
##   show progress = 0
##   0/1 column will be read as integer
## [02] Opening the file
##   Opening file data/flights.csv
##   File opened, size = 29.53MB (30960660 bytes).
##   Memory mapped ok
## [03] Detect and skip BOM
## [04] Arrange mmap to be \0 terminated
##  \n has been found in the input and different lines can end with different line endings (e.g. mi
## [05] Skipping initial rows if needed
##  Positioned on line 1 starting: <<year,month,day,dep_time,sched_>>
## [06] Detect separator, quoting rule, and ncolumns
##   Detecting sep automatically ...
##   sep=','  with 100 lines of 19 fields using quote rule 0
##  Detected 19 columns on line 1. This line is either column names or first data row. Line starts a
##   Quote rule picked = 0
##   fill=false and the most number of columns found is 19
## [07] Detect column types, good nrow estimate and whether first row is column names
##  Number of sampling jump points = 100 because (30960659 bytes from row 1 to eof) / (2 * 8882 jump
##   Type codes (jump 000)    : 555555555C5CCC5555B  Quote rule 0
##   Type codes (jump 100)    : 555555555C5CCC5555B  Quote rule 0
##  'header' determined to be true due to column 1 containing a string on row 1 and a lower type (in
##   =====
##  Sampled 10048 rows (handled \n inside quoted fields) at 101 jump points
##  Bytes from first data row on line 2 to the end of last row: 30960501
##   Line length: mean=92.03 sd=3.56 min=68 max=98
##   Estimated number of rows: 30960501 / 92.03 = 336403
##   Initial alloc = 370043 rows (336403 + 9%) using bytes/max(mean-
2*sd,min) clamped between [1.1*estn, 2.0*estn]
##   =====
## [08] Assign column names
## [09] Apply user overrides on column types
##   After 0 type and 0 drop user overrides : 555555555C5CCC5555B
## [10] Allocate memory for the datatable
##   Allocating 19 column slots (19 - 0 dropped) with 370043 rows
```

```
## [11] Read the data
##   jumps=[0..30), chunk_size=1032016, total_size=30960501
## Read 336776 rows x 19 columns from 29.53MB (30960660 bytes) file in 00:00.062 wall clock time
## [12] Finalizing the datatable
##   Type counts:
##         14 : int32     '5'
##          1 : float64   'B'
##          4 : string    'C'
## =============================
##    0.000s (  0%) Memory map 0.029GB file
##    0.003s (  4%) sep=',' ncol=19 and header detection
##    0.000s (  0%) Column type detection using 10048 sample rows
##   0.001s (  1%) Allocation of 370043 rows x 19 cols (0.033GB) of which 336776 ( 91%) rows used
##   0.059s ( 94%) Reading 30 chunks (0 swept) of 0.984MB (each chunk 11225 rows) using 6 threads
##             +             0.015s    (   25%)   Parse   to   row-
major thread buffers (grown 0 times)
##    +    0.027s ( 43%) Transpose
##    +    0.017s ( 27%) Waiting
##        0.000s  (    0%)  Rereading  0  columns  due  to  out-of-
sample type exceptions
##    0.062s        Total
```

Let's put it all together and look at the memory changes and usage. For a fair comparison, we first have to delete `flights` and collect the garbage with `gc()`.

```
# SET UP -----------------

# fix variables
DATA_PATH <- "data/flights.csv"
# load packages
library(pryr)
library(data.table)

# housekeeping
flights <- NULL
gc()
```

```
##             used  (Mb) gc trigger   (Mb)  max used
## Ncells   1069455  57.2    2100091  112.2   1889052
## Vcells 123466113 942.0  213476124 1628.7 211148494
##          (Mb)
## Ncells  100.9
## Vcells 1611.0
```

```
# check the change in memory due to each step

# DATA IMPORT ----------------
mem_change(flights <- fread(DATA_PATH))
```

```
## 35.8 MB
```

3.3 Beyond memory

In the previous example we have inspected how RAM is allocated to store objects in the R computing environment. But what if all RAM of our computer is not enough to store all the data we want to analyze?

Modern operating systems have a way to dealing with such a situation. Once all RAM is used up by the currently running programs, the OS allocates parts of the memory back to the hard-disk which then works as *virtual memory*. The following figure illustrates this point.

For example, when we implement an R-script that imports one file after the other into the R environment, ignoring the RAM capacity of our computer, the OS will start *paging* data to the virtual memory. This happens 'under the hood' without explicit instructions by the user. We quite likely notice that the computer slows down a lot when this happens.

While this default usage of virtual memory by the OS is helpful to run several applications at the same time, each taking up a moderate amount of memory, it is not a really useful tool for processing large amounts of data in one application (R). However, the underlying idea

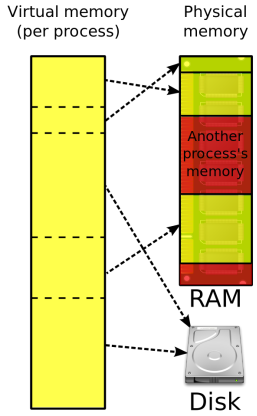


FIGURE 3.1: Virtual memory. Figure by Ehamberg (CC BY-SA 3.0).

of using both RAM and Mass storage simultaneously in order to cope with a lack of memory is very useful in the context of big data analytics.

Several R packages have been developed that exploit the idea behind virtual memory explicitly for analyzing large amounts of data. The basic idea behind these packages is to map a data set to the hard disk when loading it into R. The actual data values are stored in chunks on the hard-disk, while the structure/metadata of the data set is loaded into R. See this week's slide set as well as Walkowiak (2016), Chapter 3 for more details and example code.

4

Advanced R Programming

4.1 R-tools to investigate performance/resource allocation

package	function	purpose
`utils`	`object.size()`	Provides an estimate of the memory that is being used to store an R object.
`pryr`	`object_size()`	Works similarly to `object.size()`, but counts more accurately and includes the size of environments.
`pryr`	`compare_size()`	Makes it easy to compare the output of object_size and object.size.
`pryr`	`mem_used()`	Returns the total amount of memory (in megabytes) currently used by R.
`pryr`	`mem_change()`	Shows the change in memory (in megabytes) before and after running code.
`base`	`system.time()`	Returns CPU (and other) times that an R expression used.
`microbenchmark`	`microbenchmark()`	Highly accurate timing of R expression evaluation.
`bench`	`mark()`	Benchmark a series of functions.
`profvis`	`profvis()`	Profiles an R expression and visualizes the profiling data (usage of memory, time elapsed, etc.).

4.2 R-tools to investigate structures and types

package	function	purpose
`utils`	`str()`	Compactly display the structure of an arbitrary R object.
`base`	`class()`	Prints the class(es) of an R object.
`base`	`typeof()`	Determines the (R-internal) type or storage mode of an object.

4.3 Data types and memory/storage

Data loaded into RAM can be interpreted differently by R depending on the data *type*. Some operators or functions in R only accept data of a specific type as arguments. For example, we can store the numeric values `1.5` and `3` in the variables `a` and `b`, respectively.

```
a <- 1.5
b <- 3
a + b
```

```
## [1] 4.5
```

R interprets this data as type `double` (class 'numeric'):

```
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

```
object.size(a)
```

```
## 56 bytes
```

If, however, we define `a` and `b` as follows, R will interpret the values stored in `a` and `b` as text (`character`).

```
a <- "1.5"
b <- "3"
a + b
```

```
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

```
object.size(a)
```

```
## 56 bytes
```

Note that the symbols `1.5` take up more or less memory depending on the data-type they are stored in. This directly links to how data/information is stored/represented in binary code, which in turn is reflected in how much memory is used to store these symbols in an object as well as what we can do with it.

4.4 Data structures

For now, we have only looked at individual bytes of data. An entire data set can consist of gigabytes of data and contain both text and numeric values. R provides several classes of objects providing different data structures. Both the choice of data types and data structures to store data in can affect how much memory is needed to contain a dataset in RAM.

4.4.1 Vectors vs Factors in R

Vectors are collections of values of the same type. They can contain either all numeric values or all character values.

FIGURE 4.1: Illustration of a numeric vector (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we can initiate a character vector containing information on the home towns of persons participating in a survey.

```
hometown <- c("St.Gallen", "Basel", "St.Gallen")
hometown
```

```
## [1] "St.Gallen" "Basel"     "St.Gallen"
```

```
object.size(hometown)
```

```
## 200 bytes
```

Unlike in the data types example above, it would likely be not that practical to store these values as type `numeric` to save memory. R would not know how to translate these strings into floating point numbers. Alternatively, we could think of a correspondence table that assigns a numeric (id) code to each unique town name in the data set. This way we would save memory but it would mean additional effort to work with the data. Fortunately, basic R already implements exactly this idea in a user-friendly way in a data-structure called `factor`.

Factors are sets of categories. Thus, the values come from a fixed set of possible values.

Considering the same example as above, we can store the same information in an object of type class `factor`.

FIGURE 4.2: Illustration of a factor (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

```
hometown_f <- factor(c("St.Gallen", "Basel", "St.Gallen"))
hometown_f
```

```
## [1] St.Gallen Basel     St.Gallen
## Levels: Basel St.Gallen
```

```
object.size(hometown_f)
```

```
## 584 bytes
```

At first sight, the fact that `hometown_f` takes up more memory than its character vector sibling seems odd. But, we have encountered this kind of 'paradox' before. Again, the more sophisticated approach involves an 'overhead' (here not in terms of computing time but in terms of structure encoded in an object). `hometown_f` has more 'structure' (i.e., a mapping of numbers to 'factor levels'/category labels). This additional structure is also information that needs to be stored somewhere. As in previous examples of this 'overhead costs', this disadvantage is diminishing with larger data sets:

```
# create a large character vector
hometown_large <- rep(hometown, times = 1000)
# and the same content as factor
hometown_large_f <- factor(hometown_large)
# compare size
object.size(hometown_large)
```

```
## 24168 bytes
```

```
object.size(hometown_large_f)
```

```
## 12568 bytes
```

4.4.2 Matrices/Arrays

Matrices are two-dimensional collections of values, arrays higher-dimensional collections of values, of the same type.

FIGURE 4.3: Illustration of a numeric matrix (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we can initiate a three-row/two-column numeric matrix as follows.

```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

And a three-dimensional numeric array as follows.

```
my_array <- array(c(1,2,3,4,5,6), dim = 3)
my_array
```

```
## [1] 1 2 3
```

4.4.3 Data frames, tibbles, and data tables

Recall that data frames are the typical representation of a (table-like) data set in R. Each column can contain a vector of a given data type

(or a factor), but all columns need to be of identical length. Thus in the context of data analysis, we would say that each row of a data frame contains an observation, and each column contains a characteristic of this observation.

FIGURE 4.4: Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

The historical implementation of data frames in R is not very comfortable to work with large data sets.[1] Several newer implementations of the data-frame concept in R aim to make data processing faster. One is called `tibbles`, implemented and used in the `tidyverse` packages. The other is called `data table`, implemented in the `data.table`-package. In this course we will focus on the `data.table`-package.

Here is how we define a `data.table` in R:

```
# load package
library(data.table)
# initiate a data.table
dt <- data.table(person = c("Alice", "Ben"),
                 age = c(50, 30),
                 gender = c("f", "m"))
dt
```

```
##    person age gender
## 1:  Alice  50      f
## 2:    Ben  30      m
```

4.4.4 Lists

Similar to data frames and data tables, lists can contain different types of data in each element. For example, a list could contain different

[1] In the early days of R this was not really an issue because data sets that are rather large by today's standards (in the Gigabytes) could not have been handled properly by normal computers anyhow (due to a lack of RAM).

other lists, data frames, and vectors with differing numbers of elements.

FIGURE 4.5: Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

This flexibility can easily be demonstrated by combining some of the data structures created in the examples above:

```
my_list <- list(my_array, my_matrix, dt)
my_list
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## [[3]]
##    person age gender
## 1:  Alice  50      f
## 2:    Ben  30      m
```

4.5 Programming with (Big) Data in R

4.5.1 Typical Programming Tasks

Programming tasks in the context of data analysis typically fall into one of the following broad categories.

- Procedures to import/export data.

- Procedures to clean and filter data.
- Implement functions for statistical analysis.

When writing a program to process large amounts of data in any of these areas, it is helpful to take into consideration the following design choices:

1. Which basic (already implemented) R functions are more or less suitable as building blocks for the program?
2. How can we exploit/avoid some of R's lower-level characteristics in order to implement efficient functions?
3. Is there a need to interface with a lower-level programming language in order to speed up the code? (advanced topic)

Finally, there is an additional important point to be made regarding the implementation of functions for *statistical analysis*: Independent of *how* we write a statistical procedure in R (or in any other language, for that matter), is there an *alternative statistical procedure/algorithm* that is faster but delivers approximately the same result (as long as we use a sufficiently large data sets). The following subsections elaborate briefly on each of these points and show some code examples to further illustrate these points.

4.5.2 Building blocks for programming with big data

When writing a program in R, we can rely on many already implemented functions on which we can build. Often, there are even several functions already implemented that take care of essentially the same task. When the amount of data to be processed by these functions is not large, it doesn't matter that much which ones we choose to build our program on. However, when we are writing a program which likely has to process large amounts of data, we should think more closely about which building blocks we choose to base our program on. For example, when writing the data-import part of a program, we could use the traditional `read.csv()` or `fread()` from the `data.table`-package. The result is very similar (in many situations, the differences of the resulting objects would not matter at all).

```
# read a CSV-file the 'traditional way'
flights <- read.csv("data/flights.csv")
class(flights)
```

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

```
# alternative (needs the data.table package)
library(data.table)
flights <- fread("data/flights.csv")
class(flights)
```

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

However, the latter approach is usually much faster (see above for why this is the case in this example).

```
system.time(flights <- read.csv("data/flights.csv"))
```

```
##    user  system elapsed
##   1.128   0.008   1.136
```

```
system.time(flights <- fread("data/flights.csv"))
```

```
##    user  system elapsed
##   0.278   0.002   0.050
```

4.5.3 Writing efficient code

4.5.3.1 Memory allocation before looping

Recall the code example from the introductory lecture. When we write a `for`-loop that results in a vector or list of values, it is favorable to instruct R to pre-allocate the memory necessary to contain the final result. If we don't do that, each iteration of the loop causes R to re-allocate memory because the number of elements in the vector/list is changing. In simple terms, this means that R needs to execute more steps in each iteration.

In the following example, we compare the performance of two functions. One taking this principle into account, the other not. The function takes a numeric vector as input and returns the square root of each element of the numeric vector.

```
# naïve implementation
sqrt_vector <-
     function(x) {
          output <- c()
          for (i in 1:length(x)) {
               output <- c(output, x[i]^(1/2))
          }

          return(output)
     }

# implementation with pre-allocation of memory
sqrt_vector_faster <-
     function(x) {
          output <- rep(NA, length(x))
          for (i in 1:length(x)) {
               output[i] <-  x[i]^(1/2)
          }

          return(output)
     }
```

As a proof of concept we use `system.time()` to measure the difference in speed for various input sizes.[2]

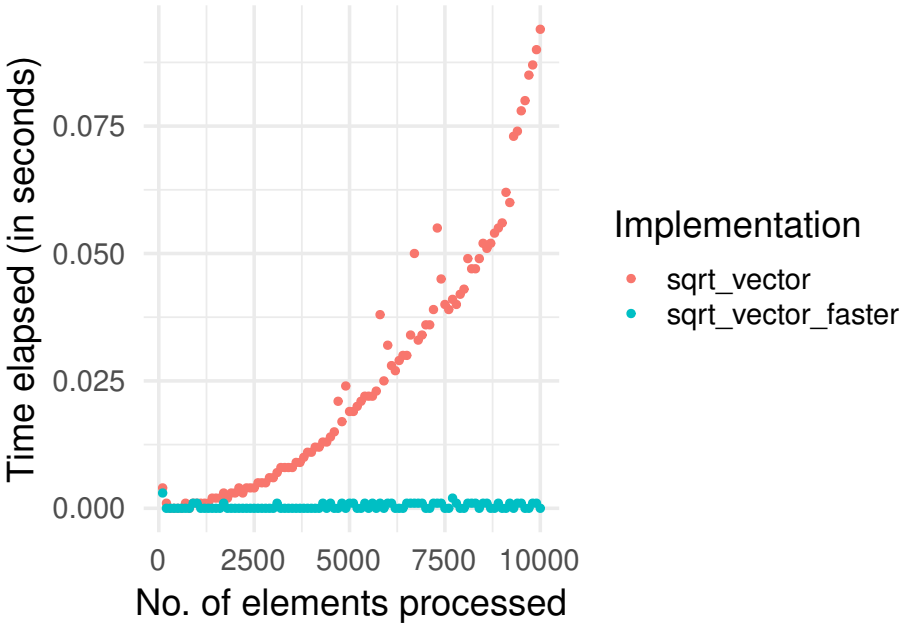
```
# the different sizes of the vectors we will put into the two functions
input_sizes <- seq(from = 100, to = 10000, by = 100)
# create the input vectors
inputs <- sapply(input_sizes, rnorm)
```

[2] We generate the numeric input by drawing vectors of (pseudo) random numbers via `rnorm()`.

```
# compute ouputs for each of the functions
output_slower <-
     sapply(inputs,
            function(x){ system.time(sqrt_vector(x))["elapsed"]
                 }
            )
output_faster <-
     sapply(inputs,
            function(x){ system.time(sqrt_vector_faster(x))["elapsed"]
                 }
            )
```

The following plot shows the difference in the performance of the two functions.

```
# load packages
library(ggplot2)

# initiate data frame for plot
plotdata <- data.frame(time_elapsed = c(output_slower, output_faster),
                       input_size = c(input_sizes, input_sizes),
                       Implementation= c(rep("sqrt_vector", length(output_slower)),
                            rep("sqrt_vector_faster", length(output_faster))))

# plot
ggplot(plotdata, aes(x=input_size, y= time_elapsed)) +
     geom_point(aes(colour=Implementation)) +
     theme_minimal(base_size = 18) +
     ylab("Time elapsed (in seconds)") +
     xlab("No. of elements processed")
```



4.5.3.2 Vectorization

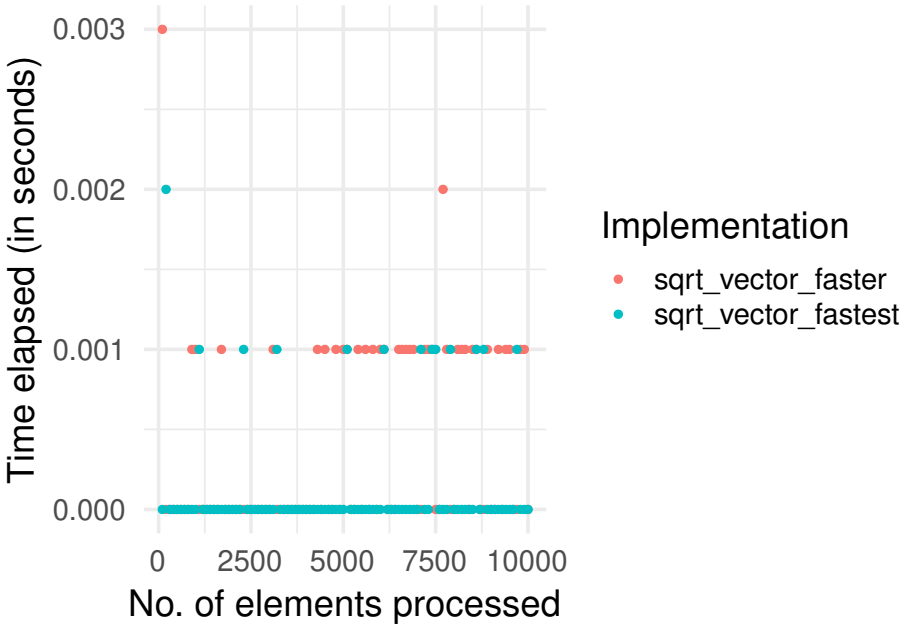
We can further improve the performance of this function by exploiting the fact that in R 'everything is a vector' and that many of the basic R functions (such as math operators) are *vectorized*. In simple terms, this means that an operation is implemented to directly work on vectors in such a way that it can take advantage of the similarity of each of the vector's elements. That is, R only has to figure out once how to apply a given function to a vector element in order to apply it to all elements of the vector. In a simple loop, R has to go through the same 'preparatory' steps again and again in each iteration.

```
# implementation with vectorization
sqrt_vector_fastest <-
     function(x) {
               output <-  x^(1/2)
          return(output)
     }

# speed test
```

```
output_fastest <-
     sapply(inputs,
            function(x){ system.time(sqrt_vector_fastest(x))["elapsed"]
                 }
            )
```

Let's have a look at whether this improves the function's performance further.

```
# load packages
library(ggplot2)

# initiate data frame for plot
plotdata <- data.frame(time_elapsed = c(output_faster, output_fastest),
                       input_size = c(input_sizes, input_sizes),
                       Implementation= c(rep("sqrt_vector_faster", length(output_faster)),
                            rep("sqrt_vector_fastest", length(output_fastest))))

# plot
ggplot(plotdata, aes(x=input_size, y= time_elapsed)) +
     geom_point(aes(colour=Implementation)) +
     theme_minimal(base_size = 18) +
     ylab("Time elapsed (in seconds)") +
     xlab("No. of elements processed")
```



In the example above, we have simply exploited the fact that many of R's basic functions (such as math operators) are vectorized. If the program we want to implement cannot directly benefit from such a function, there are basically two ways to make use of vectorization (instead of loops written in R).

One approach is to use an `apply`-type function instead of loops. Probably most widely used is `lapply()`, a function that takes a vector (atomic or list) as input and applies a function `FUN` to each of its elements. It is a straightforward alternative to `for`-loops in many situations. The following example shows how we can get the same result by either writing a loop or using `lapply()`. The aim of the code example is to import the Health News in Twitter Data Set[3] by Karami et al. (2017). The raw data consists of several text files that need to be imported to R consecutively.

The text-files are located in `data/twitter_texts/`. For either approach of importing all of these files, we first need a list of the paths to all of the files. We can get this with `list.files()`. Also, for either approach we will make use of the `fread`-function in the `data.table`-package.

[3] https://archive.ics.uci.edu/ml/datasets/Health+News+in+Twitter

```
# load packages
library(data.table)

# get a list of all file-paths
textfiles <- list.files("data/twitter_texts", full.names = TRUE)
```

Now we can read in all the text files with a `for`-loop as follows.

```
# prepare loop
all_texts <- list()
n_files <- length(textfiles)
length(all_texts) <- n_files
# read all files listed in textfiles
for (i in 1:n_files) {
     all_texts[[i]] <- fread(textfiles[i])
}
```

The imported files are now stored as `data.table`-objects in the list `all_texts`. With the following line of code we combine all of them in one `data.table`.

```
# combine all in one data.table
twitter_text <- rbindlist(all_texts)
# check result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':  42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 5
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:18
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care -
BMA poll http://bbc.in/1ChTBRv" "Short people's 'heart risk greater' http://bbc.in/1ChTANp" "New
##  - attr(*, ".internal.selfref")=<externalptr>
```

Alternatively, we can make use of `lapply` as follows in order to achieve exactly the same.

```
# prepare loop
all_texts <- lapply(textfiles, fread)
# combine all in one data.table
twitter_text <- rbindlist(all_texts)
# check result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':  42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 5
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:18
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care -
BMA poll http://bbc.in/1ChTBRv" "Short people's 'heart risk greater' http://bbc.in/1ChTANp" "New
##  - attr(*, ".internal.selfref")=<externalptr>
```

Finally, we can make use of `Vectorization()` in order to 'vectorize' (as far as possible) our own import function (written for this example).

```
# initiate the import function
import_file <-
     function(x) {
          parsed_x <- fread(x)
          return(parsed_x)
     }

# 'vectorize' it
import_files <- Vectorize(import_file, SIMPLIFY = FALSE)

# Apply the vectorized function
all_texts <- import_files(textfiles)
twitter_text <- rbindlist(all_texts)
# check the result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':  42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 5
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:18
```

```
## $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care -
BMA poll http://bbc.in/1ChTBRv" "Short people's 'heart risk greater' http://bbc.in/1ChTANp" "New
##  - attr(*, ".internal.selfref")=<externalptr>
```

4.5.4 R, beyond R

So far, we have explored idiosyncrasies of R we should be aware of when writing programs to handle and analyze large data sets. While this has shown that R has many advantages for working with data, it also revealed some aspects of R that might result in low performance compared other programming languages. A simple generic explanation for this is that R is an interpreted language[4], meaning that when we execute R code, it is processed (statement by statement) by an 'interpreter' that translates the code into machine code (without the user giving any specific instructions). In contrast, when writing code in a 'compiled language', we first have to explicitly compile the code and then run the compiled program. Running code that is already compiled is typically much faster than running R code that has to be interpreted before it can actually be processed by the CPU.

For advanced programmers, R offers various options to directly make use of compiled programs (for example, written in C, C++, or FORTRAN). In fact several of the core R functions installed with the basic R distribution are implemented in one of these lower-level programming languages and the R function we call simply interacts with these functions.

We can actually investigate this by looking at the source code of an R function. When simply typing the name of a function (such as our `import_file()`) to the console, R is printing the function's source code to the console.

```
import_file
```

```
## function(x) {
##           parsed_x <- fread(x)
##           return(parsed_x)
```

[4] https://en.wikipedia.org/wiki/Interpreted_language

```
##       }
## <bytecode: 0x5623fbb40fc8>
```

However, if we do the same for function `sum`, we don't see any actual source code.

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

Instead `.Primitive()` indicates that `sum()` is actually referring to an internal function (in this case implemented in C).

While the use of functions implemented in a lower-level language is a common technique to improve the speed of 'R' functions, it is particularly prominent in the context of functions/packages made to deal with large amounts of data (such as the `data.table` package).

5

Big Data Cleaning and Transformation

Preceding the filtering/selection/aggregation of raw data, data cleaning and transformation typically have to be run on large parts of the overall dataset. In practice, the bottleneck is often a lack of RAM. In the following, we explore two strategies that broadly build on the idea of *virtual memory* (using parts of the hard disk as RAM).

5.1 'Out-of-memory' strategies

Virtual memory is in simple words an approach to combining the RAM and mass storage components in order to cope with a lack of RAM. Modern operating systems come with a virtual memory manager that would automatically handle the swapping between RAM and the hard-disk, when running processes that use up too much RAM. However, a virtual memory manager is not specifically developed to perform this task in the context of data analysis. Several strategies have thus been developed to build on the basic idea of virtual memory in the context of data analysis tasks.

- *Chunked data files on disk*: The data analytics software 'partitions' the large dataset, maps, and stores the chunks of raw data on disk. What is actually 'read' into RAM when importing the data file with this approach is the mapping to the partitions of the actual dataset (the data structure) and some metadata describing the dataset. In R, this approach is implemented in the `ff` package and several packages building on `ff`. In this approach, the usage of disk space and the linking between RAM and files on disk is very explicit (and well visible to the user).

- *Memory mapped files and shared memory*: The data analytics software uses segments of virtual memory for the dataset and allows different programs/processes to access it in the same memory segment. Thus, virtual memory is explicitly allocated for one or several specific data analytics tasks. In R, this approach is prominently implemented in the `bigmemory` package and several packages building on `bigmemory`.

5.1.1 Chunking data with the `ff`-package

Before looking at the more detailed code examples in Walkowiak (2016), we investigate how the `ff` package (and the concept of chunked files) basically works. In order to do so, we first install and load the `ff` and `ffbase` packages, as well as the `pryr` package. We use the already known `flights.csv`-dataset as an example. When importing data via the `ff` package, we first have to set up a directory where `ff` can store the partitioned dataset (recall that this is explicitly/visibly done on disk). As in the code examples of the book, we call this new directory `ffdf` (after `ff`-data.frame).

```
# SET UP --------------

# install.packages(c("ff", "ffbase"))
# load packages
library(ff)
library(ffbase)
library(pryr)

# create directory for ff chunks, and assign directory to ff
system("mkdir ffdf")
options(fftempdir = "ffdf")
```

Now we can read in the data with `read.table.ffdf`. In order to better understand the underlying concept, we record the change in memory in the R environment with `mem_change()`.

```
mem_change(
flights <-
```

```
     read.table.ffdf(file="data/flights.csv",
                     sep=",",
                     VERBOSE=TRUE,
                     header=TRUE,
                     next.rows=100000,
                     colClasses=NA)
)
```

```
##  read.table.ffdf  1..100000  (100000)    csv-read=0.341sec  ffdf-
write=0.037sec
## read.table.ffdf 100001..200000 (100000)  csv-read=0.343sec ffdf-
write=0.025sec
## read.table.ffdf 200001..300000 (100000)  csv-read=0.354sec ffdf-
write=0.024sec
## read.table.ffdf 300001..336776 (36776)  csv-read=0.135sec ffdf-
write=0.015sec
##  csv-read=1.173sec  ffdf-write=0.101sec  TOTAL=1.274sec

## -30.1 MB
```

Note that there are two substantial differences to what we have previously seen when using `fread()`. It takes much longer to import a csv into the ffdf structure. However, the RAM allocated to it is much smaller. This is exactly what we would expect, keeping in mind what `read.table.ffdf()` does in comparison to what `fread()` does.

Now we can actually have a look at the data chunks created by `ff`, as well as how the structure of the dataset is represented in the `flights` object.

```
# show the files in the directory keeping the chunks
list.files("ffdf")
```

```
##   [1] "clone75db307fa93.ff"  "clone75db47da7f7a.ff"
##   [3] "clone75db4bd01be5.ff" "clone75db50764070.ff"
##   [5] "clone8e35360e6853.ff" "clone8e3557fa7930.ff"
##   [7] "clone8e355e79fe83.ff" "clone8e3579525b87.ff"
```

```
##   [9] "clone9ff63cab7f09.ff" "clone9ff64610abb0.ff"
##  [11] "clone9ff65c23f55c.ff" "clone9ff67a994b7.ff"
##  [13] "ff75db238f2277.ff"    "ff75db467c9ca9.ff"
##  [15] "ff75db5b8705e6.ff"    "ff8e351f52edee.ff"
##  [17] "ff8e357931e350.ff"    "ff8e3579acefd2.ff"
##  [19] "ff9ff6166e92fc.ff"    "ff9ff6333e8cb9.ff"
##  [21] "ff9ff67f69f9c8.ff"    "ffdf75db104a61b5.ff"
##  [23] "ffdf75db12381c2.ff"   "ffdf75db12f4060b.ff"
##  [25] "ffdf75db155ac869.ff"  "ffdf75db17569ab7.ff"
##  [27] "ffdf75db19bd3c44.ff"  "ffdf75db1a31be36.ff"
##  [29] "ffdf75db1acb53f1.ff"  "ffdf75db1afbc30.ff"
##  [31] "ffdf75db1bdd3923.ff"  "ffdf75db1efba4e.ff"
##  [33] "ffdf75db1efd5577.ff"  "ffdf75db1fad0746.ff"
##  [35] "ffdf75db20bb1ef4.ff"  "ffdf75db20d27dcd.ff"
##  [37] "ffdf75db275c430a.ff"  "ffdf75db27b0c953.ff"
##  [39] "ffdf75db28d5b230.ff"  "ffdf75db29b096a2.ff"
##  [41] "ffdf75db2a510673.ff"  "ffdf75db2a523ee8.ff"
##  [43] "ffdf75db2b240bf8.ff"  "ffdf75db2b9129e1.ff"
##  [45] "ffdf75db2db0aa48.ff"  "ffdf75db30419c13.ff"
##  [47] "ffdf75db31f15844.ff"  "ffdf75db3314da06.ff"
##  [49] "ffdf75db37721ddc.ff"  "ffdf75db4068263a.ff"
##  [51] "ffdf75db436e625f.ff"  "ffdf75db463b8db5.ff"
##  [53] "ffdf75db46599881.ff"  "ffdf75db473e9042.ff"
##  [55] "ffdf75db48cee0dd.ff"  "ffdf75db4990d124.ff"
##  [57] "ffdf75db4b74a57c.ff"  "ffdf75db50e1e5ae.ff"
##  [59] "ffdf75db51ec6609.ff"  "ffdf75db53cb1709.ff"
##  [61] "ffdf75db54e2b89.ff"   "ffdf75db54f4609d.ff"
##  [63] "ffdf75db561e924f.ff"  "ffdf75db5a2309c4.ff"
##  [65] "ffdf75db5b1ae970.ff"  "ffdf75db5d1285ce.ff"
##  [67] "ffdf75db5de853dd.ff"  "ffdf75db5efcb02.ff"
##  [69] "ffdf75db60d638f6.ff"  "ffdf75db641578be.ff"
##  [71] "ffdf75db662baee6.ff"  "ffdf75db66a014e2.ff"
##  [73] "ffdf75db672c164e.ff"  "ffdf75db67e14b3c.ff"
##  [75] "ffdf75db6a4f2906.ff"  "ffdf75db6ac4f9af.ff"
##  [77] "ffdf75db6b4c5127.ff"  "ffdf75db6be87b71.ff"
##  [79] "ffdf75db6c31c71.ff"   "ffdf75db6dc7633b.ff"
##  [81] "ffdf75db6e06dd86.ff"  "ffdf75db6e99304b.ff"
```

```
##  [83] "ffdf75db725a257d.ff"  "ffdf75db727076ab.ff"
##  [85] "ffdf75db72b0340e.ff"  "ffdf75db74ef3a8a.ff"
##  [87] "ffdf75db76e92b9f.ff"  "ffdf75db77643f09.ff"
##  [89] "ffdf75db78083020.ff"  "ffdf75db7a0eb09f.ff"
##  [91] "ffdf75db7b03a8df.ff"  "ffdf75db7deeba85.ff"
##  [93] "ffdf75db7ee7bfbb.ff"  "ffdf75db80afc77.ff"
##  [95] "ffdf75db9f8f75f.ff"   "ffdf75dbae90994.ff"
##  [97] "ffdf75dbafc362c.ff"   "ffdf75dbb273f69.ff"
##  [99] "ffdf75dbd1cd10.ff"    "ffdf8e3514dfade1.ff"
## [101] "ffdf8e3515fa8211.ff"  "ffdf8e3518d21571.ff"
## [103] "ffdf8e351c061b0e.ff"  "ffdf8e351d03c750.ff"
## [105] "ffdf8e351f24f13c.ff"  "ffdf8e35204f468.ff"
## [107] "ffdf8e3520749287.ff"  "ffdf8e35213843be.ff"
## [109] "ffdf8e352157e257.ff"  "ffdf8e35238a0f83.ff"
## [111] "ffdf8e352390a28e.ff"  "ffdf8e3523efdb6a.ff"
## [113] "ffdf8e35243f7204.ff"  "ffdf8e35266058d8.ff"
## [115] "ffdf8e352719254a.ff"  "ffdf8e352ac66b45.ff"
## [117] "ffdf8e352b16ccee.ff"  "ffdf8e352c9928d7.ff"
## [119] "ffdf8e352ead5d77.ff"  "ffdf8e35328682a.ff"
## [121] "ffdf8e3532e1e0bb.ff"  "ffdf8e35377dee1d.ff"
## [123] "ffdf8e3537cbb5c8.ff"  "ffdf8e353803181e.ff"
## [125] "ffdf8e3538a78b41.ff"  "ffdf8e353944cdf5.ff"
## [127] "ffdf8e353969089.ff"   "ffdf8e3539cc5f85.ff"
## [129] "ffdf8e353addce7a.ff"  "ffdf8e353be15cf0.ff"
## [131] "ffdf8e353d328ef5.ff"  "ffdf8e353d8d734a.ff"
## [133] "ffdf8e3540097a02.ff"  "ffdf8e3541eb9253.ff"
## [135] "ffdf8e3545c5183f.ff"  "ffdf8e354614e0dc.ff"
## [137] "ffdf8e3546234b48.ff"  "ffdf8e35481464cb.ff"
## [139] "ffdf8e35489d7a69.ff"  "ffdf8e354b58974e.ff"
## [141] "ffdf8e354bfb4ef3.ff"  "ffdf8e354c620628.ff"
## [143] "ffdf8e354c765b75.ff"  "ffdf8e354d797e22.ff"
## [145] "ffdf8e354ede9f1e.ff"  "ffdf8e354fc4b1cb.ff"
## [147] "ffdf8e354fe5a135.ff"  "ffdf8e3556bbe7be.ff"
## [149] "ffdf8e35585756f5.ff"  "ffdf8e355a913a9b.ff"
## [151] "ffdf8e355b526101.ff"  "ffdf8e355db9a2c5.ff"
## [153] "ffdf8e355dbc3aef.ff"  "ffdf8e355f80a38f.ff"
## [155] "ffdf8e355fd1385b.ff"  "ffdf8e3560253dd.ff"
```

```
## [157] "ffdf8e35618f7b7f.ff"  "ffdf8e356624a0da.ff"
## [159] "ffdf8e3567b93c51.ff"  "ffdf8e356b555a53.ff"
## [161] "ffdf8e356b86f764.ff"  "ffdf8e356d73161.ff"
## [163] "ffdf8e356f8dfd2f.ff"  "ffdf8e3570932676.ff"
## [165] "ffdf8e35763e2d39.ff"  "ffdf8e3577a6492c.ff"
## [167] "ffdf8e357862a5f.ff"   "ffdf8e3579b7481c.ff"
## [169] "ffdf8e357a7ffc73.ff"  "ffdf8e357b0c82.ff"
## [171] "ffdf8e357d0c49e9.ff"  "ffdf8e357def0110.ff"
## [173] "ffdf8e3592a6f2a.ff"   "ffdf8e359ff998c.ff"
## [175] "ffdf8e35a97707d.ff"   "ffdf8e35bfade79.ff"
## [177] "ffdf8e35fd54512.ff"   "ffdf9ff6110d9311.ff"
## [179] "ffdf9ff61187c490.ff"  "ffdf9ff6152ac306.ff"
## [181] "ffdf9ff61c2ecc81.ff"  "ffdf9ff62432108c.ff"
## [183] "ffdf9ff624d6cbd8.ff"  "ffdf9ff62567feaf.ff"
## [185] "ffdf9ff625f9e3d2.ff"  "ffdf9ff62618d903.ff"
## [187] "ffdf9ff6296ad395.ff"  "ffdf9ff629a507d5.ff"
## [189] "ffdf9ff629cbcef2.ff"  "ffdf9ff62c372317.ff"
## [191] "ffdf9ff62cc39202.ff"  "ffdf9ff62d35fc65.ff"
## [193] "ffdf9ff62d599839.ff"  "ffdf9ff633d492f0.ff"
## [195] "ffdf9ff63495cbc8.ff"  "ffdf9ff634fd2ad2.ff"
## [197] "ffdf9ff6395cd393.ff"  "ffdf9ff63e0bcda8.ff"
## [199] "ffdf9ff63e64c093.ff"  "ffdf9ff64079d5a1.ff"
## [201] "ffdf9ff640be3484.ff"  "ffdf9ff642384989.ff"
## [203] "ffdf9ff642644cae.ff"  "ffdf9ff6434d2842.ff"
## [205] "ffdf9ff643a34c.ff"    "ffdf9ff648526631.ff"
## [207] "ffdf9ff64a0de165.ff"  "ffdf9ff64c2ac858.ff"
## [209] "ffdf9ff64cfd0de8.ff"  "ffdf9ff64fcab41a.ff"
## [211] "ffdf9ff64fe4a7f6.ff"  "ffdf9ff650563688.ff"
## [213] "ffdf9ff650a0349b.ff"  "ffdf9ff65108664e.ff"
## [215] "ffdf9ff651294af2.ff"  "ffdf9ff6547a60a4.ff"
## [217] "ffdf9ff654e3306a.ff"  "ffdf9ff65569dd2e.ff"
## [219] "ffdf9ff655ad807a.ff"  "ffdf9ff65659d736.ff"
## [221] "ffdf9ff657887beb.ff"  "ffdf9ff65a35c247.ff"
## [223] "ffdf9ff65bac7761.ff"  "ffdf9ff65ca9604e.ff"
## [225] "ffdf9ff65e6fe4d9.ff"  "ffdf9ff66015dce1.ff"
## [227] "ffdf9ff6602d6eaa.ff"  "ffdf9ff660fb5d.ff"
## [229] "ffdf9ff663959724.ff"  "ffdf9ff664b123d2.ff"
```

```
## [231] "ffdf9ff6654a181b.ff"  "ffdf9ff66550a179.ff"
## [233] "ffdf9ff66657df92.ff"  "ffdf9ff666899228.ff"
## [235] "ffdf9ff66752f40.ff"   "ffdf9ff6676ad90.ff"
## [237] "ffdf9ff66fc2c457.ff"  "ffdf9ff66fd9a106.ff"
## [239] "ffdf9ff67007c538.ff"  "ffdf9ff6713b01bb.ff"
## [241] "ffdf9ff6736f35c7.ff"  "ffdf9ff67554004d.ff"
## [243] "ffdf9ff6759640a4.ff"  "ffdf9ff6760573f.ff"
## [245] "ffdf9ff676501a5a.ff"  "ffdf9ff67aad6e23.ff"
## [247] "ffdf9ff67caa956.ff"   "ffdf9ff67e922a3e.ff"
## [249] "ffdf9ff6848f539.ff"   "ffdf9ff6939322e.ff"
## [251] "ffdf9ff6a8f88eb.ff"   "ffdf9ff6aa707f0.ff"
## [253] "ffdf9ff6aefa2ff.ff"   "ffdf9ff6b6d3522.ff"
## [255] "ffdf9ff6dbb3e0b.ff"   "ffdfb3f6135820f4.ff"
## [257] "ffdfb3f6142c50de.ff"  "ffdfb3f61cc2e3dd.ff"
## [259] "ffdfb3f61ec07a7d.ff"  "ffdfb3f61f425c93.ff"
## [261] "ffdfb3f62296082d.ff"  "ffdfb3f624e08b.ff"
## [263] "ffdfb3f628f0eaa8.ff"  "ffdfb3f6344b78fe.ff"
## [265] "ffdfb3f6446ce386.ff"  "ffdfb3f64c83ea00.ff"
## [267] "ffdfb3f65fb8171c.ff"  "ffdfb3f66c2a1d45.ff"
## [269] "ffdfb3f6723850c1.ff"  "ffdfb3f674888c10.ff"
## [271] "ffdfb3f67562615a.ff"  "ffdfb3f67d8ec341.ff"
## [273] "ffdfb3f67db25a8e.ff"  "ffdfb3f68550586.ff"
```

```
# investigate the structure of the object created in the R environment
str(flights)
```

```
## List of 3
##  $ virtual: 'data.frame':    19 obs. of  7 variables:
##  .. $ VirtualVmode    : chr  "integer" "integer" "integer" "integer" ...
##  .. $ AsIs            : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  .. $ VirtualIsMatrix : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  .. $ PhysicalIsMatrix : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##   .. $ PhysicalElementNo: int  1 2 3 4 5 6 7 8 9 10 ...
##   .. $ PhysicalFirstCol : int  1 1 1 1 1 1 1 1 1 ...
##   .. $ PhysicalLastCol  : int  1 1 1 1 1 1 1 1 1 ...
##   .. - attr(*, "Dim")= int [1:2] 336776 19
##   .. - attr(*, "Dimorder")= int [1:2] 1 2
```

```
##  $ physical: List of 19
##  .. $ year          : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ month         : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ day           : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
```

```
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ dep_time      : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ sched_dep_time: list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
```

```
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ dep_delay     : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ arr_time      : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
```

```
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ sched_arr_time: list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ arr_delay     : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ carrier       : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
```

```
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "Levels")= chr [1:16] "9E" "AA" "AS" "B6" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ flight        : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ tailnum       : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
```

```
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
## .. .. ..- attr(*, "Levels")= chr [1:4044] "" "N0EGMQ" "N10156" "N102UW" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ origin        : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "Levels")= chr [1:3] "EWR" "JFK" "LGA"
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ dest          : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
```

```
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
## .. .. ..- attr(*, "Levels")= chr [1:105] "ABQ" "ACK" "ALB" "ATL" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ air_time      : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ distance      : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
```

```
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ hour          : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ minute        : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "class")= chr "virtual"
```

```
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ time_hour     : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 336776
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## ..  .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdf/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 336776
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##    ..    ..  ..-  attr(*,  "Levels")=  chr  [1:6936]  "2013-01-
01T10:00:00Z" "2013-01-01T11:00:00Z" "2013-01-01T12:00:00Z" "2013-
01-01T13:00:00Z" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  ..  .. ..- attr(*, "class")= chr "virtual"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  $ row.names:  NULL
## - attributes: List of 2
##  .. $ names: chr [1:3] "virtual" "physical" "row.names"
##  .. $ class: chr "ffdf"
```

5.1.2 Memory mapping with `bigmemory`

The `bigmemory`-package handles data in matrices, and therefore only accepts variables in the same data type. Before importing data via the `bigmemory`-package, we thus have to ensure that all variables in the raw data can be imported in a common type. This example follows the example of the package authors given here[1].[2]

[1] https://cran.r-project.org/web/packages/bigmemory/vignettes/Overview.pdf

[2] We only use a fraction of the data used in the package vignette example, the full raw data used there can be downloaded here[3].

```
# SET UP ----------------

# load packages
library(bigmemory)
library(biganalytics)

# import the data
flights <- read.big.matrix("data/flights.csv",
                     type="integer",
                     header=TRUE,
                     backingfile="flights.bin",
                     descriptorfile="flights.desc")
```

Note that, similar to the `ff`-example, `read.big.matrix()` initiates a local file-backing `flights.bin` on disk which is linked to the `flights`-object in RAM. From looking at the imported file, we see that various variable values have been discarded. This is due to the fact that we have forced all variables to be of type `"integer"` when importing the dataset.

```
summary(flights)
```

```
##                      min        max       mean
## year             2013.000   2013.000   2013.000
## month               1.000     12.000      6.549
## day                 1.000     31.000     15.711
## dep_time            1.000   2400.000   1349.110
## sched_dep_time    106.000   2359.000   1344.255
## dep_delay         -43.000   1301.000     12.639
## arr_time            1.000   2400.000   1502.055
## sched_arr_time      1.000   2359.000   1536.380
## arr_delay         -86.000   1272.000      6.895
## carrier             9.000      9.000      9.000
## flight              1.000   8500.000   1971.924
## tailnum
## origin
## dest
```

```
## air_time           20.000    695.000    150.686
## distance           17.000   4983.000   1039.913
## hour                1.000     23.000     13.180
## minute              0.000     59.000     26.230
## time_hour        2013.000   2014.000   2013.000
##                       NAs
## year                0.000
## month               0.000
## day                 0.000
## dep_time         8255.000
## sched_dep_time      0.000
## dep_delay        8255.000
## arr_time         8713.000
## sched_arr_time      0.000
## arr_delay        9430.000
## carrier        318316.000
## flight              0.000
## tailnum        336776.000
## origin         336776.000
## dest           336776.000
## air_time         9430.000
## distance            0.000
## hour                0.000
## minute              0.000
## time_hour           0.000
```

5.2 Typical cleaning tasks

- Normalize/standardize.
- Code additional variables (indicators, strings to categorical, etc.).
- Remove, add covariates.
- Merge data sets.
- Set data types.

1. Import raw data.

2. Clean/transform.
3. Store for analysis.
    - Write to file.
    - Write to database.

- RAM:
  - Raw data does not fit into memory.
  - Transformations enlarge RAM allocation (copying).
- Mass Storage: Reading/Writing
- CPU: Parsing (data types)

5.2.1 Data Preparation with `ff`

5.2.1.1 Set up

The following examples are based on Walkowiak (2016), Chapter 3. You can download the original datasets used in these examples from the book's GitHub repository[4]. The set up for our analysis script involves the loading of the `ff` and `ffbase` packages, the intitiation of fix variables to hold the paths to the data sets, as well as the creation and assignment of a new local directory `ffdf` in which the binary flat files-partitioned chunks of the original datasets will be stored.

```
## SET UP ------------------------

# create and set directory for ff files
system("mkdir ffdf")
options(fftempdir = "ffdf")

# load packages
library(ff)
library(ffbase)
library(pryr)

# fix vars
```

[4] https://github.com/PacktPublishing/Big-Data-Analytics-with-R/tree/master/Chapter%203

```
FLIGHTS_DATA <- "data/flights_sep_oct15.txt"
AIRLINES_DATA <- "data/airline_id.csv"
```

5.2.1.2 Data import

In a first step we read (or 'upload') the data into R. This step involves the creation of the binary chunked files as well as the mapping of these files and the metadata. In comparison to the traditional `read.csv` approach, you will notice two things. On the one hand the data import takes longer, on the other hand it uses up much less RAM than than with `read.csv`.

```
# DATA IMPORT ------------------

# check memory used
mem_used()
```

```
## 1.09 GB
```

```
# 1. Upload flights_sep_oct15.txt and airline_id.csv files from flat files.

system.time(flights.ff <- read.table.ffdf(file=FLIGHTS_DATA,
                                          sep=",",
                                          VERBOSE=TRUE,
                                          header=TRUE,
                                          next.rows=100000,
                                          colClasses=NA))
```

```
##  read.table.ffdf  1..100000  (100000)    csv-read=0.444sec  ffdf-
write=0.05sec
## read.table.ffdf 100001..200000 (100000)  csv-read=0.458sec ffdf-
write=0.038sec
## read.table.ffdf 200001..300000 (100000)  csv-read=0.455sec ffdf-
write=0.045sec
## read.table.ffdf 300001..400000 (100000)  csv-read=0.476sec ffdf-
write=0.05sec
```

```
## read.table.ffdf 400001..500000 (100000)  csv-read=0.476sec ffdf-
write=0.044sec
## read.table.ffdf 500001..600000 (100000)  csv-read=0.457sec ffdf-
write=0.037sec
## read.table.ffdf 600001..700000 (100000)  csv-read=0.463sec ffdf-
write=0.037sec
## read.table.ffdf 700001..800000 (100000)  csv-read=0.463sec ffdf-
write=0.041sec
## read.table.ffdf 800001..900000 (100000)  csv-read=0.455sec ffdf-
write=0.038sec
## read.table.ffdf 900001..951111 (51111)  csv-read=0.233sec ffdf-
write=0.035sec
##  csv-read=4.38sec  ffdf-write=0.415sec  TOTAL=4.795sec
```

```
##    user  system elapsed
##   4.665   0.124   4.796
```

```
system.time(airlines.ff <- read.csv.ffdf(file= AIRLINES_DATA,
                            VERBOSE=TRUE,
                            header=TRUE,
                            next.rows=100000,
                            colClasses=NA))
```

```
##   read.table.ffdf   1..1607   (1607)     csv-read=0.004sec   ffdf-
write=0.003sec
##  csv-read=0.004sec  ffdf-write=0.003sec  TOTAL=0.007sec
```

```
##    user  system elapsed
##   0.004   0.002   0.007
```

```
# check memory used
mem_used()
```

```
## 1.09 GB
```

Comparison with `read.table`

```
##Using read.table()
system.time(flights.table <- read.table(FLIGHTS_DATA,
                                        sep=",",
                                        header=TRUE))
```

```
##    user  system elapsed
##   4.462   0.068   4.530
```

```
gc()
```

```
##             used   (Mb) gc trigger   (Mb)  max used
## Ncells   1301241   69.5    2100091  112.2   2100091
## Vcells 145069021 1106.8  213476124 1628.7 212853893
##          (Mb)
## Ncells  112.2
## Vcells 1624.0
```

```
system.time(airlines.table <- read.csv(AIRLINES_DATA,
                                       header = TRUE))
```

```
##    user  system elapsed
##   0.002   0.000   0.001
```

```
# check memory used
mem_used()
```

```
## 1.23 GB
```

5.2.1.3 Inspect imported files

A particularly useful aspect of working with the ff-package and the packages building on them is that many of the simple R functions that work on usual data.frames in RAM also work on ffdfs. Hence, without actually having loaded the entire raw data of a large dataset into RAM, we can quickly get an overview of the key characteristics such as the number of observations and the number of variables.

```
# 2. Inspect the ffdf objects.
## For flights.ff object:
class(flights.ff)
```

```
## [1] "ffdf"
```

```
dim(flights.ff)
```

```
## [1] 951111     28
```

```
## For airlines.ff object:
class(airlines.ff)
```

```
## [1] "ffdf"
```

```
dim(airlines.ff)
```

```
## [1] 1607    2
```

5.2.1.4 Data cleaning and transformation

After inspecting the data, we go through several steps of cleaning and transformation, with the goal of then merging the two datasets. That is, we want to create a new dataset that contains detailed flights information but with additional information on the carriers/airlines. First, we want to rename some of the variables.

```
# step 1:
## Rename "Code" variable from airlines.ff to "AIRLINE_ID" and "Description" into "AIRLINE_NM"
names(airlines.ff) <- c("AIRLINE_ID", "AIRLINE_NM")
names(airlines.ff)
```

```
## [1] "AIRLINE_ID" "AIRLINE_NM"
```

```
str(airlines.ff[1:20,])
```

```
## 'data.frame':    20 obs. of  2 variables:
##  $ AIRLINE_ID: int  19031 19032 19033 19034 19035 19036 19037 19038 19039 19040 ...
##        $    AIRLINE_NM:    Factor    w/    1607    levels    "40-
Mile Air: Q5",..: 945 1025 503 721 64 725 1194 99 1395 276 ...
```

Now we can join the two datasets via the unique airline identifier `"AIRLINE_ID"`. Note that these kind of operations would usually take up substantially more RAM on the spot, if both original datasets would also be fully loaded into RAM. As illustrated by the `mem_change()`-function, this is not the case here. All that is needed is a small chunk of RAM to keep the metadata and mapping-information of the new `ffdf` object, all the actual data is cached on the hard disk.

```
# merge of ffdf objects
mem_change(flights.data.ff <- merge.ffdf(flights.ff, airlines.ff, by="AIRLINE_ID"))
```

```
## 781 kB
```

```
#The new object is only 551.2 Kb in size
class(flights.data.ff)
```

```
## [1] "ffdf"
```

```
dim(flights.data.ff)
```

```
## [1] 951111     29
```

```
names(flights.data.ff)
```

```
##  [1] "YEAR"              "MONTH"
##  [3] "DAY_OF_MONTH"      "DAY_OF_WEEK"
##  [5] "FL_DATE"           "UNIQUE_CARRIER"
##  [7] "AIRLINE_ID"        "TAIL_NUM"
##  [9] "FL_NUM"            "ORIGIN_AIRPORT_ID"
## [11] "ORIGIN"            "ORIGIN_CITY_NAME"
## [13] "ORIGIN_STATE_NM"   "ORIGIN_WAC"
```

```
## [15] "DEST_AIRPORT_ID"   "DEST"
## [17] "DEST_CITY_NAME"    "DEST_STATE_NM"
## [19] "DEST_WAC"          "DEP_TIME"
## [21] "DEP_DELAY"         "ARR_TIME"
## [23] "ARR_DELAY"         "CANCELLED"
## [25] "CANCELLATION_CODE" "DIVERTED"
## [27] "AIR_TIME"          "DISTANCE"
## [29] "AIRLINE_NM"
```

Inspect difference to in-memory operation

```
##For flights.table:
names(airlines.table) <- c("AIRLINE_ID", "AIRLINE_NM")
names(airlines.table)
```

```
## [1] "AIRLINE_ID" "AIRLINE_NM"
```

```
str(airlines.table[1:20,])
```

```
## 'data.frame':    20 obs. of  2 variables:
##  $ AIRLINE_ID: int  19031 19032 19033 19034 19035 19036 19037 19038 19039 19040 ...
##  $ AIRLINE_NM: chr  "Mackey International Inc.: MAC" "Munz Northern Airlines Inc.: XY" "Cochise
```

```
# check memory usage of merge in RAM
mem_change(flights.data.table <- merge(flights.table,
                                       airlines.table,
                                       by="AIRLINE_ID"))
```

```
## 160 MB
```

```
#The new object is already 105.7 Mb in size
#A rapid spike in RAM use when processing
```

5.2.1.5 Subsetting

Now, we want to filter out some observations as well as select only specific variables for a subset of the overall dataset.

```
mem_used()
```

```
## 1.39 GB
```

```
# Subset the ffdf object flights.data.ff:
subs1.ff <- subset.ffdf(flights.data.ff, CANCELLED == 1,
                        select = c(FL_DATE, AIRLINE_ID,
                                   ORIGIN_CITY_NAME,
                                   ORIGIN_STATE_NM,
                                   DEST_CITY_NAME,
                                   DEST_STATE_NM,
                                   CANCELLATION_CODE))

dim(subs1.ff)
```

```
## [1] 4529    7
```

```
mem_used()
```

```
## 1.39 GB
```

5.2.1.6 Save/load/export ffdf-files

In order to better organize and easily reload newly created `ffdf`s, we can explicitly save them to disk.

```
# Save a newly created ffdf object to a data file:

save.ffdf(subs1.ff, overwrite = TRUE) #7 files (one for each column) created in the ffdb direc
```

If we want to reload a previously saved `ffdf`, we do not have to go through the chunking of a raw data file again, but can very quickly load the data mapping and metadata into RAM in order to further work with the data (stored on disk).

```
# Loading previously saved ffdf files:
rm(subs1.ff)
gc()
```

```
##             used   (Mb) gc trigger   (Mb)  max used
## Ncells   1321733   70.6    4565612  243.9   3203308
## Vcells 165059521 1259.4  256251348 1955.1 212853893
##          (Mb)
## Ncells  171.1
## Vcells 1624.0
```

```
load.ffdf("ffdb")
# check the class and structure of the loaded data
class(subs1.ff)
```

```
## [1] "ffdf"
```

```
str(subs1.ff)
```

```
## List of 3
##  $ virtual: 'data.frame':    7 obs. of  7 variables:
##  .. $ VirtualVmode    : chr  "integer" "integer" "integer" "integer" ...
##  .. $ AsIs            : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  .. $ VirtualIsMatrix : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  .. $ PhysicalIsMatrix : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##   .. $ PhysicalElementNo: int  1 2 3 4 5 6 7
##   .. $ PhysicalFirstCol : int  1 1 1 1 1 1 1
##   .. $ PhysicalLastCol  : int  1 1 1 1 1 1 1
##   .. - attr(*, "Dim")= int [1:2] 4529 7
##   .. - attr(*, "Dimorder")= int [1:2] 1 2
##  $ physical: List of 7
##  .. $ FL_DATE          : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
```

```
##  .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "Levels")= chr [1:61] "2015-09-01" "2015-09-
02" "2015-09-03" "2015-09-04" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ AIRLINE_ID       : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
##  .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ ORIGIN_CITY_NAME : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
##  .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
```

```
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
## .. .. ..- attr(*, "Levels")= chr [1:305] "Abilene, TX" "Akron, OH" "Albany, GA" "Albany, NY" ..
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ ORIGIN_STATE_NM  : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
## .. .. ..- attr(*, "Levels")= chr [1:52] "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ DEST_CITY_NAME   : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
```

```
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
## .. .. ..- attr(*, "Levels")= chr [1:306] "Abilene, TX" "Akron, OH" "Albany, GA" "Albany, NY" ..
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ DEST_STATE_NM    : list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
## .. .. ..- attr(*, "Levels")= chr [1:52] "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  .. $ CANCELLATION_CODE: list()
##  ..  ..- attr(*, "physical")=Class 'ff_pointer' <externalptr>
##  ..  .. ..- attr(*, "vmode")= chr "integer"
##  ..  .. ..- attr(*, "maxlength")= int 4529
##  ..  .. ..- attr(*, "pattern")= chr "ffdf"
## .. .. ..- attr(*, "filename")= chr "/home/umatter/Dropbox/Teaching/HSG/BigData/BigData/ffdb/
##  ..  .. ..- attr(*, "pagesize")= int 65536
##  ..  .. ..- attr(*, "finalizer")= chr "close"
##  ..  .. ..- attr(*, "finonexit")= logi TRUE
##  ..  .. ..- attr(*, "readonly")= logi FALSE
##  ..  .. ..- attr(*, "caching")= chr "mmnoflush"
##  ..  ..- attr(*, "virtual")= list()
##  ..  .. ..- attr(*, "Length")= int 4529
##  ..  .. ..- attr(*, "Symmetric")= logi FALSE
##  ..  .. ..- attr(*, "Levels")= chr [1:4] "" "A" "B" "C"
```

```
##  ..  .. ..- attr(*, "ramclass")= chr "factor"
##  .. .. - attr(*, "class") =  chr [1:2] "ff_vector" "ff"
##  $ row.names:  NULL
## - attributes: List of 2
##  .. $ names: chr [1:2] "virtual" "physical"
##  .. $ class: chr "ffdf"
```

```
dim(subs1.ff)
```

```
## [1] 4529    7
```

```
dimnames(subs1.ff)
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] "FL_DATE"           "AIRLINE_ID"
## [3] "ORIGIN_CITY_NAME"  "ORIGIN_STATE_NM"
## [5] "DEST_CITY_NAME"    "DEST_STATE_NM"
## [7] "CANCELLATION_CODE"
```

In case we want to store an `ffdf` dataset in a format more accessible for other users (such as csv), we can do so as follows. This last step is also quite common in practice. The initial raw data set is very large, thus we perform all the theoretically very memory-intense tasks of preparing the analytic dataset via `ff` and then store the (often much smaller) analytic dataset in a more accessible csv file in order to later read it into RAM and run more computationally intense analyses directly in RAM.

```
#  Export subs1.ff into CSV and TXT files:
write.csv.ffdf(subs1.ff, "subset1.csv")
```

6

Data Aggregation

6.1 Data aggregation: The 'split-apply-combine' strategy

The 'split-apply-combine' strategy plays an important role in many data analysis tasks, ranging from data preparation to summary statistics and model-fitting.[1] The strategy can be defined as "break up a problem into manageable pieces, operate on each piece independently and then put all the pieces back together." (Wickham, 2011, p. 1)

Many R users are familiar with the basic concept of split-apply-combine implemented in the `plyr`-package intended for the usual in-memory operations (data set fits into RAM). Here, we explore the options for split-apply-combine approaches to large data sets that do not fit into RAM.

6.2 Data aggregation tutorial

In this tutorial we explore the world of New York's famous Yellow Caps. The NYC Taxi & Limousine Commission (TLC) provides detailed data on all trip records including pick-up and drop-off times/locations. When combining all available trip records (2009-2018), we get a rather large data set of over 200GB. The code examples below illustrate how to collect and compile the entire data set. In order to avoid long computing times, the code examples shown below are based on a small

[1] Moreover, 'split-apply-combine' is closely related to a core strategy of Big Data analytics with distributed systems: Map/Reduce - more on this in the lecture on distributed systems.

sub-set of the actual raw data (however, all examples involving virtual memory, are in theory scalable to the extent of the entire raw data set).

6.2.1 Gathering and Compilation of all the raw data

The raw data consists of several monthly CSV-files and can be downloaded via the TLC's website[2]. The following short R-script automates the downloading of all available trip-record files. *NOTE*: Downloading all files can take several hours and will occupy over 200GB!

```
##################################
# Fetch all TLC trip recrods
# Data source:
# https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
# Input: Monthly csv files from urls
# Output: one large csv file
# UM, St. Gallen, January 2019
##################################

# SET UP -----------------

# load packages
library(data.table)
library(rvest)
library(httr)

# fix vars
BASE_URL <- "https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2018-01.csv"
OUTPUT_PATH <- "data/tlc_trips.csv"
START_DATE <- as.Date("2009-01-01")
END_DATE <- as.Date("2018-06-01")


# BUILD URLS -----------
```

[2] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

```
# parse base url
base_url <- gsub("2018-01.csv", "", BASE_URL)
# build urls
dates <- seq(from= START_DATE,
                  to = END_DATE,
                  by = "month")
year_months <- gsub("-01$", "", as.character(dates))
data_urls <- paste0(base_url, year_months, ".csv")

# FETCH AND STACK CSVS ----------------

# download, parse all files, write them to one csv
for (url in data_urls) {

     # download to temporary file
     tmpfile <- tempfile()
     download.file(url, destfile = tmpfile)

     # parse downloaded file, write to output csv, remove tempfile
     csv_parsed <- fread(tmpfile)
     fwrite(csv_parsed,
            file =  OUTPUT_PATH,
            append = TRUE)
     unlink(tmpfile)

}
```

6.2.2 Data aggregation with chunked data files

In this first part of the data aggregation tutorial, we will focus on the `ff`-based approach to employ parts of the hard disk as 'virtual memory'. This means, all of the examples are easily scalable without risking too much memory pressure. Given the size of the entire TLC database

(over 200GB), we will only use one million taxi trips records of January 2009.[3]

6.2.2.1 Data import

First, we read the raw taxi trips records into R with the `ff`-package.

```
# load packages
library(ff)
library(ffbase)

# set up the ff directory (for data file chunks)
if (!dir.exists("fftaxi")){
     system("mkdir fftaxi")
}
options(fftempdir = "fftaxi")

# import a few lines of the data, setting the column classes explicitly
col_classes <- c(V1 = "factor",
                 V2 = "POSIXct",
                 V3 = "POSIXct",
                 V4 = "integer",
                 V5 = "numeric",
                 V6 = "numeric",
                 V7 = "numeric",
                 V8 = "numeric",
                 V9 = "numeric",
                 V10 = "numeric",
                 V11 = "numeric",
                 V12 = "factor",
                 V13 = "numeric",
                 V14 = "numeric",
                 V15 = "factor",
                 V16 = "numeric",
```

[3] Note that the code examples below could also be run based on the entire TLC database (provided that there is enough hard-disk space available). But, creating the `ff` chunked file structure for a 200GB CSV would take hours or even days.

```
                 V17 = "numeric",
                 V18 = "numeric")

# import the first one million observations
taxi <- read.table.ffdf(file = "data/tlc_trips.csv",
                        sep = ",",
                        header = TRUE,
                        next.rows = 100000,
                        colClasses= col_classes,
                        nrows = 1000000
                        )
```

Following the data documentation provided by TLC, we give the columns of our data set more meaningful names and remove the empty columns (some covariates are only collected in later years).

```
# first, we remove the empty vars V8 and V9
taxi$V8 <- NULL
taxi$V9 <- NULL


# set covariate names according to the data dictionary
# see https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf
# note instead of taxizonne ids, long/lat are provided

varnames <- c("vendor_id",
              "pickup_time",
              "dropoff_time",
              "passenger_count",
              "trip_distance",
              "start_lat",
              "start_long",
              "dest_lat",
              "dest_long",
              "payment_type",
              "fare_amount",
```

```
              "extra",
              "mta_tax",
              "tip_amount",
              "tolls_amount",
              "total_amount")
names(taxi) <- varnames
```

When inspecting the factor variables of the data set, we notice that some of the values are not standardized/normalized and the resulting factor levels are, therefore, somewhat ambiguous. We better clean this before getting into data aggregation tasks. Note the `ff`-specific syntax needed to recode the factor.

```
# inspect the factor levels
levels(taxi$payment_type)
```

```
## [1] "Cash"      "CASH"      "Credit"    "CREDIT"
## [5] "Dispute"   "No Charge"
```

```
# recode them
levels(taxi$payment_type) <- tolower(levels(taxi$payment_type))
taxi$payment_type <- ff(taxi$payment_type,
                        levels = unique(levels(taxi$payment_type)),
                        ramclass = "factor")
# check result
levels(taxi$payment_type)
```

```
## [1] "cash"      "credit"    "dispute"   "no charge"
```

6.2.2.2 Aggregation with split-apply-combine

First, we have a look at whether trips paid with credit card tend to involve lower tip amounts than trips paid by cash. In order to do so, we create a table that shows the average amount of tip paid for each payment-type category.

In simple words, this means we first split the data set into subsets,

each of which containing all observations belonging to a distinct payment type. Then, we compute the arithmetic mean of the tip-column of each of these subsets. Finally, we combine all of these results in one table (i.e., the split-apply-combine strategy). When working with `ff`, the `ffdfply()`-function provides a user-friendly implementation of split-apply-combine type of tasks.

```
# load packages
library(doBy)

# split-apply-combine procedure on data file chunks
tip_pcategory <- ffdfdply(taxi,
                          split = taxi$payment_type,
                          BATCHBYTES = 100000000,
                          FUN = function(x) {
                               summaryBy(tip_amount~payment_type,
                                         data = x,
                                         FUN = mean,
                                         na.rm = TRUE)})
```

```
## 2022-01-20 16:13:12, calculating split sizes

## 2022-01-20 16:13:12, building up split locations

## 2022-01-20 16:13:13, working on split 1/2, extracting data in RAM of 1 split elements, totalling

## 2022-01-20 16:13:13, ... applying FUN to selected data

## 2022-01-20 16:13:13, ... appending result to the output ffdf

## 2022-01-20 16:13:13, working on split 2/2, extracting data in RAM of 3 split elements, totalling

## 2022-01-20 16:13:14, ... applying FUN to selected data

## 2022-01-20 16:13:14, ... appending result to the output ffdf
```

Note how the output describes the procedure step by step. Now we can have a look at the resulting summary statistic in the form of a `data.frame()`.

```
as.data.frame(tip_pcategory)
```

```
##   payment_type tip_amount.mean
## 1         cash       0.0008162
## 2       credit       2.1619737
## 3      dispute       0.0035075
## 4    no charge       0.0041056
```

The result would go against our initial hypothesis. However, the comparison is a little flawed. If trips paid by credit card also tend to be longer, the result is not too surprising. We should thus look at the share of tip (or percentage), given the overall amount paid for the trip.

We add an additional variable `percent_tip` and then repeat the aggregation exercise for this variable.

```
# add additional column with the share of tip
taxi$percent_tip <- (taxi$tip_amount/taxi$total_amount)*100

# recompute the aggregate stats
tip_pcategory <- ffdfdply(taxi,
                          split = taxi$payment_type,
                          BATCHBYTES = 100000000,
                          FUN = function(x) {
                               summaryBy(percent_tip~payment_type, # note the difference here
                                         data = x,
                                         FUN = mean,
                                         na.rm = TRUE)})
```

```
## 2022-01-20 16:13:14, calculating split sizes

## 2022-01-20 16:13:14, building up split locations

## 2022-01-20 16:13:14, working on split 1/2, extracting data in RAM of 1 split elements, totalling

## 2022-01-20 16:13:14, ... applying FUN to selected data

## 2022-01-20 16:13:14, ... appending result to the output ffdf
```

```
## 2022-01-20 16:13:14, working on split 2/2, extracting data in RAM of 3 split elements, totalling
## 2022-01-20 16:13:15, ... applying FUN to selected data
## 2022-01-20 16:13:15, ... appending result to the output ffdf
```

```
# show result as data frame
as.data.frame(tip_pcategory)
```

```
##   payment_type percent_tip.mean
## 1         cash         0.005978
## 2       credit        16.004173
## 3      dispute         0.045660
## 4    no charge         0.040433
```

6.2.2.3 Cross-tabulation of `ff` vectors

Also in relative terms, trips paid by credit card tend to be tipped more. However, are there actually many trips paid by credit card? In order to figure this out, we count the number of trips per payment type by applying the `table.ff`-function provided in `ffbase`.

```
table.ff(taxi$payment_type)
```

```
##
##      cash    credit   dispute no charge
##    781295    215424       536      2745
```

Incidentally, trips paid in cash are way more frequent than trips paid by credit card. Again using the `table.ff`-function, we investigate what factors might be correlated with payment types. First, we have a look at whether payment type is associated with the number of passengers in a trip.

```
# select the subset of observations only containing trips paid by credit card or cash
taxi_sub <- subset.ffdf(taxi, payment_type=="credit" | payment_type == "cash")
taxi_sub$payment_type <- ff(taxi_sub$payment_type,
                        levels = c("credit", "cash"),
```

```
                        ramclass = "factor")

# compute the cross tabulation
crosstab <- table.ff(taxi_sub$passenger_count,
                     taxi_sub$payment_type
                     )
# add names to the margins
names(dimnames(crosstab)) <- c("Passenger count", "Payment type")
# show result
crosstab
```

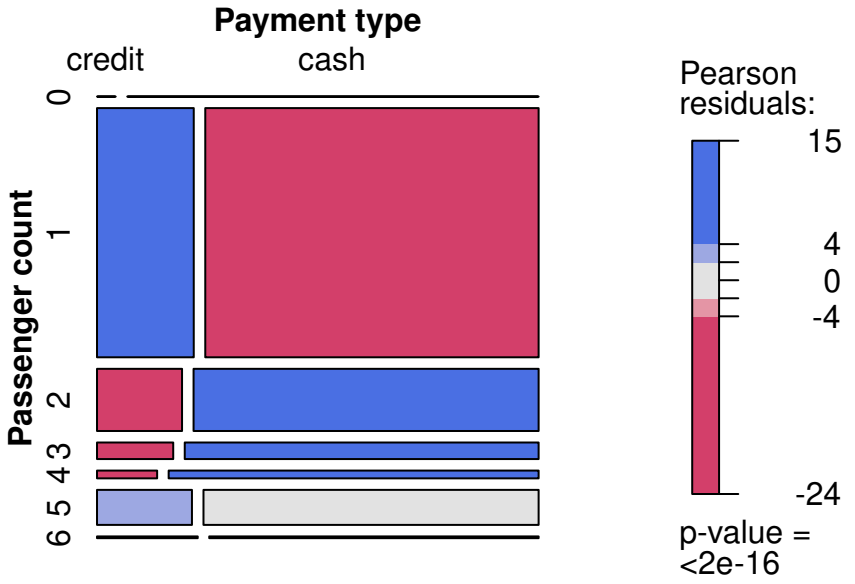
```
##                 Payment type
## Passenger count credit   cash
##               0      2     44
##               1 149990 516828
##               2  32891 133468
##               3   7847  36439
##               4   2909  17901
##               5  20688  73027
##               6   1097   3588
```

From the raw numbers it is hard to see whether there are significant differences between the categories cash and credit. We therefore use a visualization technique called 'mosaic plot' to visualize the cross-tabulation.

```
# install.packages(vcd)
# load package for mosaic plot
library(vcd)
```

```
## Loading required package: grid
```

```
# generate a mosaic plot
mosaic(crosstab, shade = TRUE)
```



The plot suggests that trips involving more than one passenger tend to be paid rather by cash than by credit card.

6.2.3 High-speed in-memory data aggregation with `data.table`

For large data sets that still fit into RAM, the `data.table`-package provides very fast and elegant functions to compute aggregate statistics.

6.2.3.1 Data import

We use the already familiar `fread()` to import the same first million observations from the January 2009 taxi trips records.

```
# load packages
library(data.table)

# import data into RAM (needs around 200MB)
taxi <- fread("data/tlc_trips.csv",
              nrows = 1000000)
```

6.2.3.2 Data preparation

We prepare/clean the data as in the `ff`-approach above.

```
# first, we remove the empty vars V8 and V9
taxi$V8 <- NULL
taxi$V9 <- NULL


# set covariate names according to the data dictionary
# see https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf
# note instead of taxizonne ids, long/lat are provided

varnames <- c("vendor_id",
              "pickup_time",
              "dropoff_time",
              "passenger_count",
              "trip_distance",
              "start_lat",
              "start_long",
              "dest_lat",
              "dest_long",
              "payment_type",
              "fare_amount",
              "extra",
              "mta_tax",
              "tip_amount",
              "tolls_amount",
              "total_amount")
names(taxi) <- varnames

# clean the factor levels
taxi$payment_type <- tolower(taxi$payment_type)
taxi$payment_type <- factor(taxi$payment_type, levels = unique(taxi$payment_type))
```

Note the simpler syntax of essentially doing the same thing, but all in-memory.

6.2.3.3 `data.table`-syntax for 'split-apply-combine' operations

With the `[]`-syntax we index/subset usual `data.frame` objects in R. When working with `data.table`s, much more can be done in the step of 'sub-setting' the frame.[4]

For example, we can directly compute on columns.

```
taxi[, mean(tip_amount/total_amount)]
```

```
## [1] 0.03452
```

Moreover, in the same step, we can 'split' the rows *by* specific groups and apply the function to each subset.

```
taxi[, .(percent_tip = mean((tip_amount/total_amount)*100)), by = payment_type]
```

```
##    payment_type percent_tip
## 1:         cash    0.005978
## 2:       credit   16.004173
## 3:    no charge    0.040433
## 4:      dispute    0.045660
```

Similarly, we can use `data.table`'s `dcast()` for cross-tabulation-like operations.

```
dcast(taxi[payment_type %in% c("credit", "cash")],
      passenger_count~payment_type,
      fun.aggregate = length,
      value.var = "vendor_id")
```

```
##    passenger_count   cash credit
## 1:               0     44      2
## 2:               1 516828 149990
## 3:               2 133468  32891
## 4:               3  36439   7847
```

[4] See https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html for a detailed introduction to the syntax.

```
## 5:              4  17901   2909
## 6:              5  73027  20688
## 7:              6   3588   1097
```

7

Data Storage and Databases

7.1 (Big) Data Storage

So far, we have primarily been concerned with situations in which a data set is too large to fit into RAM, making an analysis of these data either impossible or very slow (inefficient) when using standard tools. Thus, we have explored concepts and tools that help us use the available RAM (and virtual memory) most efficiently for data analysis tasks.

In this lecture, we are concerned with (I) how we can store large data sets permanently on a mass storage device in an efficient way (here, efficient can be understood as 'not taking up too much space') and (II) how we can load (parts of) this data set in an efficient way (here, efficient~fast) for analysis.

We look at this problem in two situations:

- The data needs to be stored locally (e.g., on the hard disk of our laptop).
- The data can be stored on a server 'in the cloud'.

Various tools have been developed over the last few years to improve the efficiency of storing and accessing large amounts of data (see Walkowiak (2016), chapters 5 and 6 for an overview). Here, we focus on the basic concept of Relational Database Systems (RDBMS) and a well-known tool based on this concept, the Structured Query Language (SQL; more specifically, SQLite)

7.2 RDBMS basics

RDBMSs have two key features that tackle the two efficiency concerns mentioned above:

- The *relational data model*: The overall data set is split by columns (covariates) into tables in order to reduce the storage of redundant variable-value repetitions. The resulting database tables are then linked via key-variables (unique identifiers). Thus (simply put), each type of entity on which observations exist resides in its own database table. Within this table, each observation has it's unique id. Keeping the data in such a structure is very efficient in terms of storage space used.
- *Indexing*: The key-columns of the database tables are indexed, meaning (in simple terms) ordered on disk. Indexing a table takes time but it has to be performed only once (unless the content of the table changes). The resulting index is then stored on disk as part of the database. These indices substantially reduce the number of disk accesses required to query/find specific observations. Thus, they make the loading of specific parts of the data for analysis much more efficient.

The loading/querying of data from an RDBMS typically involves the selection of specific observations (rows) and covariates (columns) from different tables. Due to the indexing, observations are selected efficiently, and the defined relations between tables (via keys) facilitate the joining of columns to a new table (the queried data).

7.3 Getting started with (R)SQLite

SQLite[1] is a free full-featured SQL database engine and widely used across platforms and is typically pre-installed with Windows and OSX

[1] https://sqlite.org/index.html

distributions. It is perfect to learn how to use RDBMSs/SQL. The R-package `RSQLite` embeds SQLite in R. That is, it provides functions that allow us to us SQLite directly from within R.

7.3.1 First steps in SQLite

In the terminal, we can directly call SQLite as a command-line tool (on most modern computers this is now `sqlite3`). In this short example, we set up our first SQLite database, using the command line. In the file structure of the course repository, we first switch to the data directory.

```
cd materials/data
```

With one simple command, we both create a new SQLite database called `mydb.sqlite` and connect/run SQLite.

```
sqlite3 mydb.sqlite
```

This created a new file `mydb.sqlite` in our `data` directory. And we are now running `sqlite` in our terminal. But, the database is still empty. There are no tables in it. We can check the tables of a database with the following SQL command `.tables`.

```
.tables
```

As expected, nothing is returned. Now, let's create our first table and import the `economics.csv` data set to it. In SQLite, it makes sense to first set up an empty table in which all column data types are defined before importing data from a CSV-file to it. If a CSV is directly imported to a new table (without type definitions), all columns will be set to `TEXT` by default.

```
-- Create the new table
CREATE TABLE econ(
"date" DATE,
"pce" REAL,
```

```
"pop" INTEGER,
"psavert" REAL,
"uempmed" REAL,
"unemploy" INTEGER
);

-- prepare import
.mode csv
-- import data from csv
.import economics.csv econ
```

Now we can have a look at the new database table in SQLite. `.tables` shows that we now hove one table called `econ` in our database and `.schema` displays the structure of the new `econ` table.

```
.tables
```

```
# econ
```

```
.schema econ
```

```
# CREATE TABLE econ(
# "date" DATE,
# "pce" REAL,
# "pop" INTEGER,
# "psavert" REAL,
# "uempmed" REAL,
# "unemploy" INTEGER
# );
```

With this, we can actually start querying data with SQLite. In order to make the query results easier to read, we first set two options regarding how query results are displayed in the terminal. `.header on` enables the display of the column names in the returned query results. And `.mode columns` arranges the query results in columns.

```
.header on
```

TABLE 7.1: 1 records

date	pce	pop	psavert	uempmed	unemploy
1968-01-01	531.5	199808	11.7	5.1	2878

TABLE 7.2: Displaying records 1 - 10

date
2009-09-01
2009-10-01
2009-11-01
2009-12-01
2010-01-01
2010-02-01
2010-03-01
2010-04-01
2010-11-01
date

```
.mode columns
```

In our first query, we select all (*) variable values of the observation of January 1968.

```
select * from econ where date = '1968-01-01';
```

Now let's select all year/months in which there were more than 15 million unemployed, ordered by date.

```
select date from econ
where unemploy > 15000
order by date;
```

When done working with the database, we can exit SQLite with the `.quit` command.

7.3.2 Indices and joins

So far we have only had a look at the basic functionality of SQLite. But we have not really looked at the key features of an RDBMS (indexing and relations between tables).

We set up a new database called `air.sqlite` and import the csv-file `flights.csv` (used in previous lectures) as a first table.

```
# create database and run sqlite
sqlite3 air.sqlite
```

```
-- import csvs
.mode csv
.import flights.csv flights
```

Again, we can check if everything worked out well with `.tables` and `.schema`.

```
.tables
.schema flights
```

In `flights`, each row describes a flight (the day it took place, its origin, its destination etc.). It contains a covariate `carrier` containing the unique ID of the respective airline/carrier carrying out the flight as well as the covariates `origin` and `dest`. The latter two variables contain the unique IATA-codes of the airports from which the flights departed and where they arrived, respectively. In `flights` we thus have observations at the level of individual flights.

Now we extend our database in a meaningful way, following the relational data model idea. From the [ASA's website], we download two additional csv files containing data that relate to the flights table:

- `airports.csv`[2]: Describes the locations of US Airports (relates to `origin` and `dest`).

[2] http://stat-computing.org/dataexpo/2009/airports.csv

- `carriers.csv`[3]: A listing of carrier codes with full names (relates to the `carrier`-column in `flights`.

In this code example, the two csvs have already been downloaded to the `materials/data`-folder.

```
-- import airport data
.mode csv
.import airports.csv airports
.import carriers.csv carriers

-- inspect the result
.tables
.schema airports
.schema carriers
```

Now we can run our first query involving the relation between tables. The aim of the exercise is to query flights data (information on departure delays per flight number and date; from the `flights`-table) for all `United Air Lines Inc.`-flights (information from the `carriers` table) departing from `Newark Intl` airport (information from the `airports`-table). In addition, we want the resulting table ordered by flight number. For the sake of the exercise, we only show the first 10 results of this query (`LIMIT 10`).

```
SELECT
year,
month,
day,
dep_delay,
flight
FROM (flights INNER JOIN airports ON flights.origin=airports.iata)
INNER JOIN carriers ON flights.carrier = carriers.Code
WHERE carriers.Description = 'United Air Lines Inc.'
AND airports.airport = 'Newark Intl'
```

[3] http://stat-computing.org/dataexpo/2009/carriers.csv

TABLE 7.3: Displaying records 1 - 10

year	month	day	dep_delay	flight
2013	1	4	0	1
2013	1	5	-2	1
2013	3	6	1	1
2013	2	13	-2	3
2013	2	16	-9	3
2013	2	20	3	3
2013	2	23	-5	3
2013	2	26	24	3
2013	2	27	10	3
2013	1	5	3	10

```
ORDER BY flight
LIMIT 10;
```

Note that this query has been executed without indexing any of the tables first. Thus SQLite could not take any 'shortcuts' when matching the ID columns in order to join the tables for the query output. That is, SQLite had to scan the entire columns to find the matches. Now we index the respective id columns and re-run the query

```
CREATE INDEX iata_airports ON airports (iata);
CREATE INDEX origin_flights ON flights (origin);
CREATE INDEX carrier_flights ON flights (carrier);
CREATE INDEX code_carriers ON carriers (code);
```

Now we can re-run the query from above. Note that SQLite optimizes the efficiency of the query without our explicit instructions. If there are indices it can use to speed up the query, it will do so.

```
SELECT
year,
month,
```

TABLE 7.4: Displaying records 1 - 10

year	month	day	dep_delay	flight
2013	1	4	0	1
2013	1	5	-2	1
2013	3	6	1	1
2013	2	13	-2	3
2013	2	16	-9	3
2013	2	20	3	3
2013	2	23	-5	3
2013	2	26	24	3
2013	2	27	10	3
2013	1	5	3	10

```
day,
dep_delay,
flight
FROM (flights INNER JOIN airports ON flights.origin=airports.iata)
INNER JOIN carriers ON flights.carrier = carriers.Code
WHERE carriers.Description = 'United Air Lines Inc.'
AND airports.airport = 'Newark Intl'
ORDER BY flight
LIMIT 10;
```

You find the final `air.sqlite`, including all the indices and tables as `materials/data/air_final.sqlite` in the course's code repository.

7.4 SQLite from within R

The `RSQLite`-package provides various R functions to control basically all of SQLite's functionalities from within R. In the following example, we explore how `RSQLite` can be used to set up and query the `air.sqlite` shown in the example above.

7.4.1 Creating a new database with `RSQLite`

Similarly to the raw SQLite-syntax, connecting to a database that does not exist yet, actually creates this (empty database). Note that for all interactions with the database from within R, we need to refer to the connection (here: `con_air`).

```
# load packages
library(RSQLite)

# initiate the database
con_air <- dbConnect(SQLite(), "data/air.sqlite")
```

7.4.2 Importing data

With `RSQLite` we can easily add `data.frame`s as SQLite tables to the database.

```
# import data into current R sesssion
flights <- fread("data/flights.csv")
airports <- fread("data/airports.csv")
carriers <- fread("data/carriers.csv")

# add tables to database
dbWriteTable(con_air, "flights", flights)
dbWriteTable(con_air, "airports", airports)
dbWriteTable(con_air, "carriers", carriers)
```

7.4.3 Issue queries

Now we can query the database from within R. By default, `RSQLite` returns the query results as `data.frame`s. Queries are simply character strings written in SQLite.

```
# define query
delay_query <-
```

```
"SELECT
year,
month,
day,
dep_delay,
flight
FROM (flights INNER JOIN airports ON flights.origin=airports.iata)
INNER JOIN carriers ON flights.carrier = carriers.Code
WHERE carriers.Description = 'United Air Lines Inc.'
AND airports.airport = 'Newark Intl'
ORDER BY flight
LIMIT 10;
"

# issue query
delays_df <- dbGetQuery(con_air, delay_query)
delays_df
```

```
##    year month day dep_delay flight
## 1  2013     1   4         0      1
## 2  2013     1   5        -2      1
## 3  2013     3   6         1      1
## 4  2013     2  13        -2      3
## 5  2013     2  16        -9      3
## 6  2013     2  20         3      3
## 7  2013     2  23        -5      3
## 8  2013     2  26        24      3
## 9  2013     2  27        10      3
## 10 2013     1   5         3     10
```

8

(Big) Data Visualization

8.1 Data Set

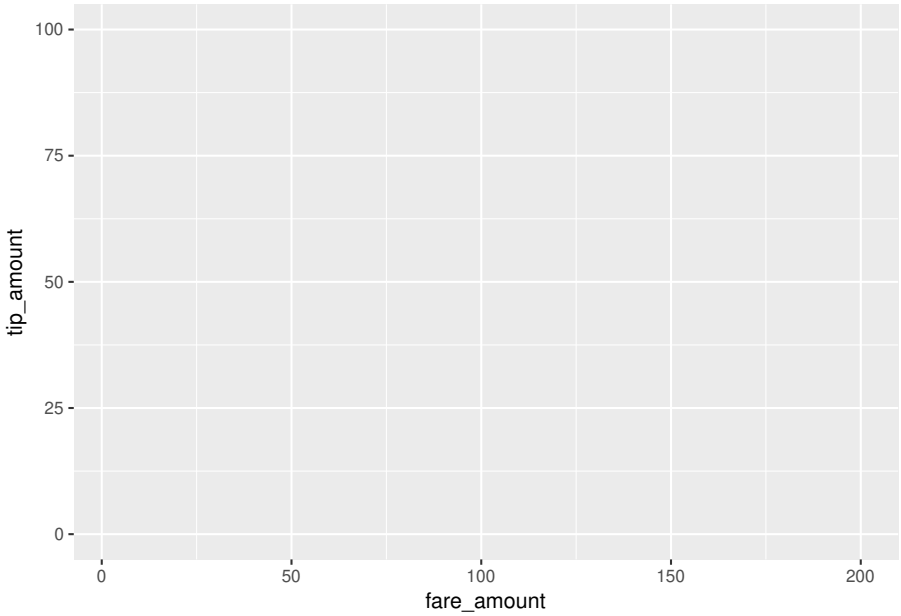
In this tutorial we will work with the TLC data used in the data aggregation session. The raw data consists of several monthly CSV-files and can be downloaded via the TLC's website[1]. Again, we work only with the first million observations.

In order to better understand the large data set at hand (particularly regarding the determinants of tips paid) we use `ggplot2` to visualize some key aspects of the data.

First, let's look at the raw relationship between fare paid and the tip paid. We set up the canvas with `ggplot`.

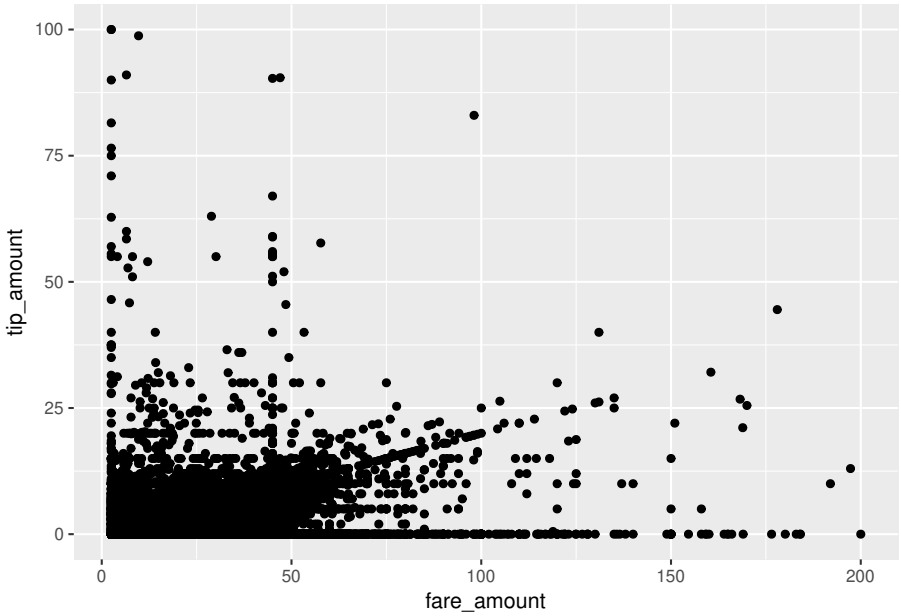
```
# load packages
library(ggplot2)

# set up the canvas
taxiplot <- ggplot(taxi, aes(y=tip_amount, x= fare_amount))
taxiplot
```

[1] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page



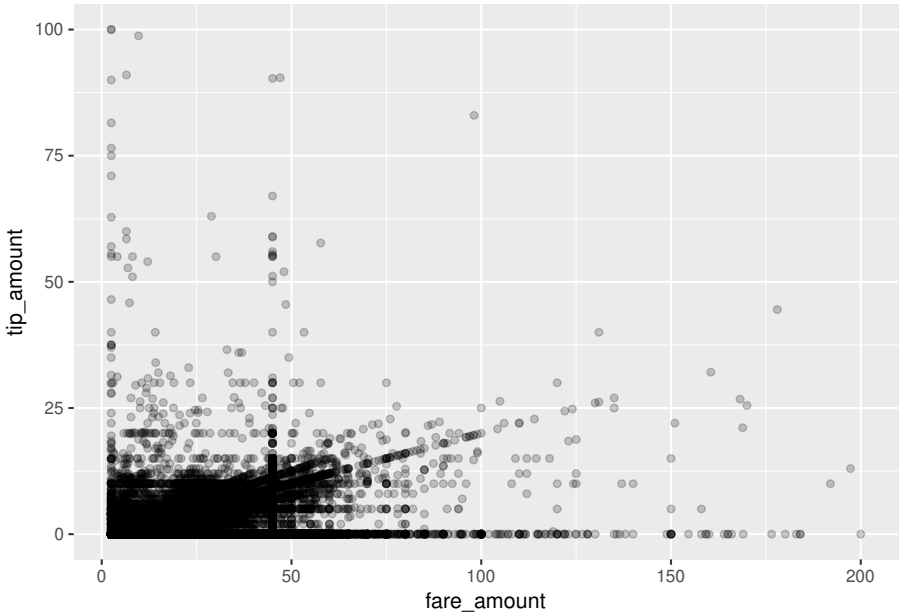
Now we visualize the co-distribution of the two variables with a simple scatter-plot.

```
# simple x/y plot
taxiplot +
     geom_point()
```



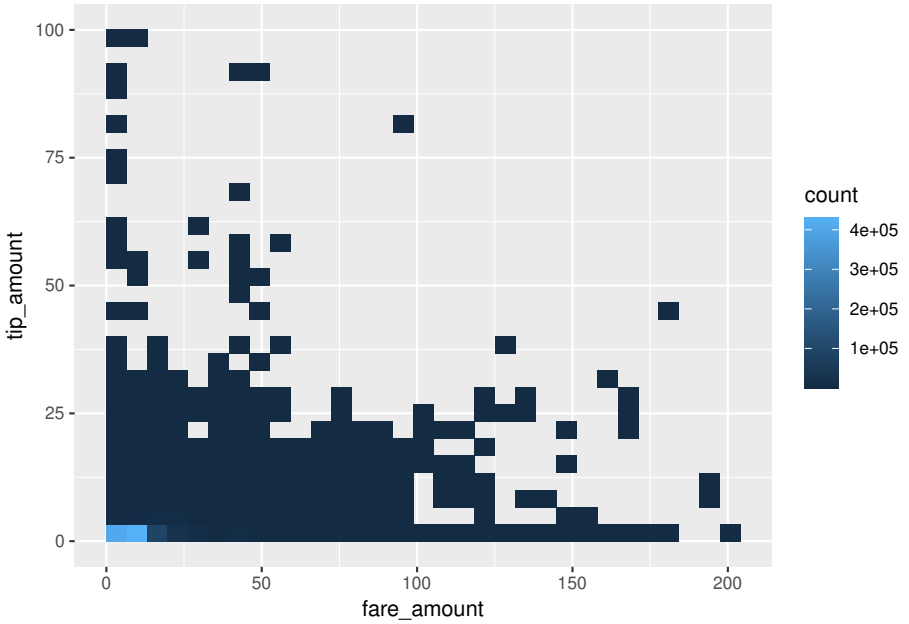
Note that this took quite a while, as R had to literally plot one million dots on the canvas. Moreover many dots fall within the same area, making it impossible to recognize how much mass there actually is. This is typical for visualization exercises with large data sets. One way to improve this, is by making the dots more transparent by setting the `alpha` parameter.

```
# simple x/y plot
taxiplot +
     geom_point(alpha=0.2)
```



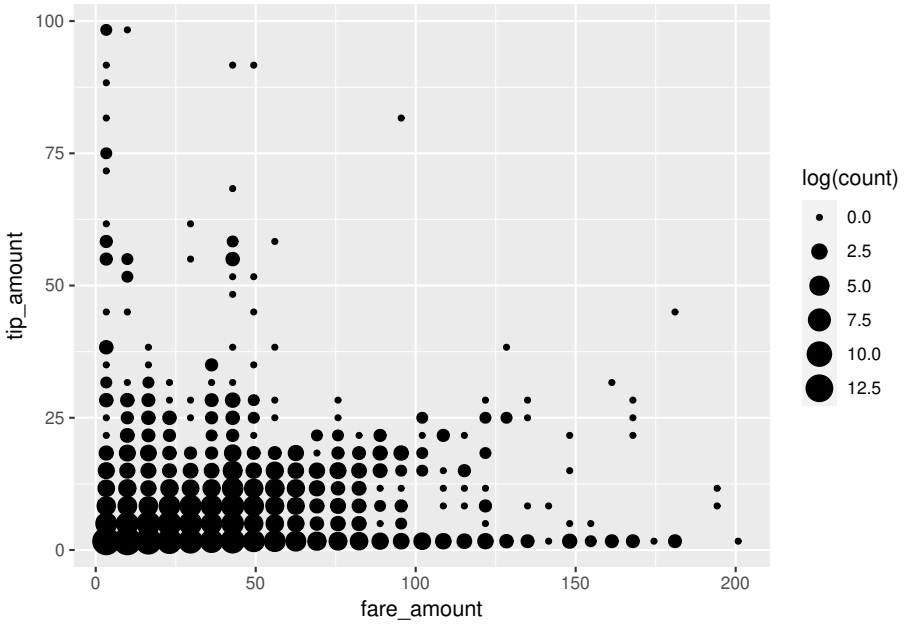
Alternatively, we can compute two-dimensional bins. Thereby, the canvas is split into rectangles and the number of observations falling into each respective rectangle is computed. The visualization is based on plotting the rectangles with counts greater than 0 and the shading of the rectangles indicates the count values.

```
# 2-dimensional bins
taxiplot +
     geom_bin2d()
```



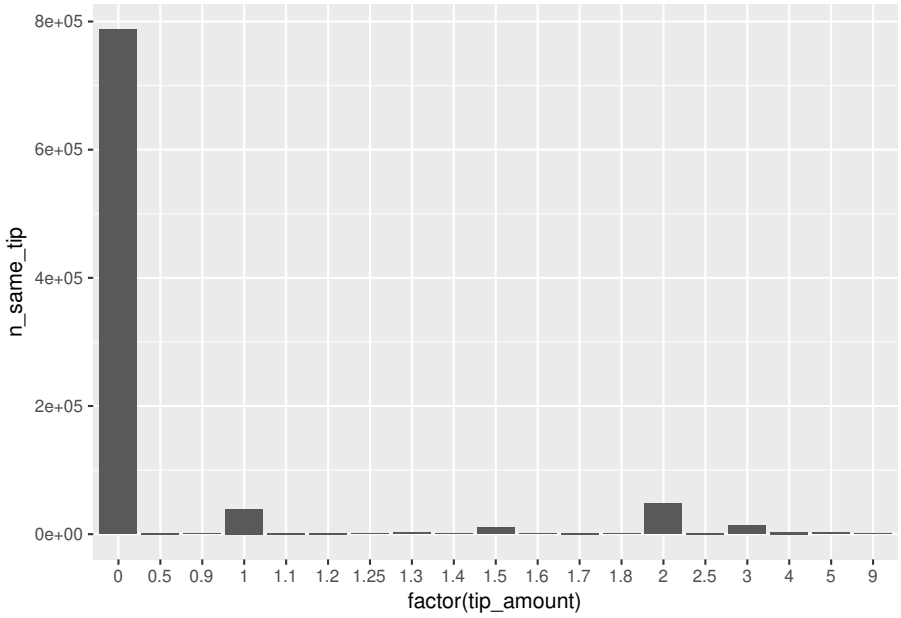
A large part of the tip/fare observations seem to be in the very lower-left corner of the pane, while most other trips seem to be evenly distributed. However, we fail to see slighter differences in this visualization. In order to reduce the dominance of the 2d-bins with very high counts, we display the natural logarithm of counts and display the bins as points.

```
# 2-dimensional bins
taxiplot +
     stat_bin_2d(geom="point",
                 mapping= aes(size = log(..count..))) +
     guides(fill = FALSE)
```



We note that there are many cases with very low fare amounts, many cases with no or hardly any tip, and quite a lot of cases with very high tip amounts (in relation to the rather low fare amount). In the following, we dissect this picture by having a closer look at 'typical' tip amounts and whether they differ by type of payment.

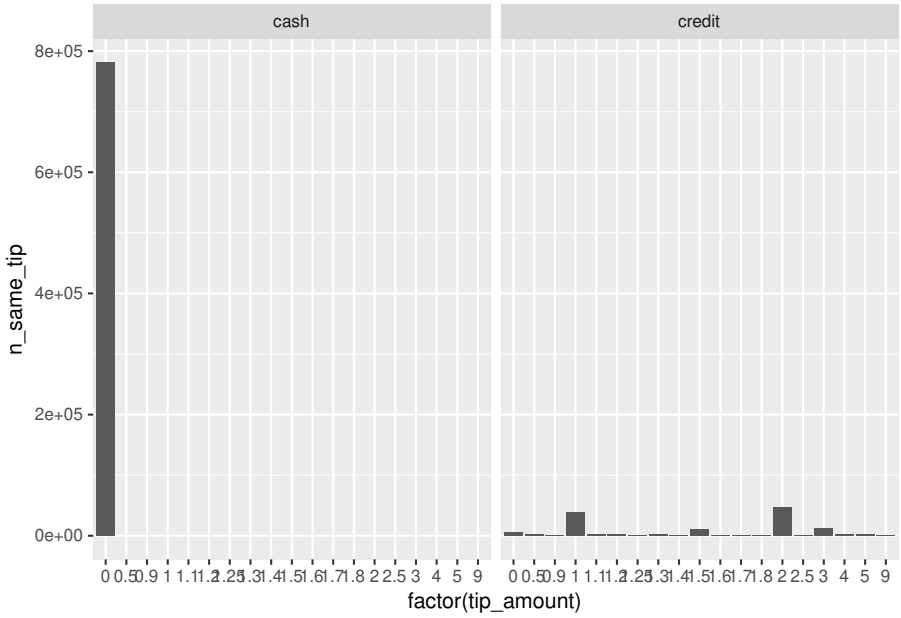
```
# compute frequency of per tip amount and payment method
taxi[, n_same_tip:= .N, by= c("tip_amount", "payment_type")]
frequencies <- unique(taxi[payment_type %in% c("credit", "cash"),
                           c("n_same_tip", "tip_amount", "payment_type")][order(n_same_tip, de


# plot top 20 frequent tip amounts
fare <- ggplot(data = frequencies[1:20], aes(x = factor(tip_amount), y = n_same_tip))
fare + geom_bar(stat = "identity")
```



Indeed, paying no tip at all is quite frequent, overall.[2] The bar plot also indicates that there seem to be some 'focal points' in the amount of tips paid. Clearly, paying one USD or two USD is more common than paying fractions. However, fractions of dollars might be more likely if tips are paid in cash and customers simply add some loose change to the fare amount paid.

```
fare + geom_bar(stat = "identity") +
     facet_wrap("payment_type")
```

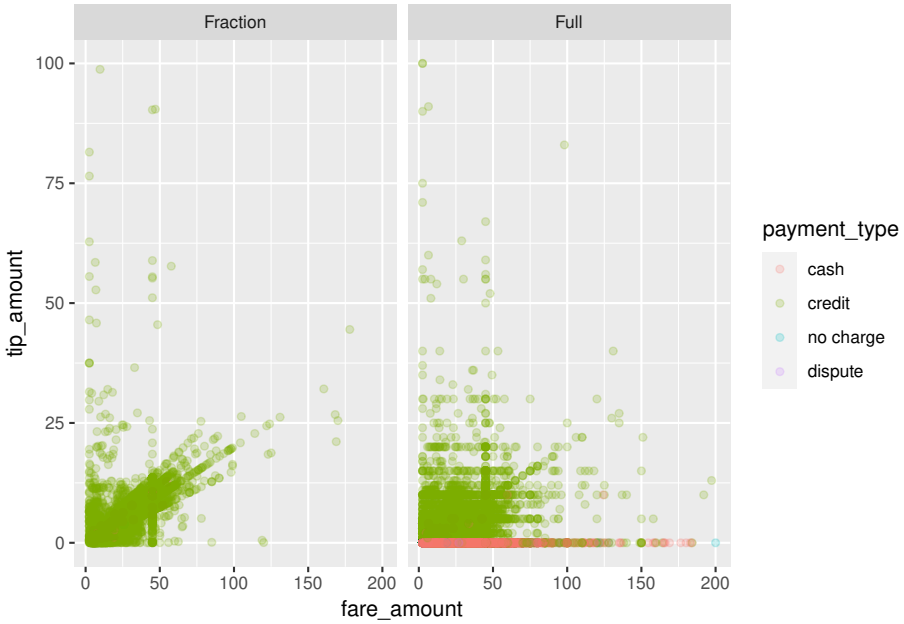
[2] Or, could there be another explanation for this pattern in the data?



Clearly, it looks like trips paid in cash tend not to be tipped (at least in this subsample).

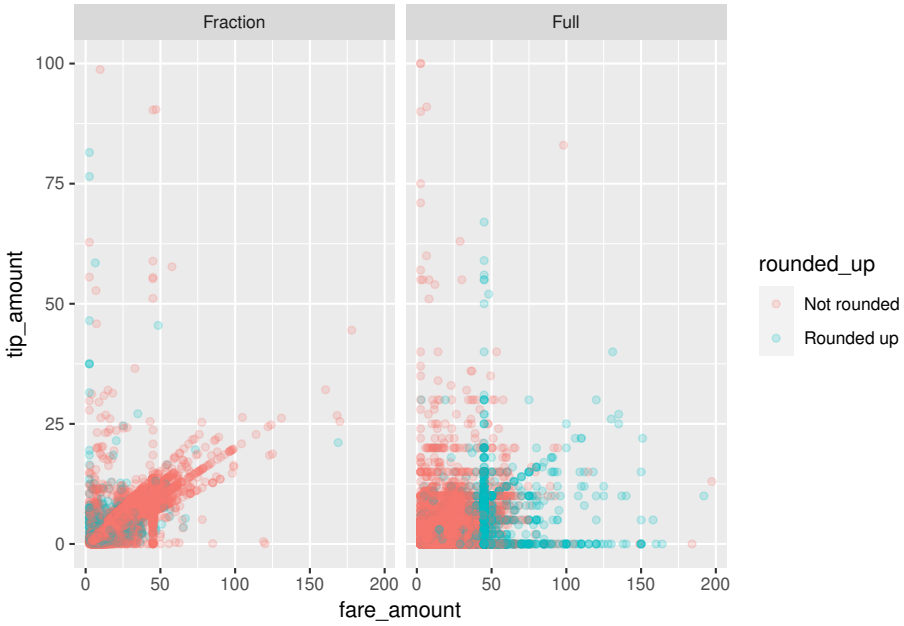
Let's try to tease this information out of the initial points plot. Trips paid in cash are often not tipped, we thus should indicate the payment method. Moreover, tips paid in full dollar amounts might indicate a habit.

```
# indicate natural numbers
taxi[, dollar_paid := ifelse(tip_amount == round(tip_amount,0), "Full", "Fraction"),]


# extended x/y plot
taxiplot +
     geom_point(alpha=0.2, aes(color=payment_type)) +
     facet_wrap("dollar_paid")
```



Now the picture is getting clearer. Paying tip seems to follow certain rules of thumb. Certain fixed amounts tend to be paid independent of the fare amount (visible in the straight lines of dots on the right-hand panel). At the same time, the pattern in the left panel indicates another habit: computing the amount of tip as a linear function of the total fare amount ('pay 10% tip'). A third habit might be to determine the amount of tip by 'rounding up' the total amount paid. In the following, we try to tease the latter out, only focusing on credit card payments.

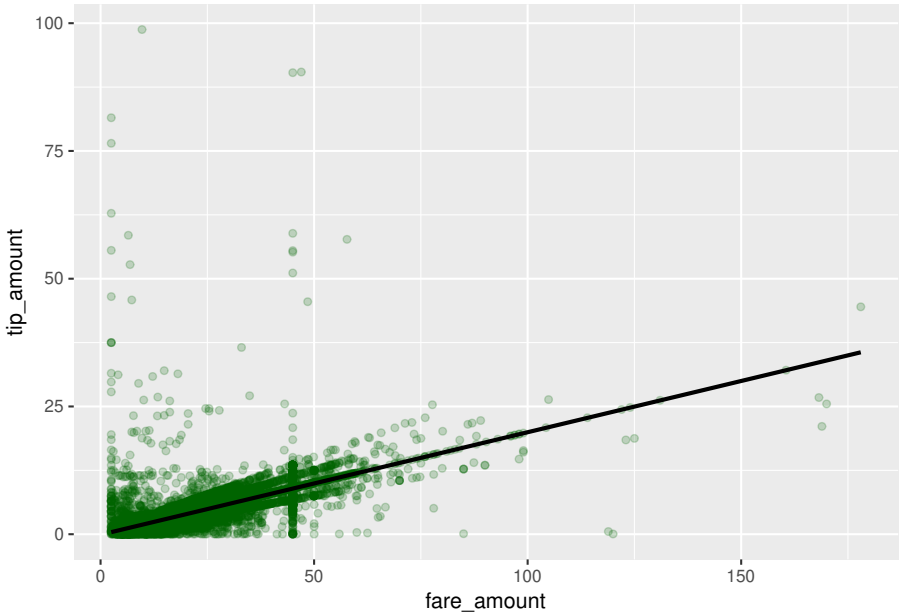
```
taxi[, rounded_up := ifelse(fare_amount + tip_amount == round(fare_amount + tip_amount, 0),
                            "Rounded up",
                            "Not rounded")]
# extended x/y plot
taxiplot +
     geom_point(data= taxi[payment_type == "credit"],
                alpha=0.2, aes(color=rounded_up)) +
     facet_wrap("dollar_paid")
```



Now we can start modelling. A reasonable first shot is to model the tip amount as a linear function of the fare amount, conditional on the no-zero tip amounts paid as fractions of a dollar.

```
modelplot <- ggplot(data= taxi[payment_type == "credit" & dollar_paid == "Fraction" & 0 < tip_
                    aes(x = fare_amount, y = tip_amount))
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Finally, we prepare the plot for reporting. `ggplot2` provides several pre-defined 'themes' for plots which define all kind of aspects of a plot (background color, line colors, font size etc.). The easiest way to tweak the design of your final plot in a certain direction is to just add such a pre-defined theme at the end of your plot. Some of the pre-defined themes allow you to change a few aspects such as the font type and the base size of all the texts in the plot (labels and tick numbers etc.). Here, we use the `theme_bw()`, increase the font size and switch to serif-type font. `theme_bw()` is one of the complete themes already shipped with the basic `ggplot2` installation.[3] Many more themes can be found in additional R packages (see, for example, the `ggthemes` package[5]).

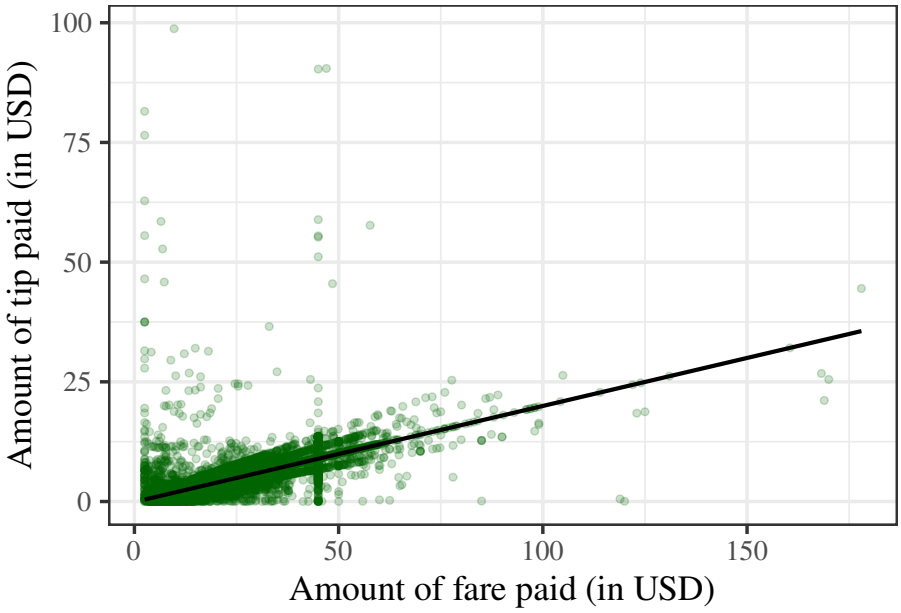
```
modelplot <- ggplot(data= taxi[payment_type == "credit" & dollar_paid == "Fraction" & 0 < tip_
                    aes(x = fare_amount, y = tip_amount))
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black") +
```

[3] See the ggplot2 documentation[4] for a list of all pre-defined themes shipped with the basic installation.

[5] https://cran.r-project.org/web/packages/ggthemes/index.html

```
     ylab("Amount of tip paid (in USD)") +
     xlab("Amount of fare paid (in USD)") +
     theme_bw(base_size = 18, base_family = "serif")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



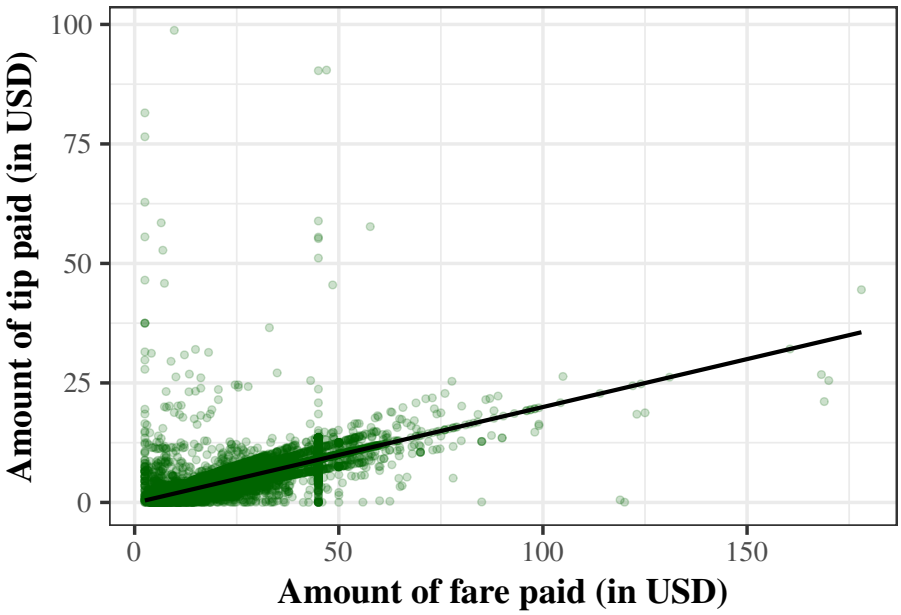
8.2 Excursus: modify and create themes

Apart from using pre-defined themes as illustrated above, we can use the `theme()` function to further modify the design of a plot. For example, we can print the axis labels ('axis titles') in bold.

```
modelplot <- ggplot(data= taxi[payment_type == "credit" & dollar_paid == "Fraction" & 0 < tip_
                    aes(x = fare_amount, y = tip_amount))
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black") +
```

```
    ylab("Amount of tip paid (in USD)") +
    xlab("Amount of fare paid (in USD)") +
    theme_bw(base_size = 18, base_family = "serif") +
    theme(axis.title = element_text(face="bold"))
```

```
## `geom_smooth()` using formula 'y ~ x'
```



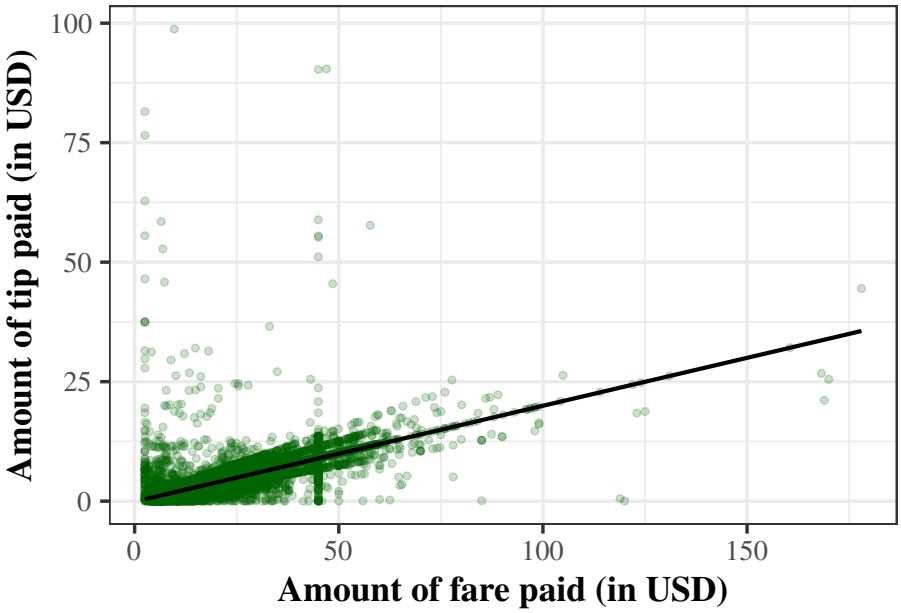
There is a large list of plot design aspects that can be modified in this way (see `?theme()` for details).

8.2.1 Create your own theme: simple approach

Extensive design modifications via `theme()` can involve many lines of code, making your plot code harder to read/understand. In practice, you might want to define your specific theme once and then apply this theme to all of your plots. In order to do so it makes sense to choose one of the existing themes as a basis and then modify its design aspects until you have the design you are looking for. Following the design choices in the examples above, we can create our own `theme_my_serif()` as follows.

```
# 'define' a new theme
theme_my_serif <-
  theme_bw(base_size = 18, base_family = "serif") +
  theme(axis.title = element_text(face="bold"))

# apply it
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black") +
     ylab("Amount of tip paid (in USD)") +
     xlab("Amount of fare paid (in USD)") +
  theme_my_serif
```

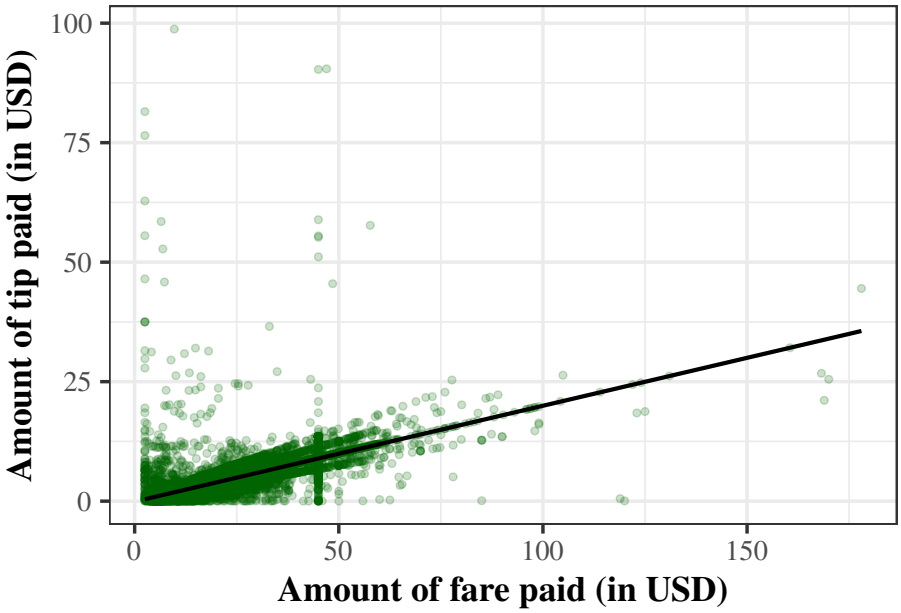
```
## `geom_smooth()` using formula 'y ~ x'
```



This practical approach does not require you to define every aspect of a theme. If you indeed want completely define every aspect of a theme, you can set `complete=TRUE` when calling the theme function.

```
# 'define' a new theme
my_serif_theme <-
  theme_bw(base_size = 18, base_family = "serif") +
  theme(axis.title = element_text(face="bold"), complete = TRUE)

# apply it
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black") +
     ylab("Amount of tip paid (in USD)") +
     xlab("Amount of fare paid (in USD)") +
  theme_my_serif
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Note that since we have only defined one aspect (bold axis titles), the rest of the elements follow the default theme.

Importantly, the approach outlined above does technically not really create a new theme like `theme_bw()`, as these pre-defined themes are implemented as functions. Note that we add the new theme simply

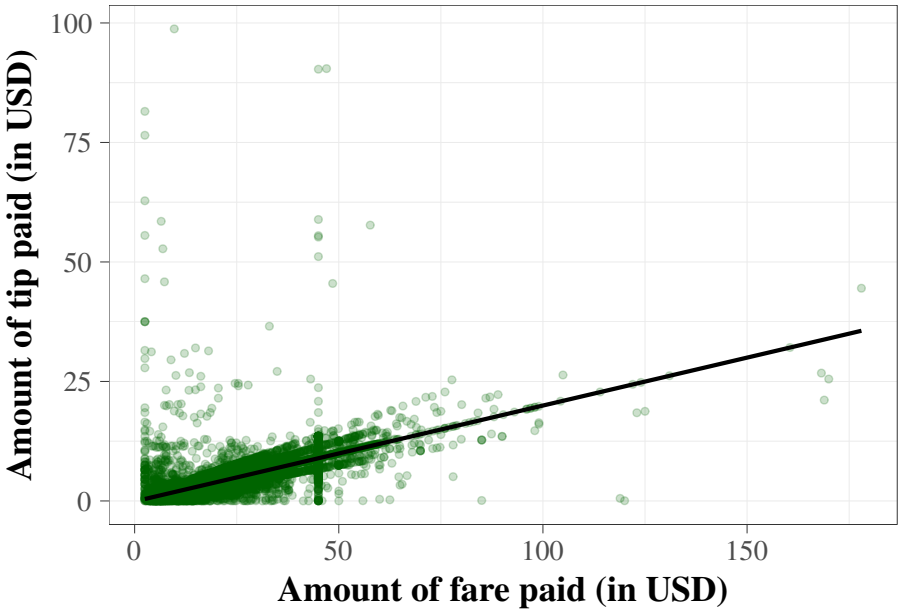
with `+ theme_my_serif` to the plot (no parentheses). In practice this is the most simple approach and it provides all the functionality you need in order to apply your own 'theme' to each of your plots.

8.2.2 Implementing actual themes as functions.

If you really want to implement a theme as a function. The following blueprint can get you started.

```
# define own theme
theme_my_serif <-
  function(base_size = 15,
           base_family = "",
           base_line_size = base_size/170,
           base_rect_size = base_size/170){

    theme_bw(base_size = base_size,
             base_family = base_family,
             base_line_size = base_size/170,
             base_rect_size = base_size/170) %+replace%    # use theme_bw() as a basis but rep
    theme(
      axis.title = element_text(face="bold")
    )
  }

# apply the theme
# apply it
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black") +
     ylab("Amount of tip paid (in USD)") +
     xlab("Amount of fare paid (in USD)") +
  theme_my_serif(base_size = 18, base_family="serif")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



8.3 Visualize Time and Space

The previous visualization exercises were focused on visually exploring patterns in the tipping behavior of people taking a NYC yellow cap ride. Based on the same data set, will explore the time dimension and spatial dimension of the TLC Yellow Cap data. That is, we explore where trips tend to start and end, depending on the time of the day.

8.3.1 Preparations

For the visualization of spatial data we first load additional packages that give R some GIS[6] features.

```
# load GIS packages
library(rgdal)
library(rgeos)
```

[6] https://en.wikipedia.org/wiki/Geographic_information_system

Moreover, we download and import a so-called 'shape file'[7] (a geospatial data format) of New York City. This will be the basis for our visualization of the spatial dimension of taxi trips. The file is downloaded from New York's Department of City Planning[8] and indicates the city's community district borders.[9]

```
# download the zipped shapefile to a temporary file, unzip
URL <- "https://www1.nyc.gov/assets/planning/download/zip/data-maps/open-data/nycd_19a.zip"
tmp_file <- tempfile()
download.file(URL, tmp_file)
file_path <- unzip(tmp_file, exdir= "data")
# delete the temporary file
unlink(tmp_file)
```

Now we can import the shape file and have a look at how the GIS data is structured.

```
# read GIS data
nyc_map <- readOGR(file_path[1], verbose = FALSE)

# have a look at the GIS data
summary(nyc_map)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##      min     max
## x 913175 1067383
## y 120122  272844
## Is projected: TRUE
## proj4string :
## [+proj=lcc +lat_0=40.1666666666667 +lon_0=-74
## +lat_1=41.0333333333333 +lat_2=40.6666666666667
## +x_0=300000 +y_0=0 +datum=NAD83 +units=us-ft
```

[7] https://en.wikipedia.org/wiki/Shapefile

[8] https://www1.nyc.gov/site/planning/index.page

[9] Similar files are provided online by most city authorities in developed countries. See, for example, GIS Data for the City and Canton of Zurich: https://maps.zh.ch/.

```
## +no_defs]
## Data attributes:
##      BoroCD      Shape_Leng       Shape_Area
##  Min.   :101   Min.   : 23963   Min.   :2.43e+07
##  1st Qu.:206   1st Qu.: 36611   1st Qu.:4.84e+07
##  Median :308   Median : 52246   Median :8.27e+07
##  Mean   :297   Mean   : 74890   Mean   :1.19e+08
##  3rd Qu.:406   3rd Qu.: 85711   3rd Qu.:1.37e+08
##  Max.   :595   Max.   :270660   Max.   :5.99e+08
```

Note that the coordinates are not in the usual longitude and latitude units. The original map uses a different projection than the TLC data of cap trips records. Before plotting, we thus have to change the projection to be in line with the TLC data.

```
# transform the projection
nyc_map <- spTransform(nyc_map, CRS("+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84
# check result
summary(nyc_map)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##      min    max
## x -74.26 -73.70
## y  40.50  40.92
## Is projected: FALSE
## proj4string : [+proj=longlat +datum=WGS84 +no_defs]
## Data attributes:
##      BoroCD      Shape_Leng       Shape_Area
##  Min.   :101   Min.   : 23963   Min.   :2.43e+07
##  1st Qu.:206   1st Qu.: 36611   1st Qu.:4.84e+07
##  Median :308   Median : 52246   Median :8.27e+07
##  Mean   :297   Mean   : 74890   Mean   :1.19e+08
##  3rd Qu.:406   3rd Qu.: 85711   3rd Qu.:1.37e+08
##  Max.   :595   Max.   :270660   Max.   :5.99e+08
```

One last preparatory step is to convert the map data to a `data.frame` for plotting with `ggplot`.

```
nyc_map <- fortify(nyc_map)
```

8.3.2 Pick-up and drop-off locations

Since trips might actually start or end outside of NYC, we first restrict the sample of trips to those within the boundary box of the map. For the sake of the exercise, we only select a random sample of `50000` trips from the remaining trip records.

```
# taxi trips plot data
taxi_trips <- taxi[start_long <= max(nyc_map$long) &
                       start_long >= min(nyc_map$long) &
                       dest_long <= max(nyc_map$long) &
                       dest_long >= min(nyc_map$long) &
                       start_lat <= max(nyc_map$lat) &
                       start_lat >= min(nyc_map$lat) &
                       dest_lat <= max(nyc_map$lat) &
                       dest_lat >= min(nyc_map$lat)
                       ]
taxi_trips <- taxi_trips[sample(nrow(taxi_trips), 50000)]
```

In order to visualize how the cap traffic is changing over the course of the day, we add an additional variable called `start_time` in which we store the time (hour) of the day a trip started.

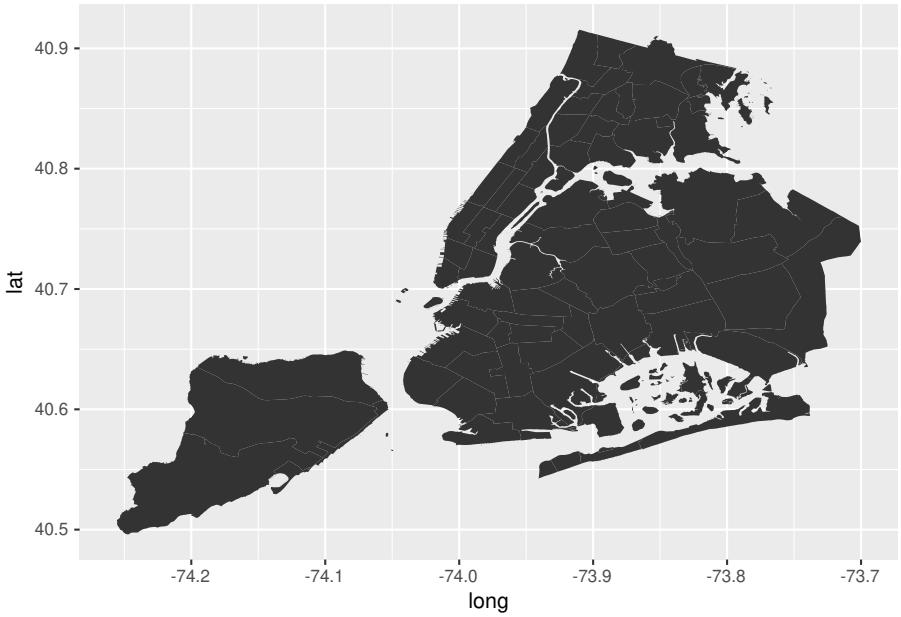
```
taxi_trips$start_time <- hour(taxi_trips$pickup_time)
```

Particularly, we want to look at differences between, morning, afternoon, and evening/night.

```
# define new variable for facets
taxi_trips$time_of_day <- "Morning"
taxi_trips[start_time > 12 & start_time < 17]$time_of_day <- "Afternoon"
taxi_trips[start_time %in% c(17:24, 0:5)]$time_of_day <- "Evening/Night"
taxi_trips$time_of_day  <- factor(taxi_trips$time_of_day, levels = c("Morning", "Afternoon", '
```

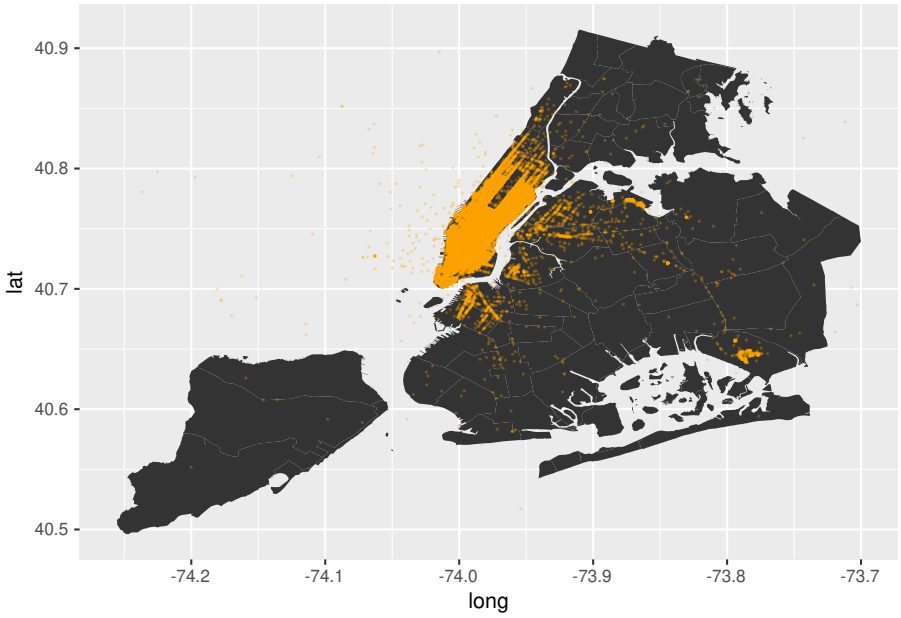
We initiate the plot by first setting up the canvas with our taxi trips data. Then, we add the map as a first layer.

```
# set up the canvas
locations <- ggplot(taxi_trips, aes(x=long, y=lat))
# add the map geometry
locations <- locations + geom_map(data = nyc_map,
                                  map = nyc_map,
                                  aes(map_id = id))
locations
```



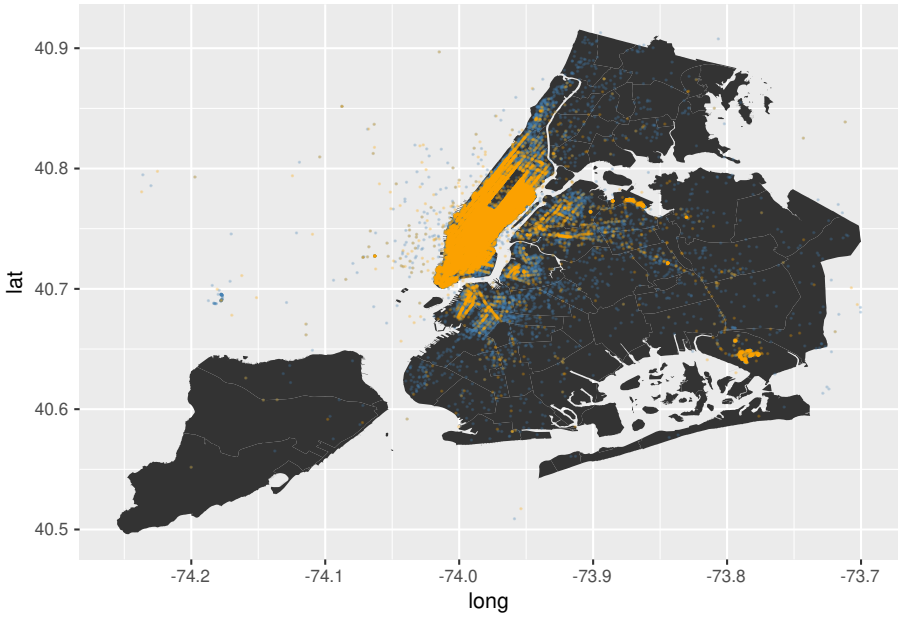
Now we can start adding the pick-up and drop-off locations of cap trips.

```
# add pick-up locations to plot
locations +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
                alpha = 0.2)
```



As to be expected, most of the trips start in Manhattan. Now let's look at where trips end.

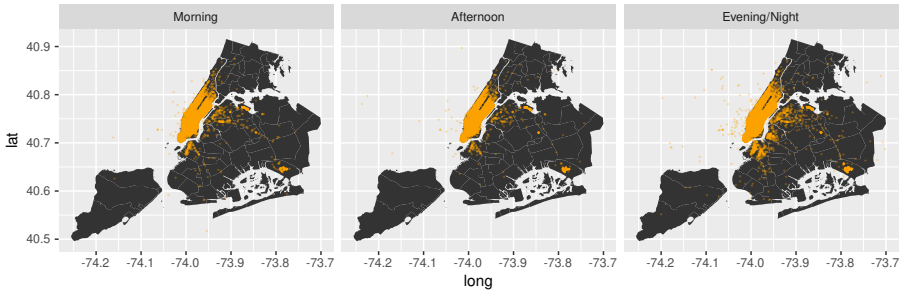
```
# add pick-up locations to plot
locations +
     geom_point(aes(x=dest_long, y=dest_lat),
                color="steelblue",
                size = 0.1,
                alpha = 0.2) +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
                alpha = 0.2)
```



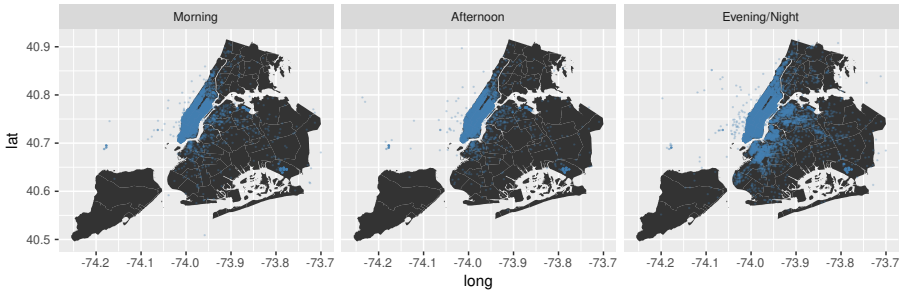
Incidentally, more trips tend to end outside of Manhattan. And the destinations seem to be broader spread across the city then the pick-up locations. Most destinations are still in Manhattan, though.

Now let's have a look at how this picture changes depending on the time of the day.

```
# pick-up locations
locations +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
                alpha = 0.2) +
     facet_wrap(vars(time_of_day))
```

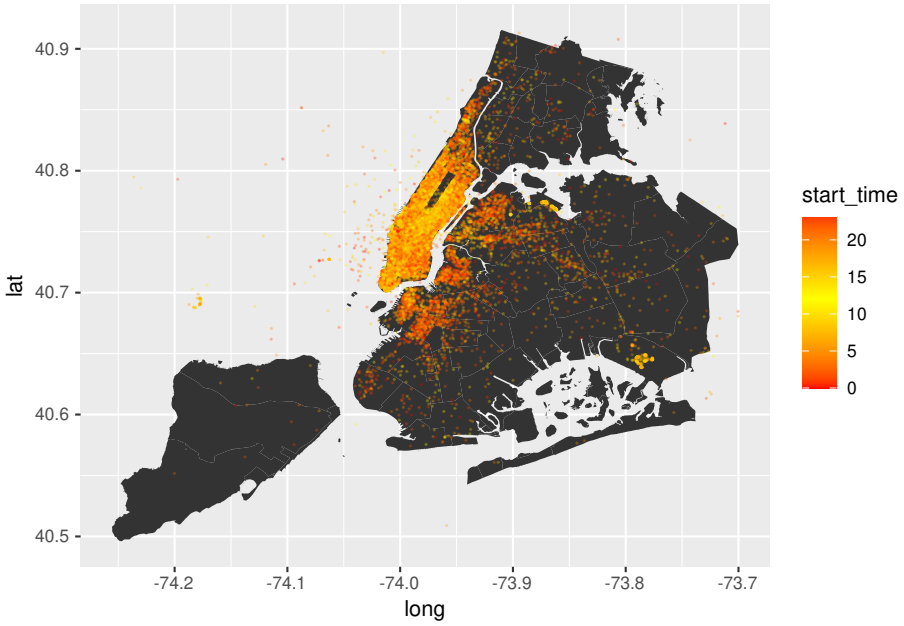



```
# drop-off locations
locations +
     geom_point(aes(x=dest_long, y=dest_lat),
                color="steelblue",
                size = 0.1,
                alpha = 0.2) +
     facet_wrap(vars(time_of_day))
```



Alternatively, we can plot the hours on a continuous scale.

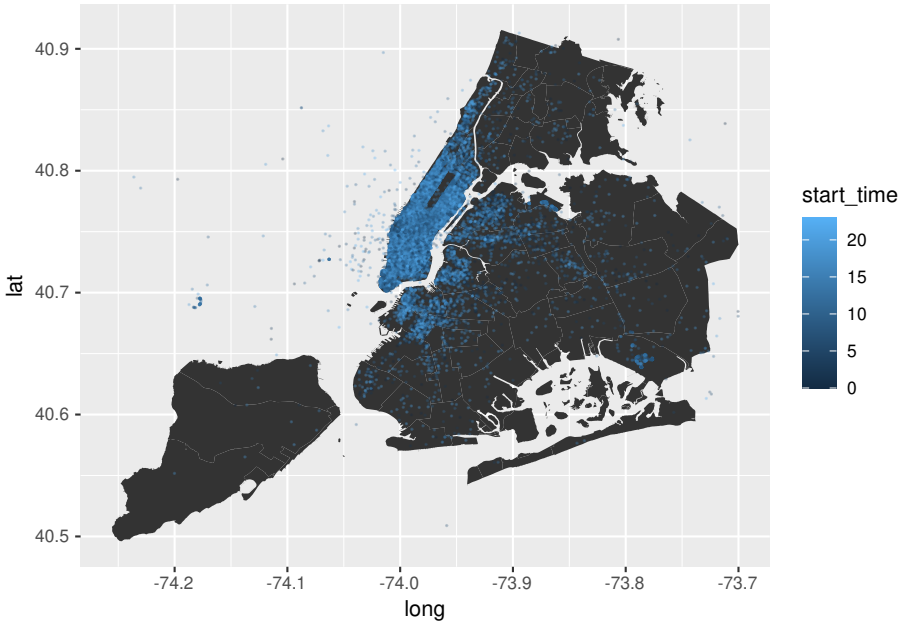
```
# drop-off locations
locations +
     geom_point(aes(x=dest_long, y=dest_lat, color = start_time ),
                size = 0.1,
                alpha = 0.2) +
     scale_colour_gradient2( low = "red", mid = "yellow", high = "red",
                             midpoint = 12)
```



8.4 Excursus: change color schemes

In the example above we use `scale_colour_gradient2()` to modify the color gradient used to visualize the start time of taxi trips. By default, ggplot would plot the following (default gradient color setting):

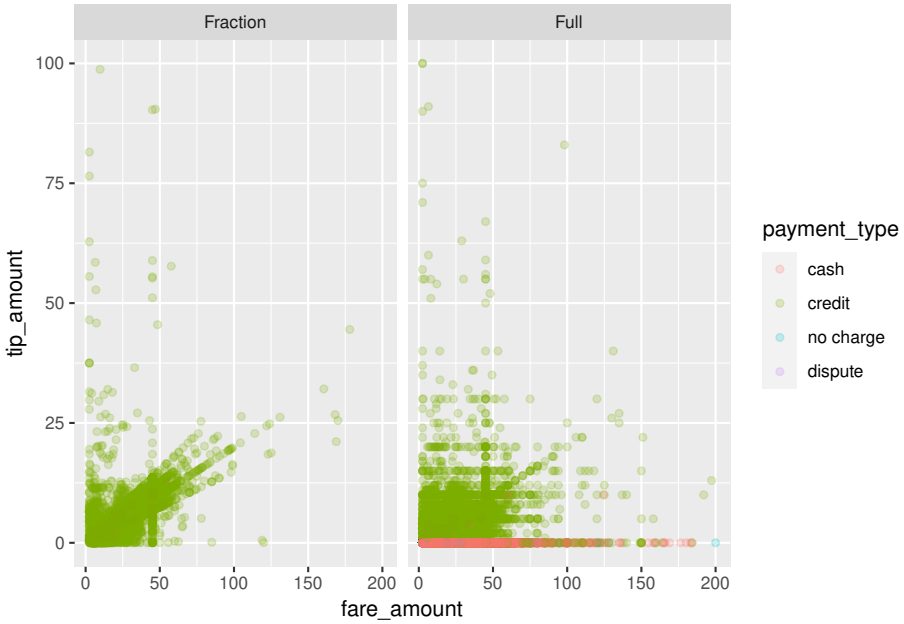
```
# drop-off locations
locations +
     geom_point(aes(x=dest_long, y=dest_lat, color = start_time ),
                size = 0.1,
                alpha = 0.2)
```



`ggplot2` offers various functions to modify the color scales used in a plot. In the case of the example above, we visualize values of a continuous variable. Hence we use a gradient color scale. In case of categorical variables we need to modify the default discrete color scale.

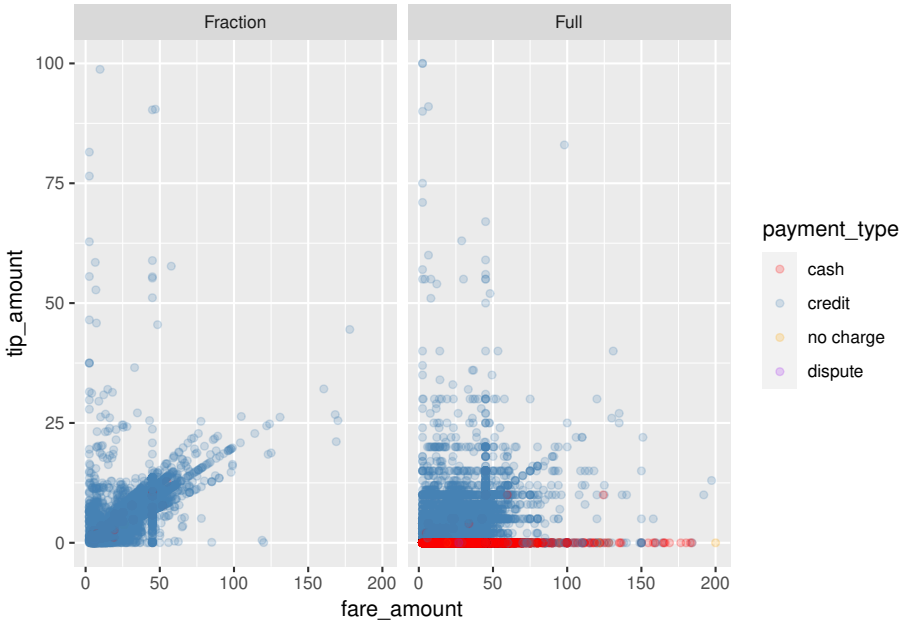
Recall the plot illustrating tipping behavior, where we highlight in which observations the client payd with credit card, cash, etc.

```
# indicate natural numbers
taxi[, dollar_paid := ifelse(tip_amount == round(tip_amount,0), "Full", "Fraction"),]


# extended x/y plot
taxiplot +
     geom_point(alpha=0.2, aes(color=payment_type)) +
     facet_wrap("dollar_paid")
```



Since we do not further specify the discrete color scheme to be used, ggplot simply uses its default color scheme for this plot. We can change this as follows.

```
# indicate natural numbers
taxi[, dollar_paid := ifelse(tip_amount == round(tip_amount,0), "Full", "Fraction"),]


# extended x/y plot
taxiplot +
     geom_point(alpha=0.2, aes(color=payment_type)) +
     facet_wrap("dollar_paid") +
     scale_color_discrete(type = c("red", "steelblue", "orange", "purple"))
```


Fraction
Full
tip_amount
fare_amount
payment_type
cash
credit
no charge
dispute
0
25
50
75
100
0
50
100
150
200

9

Cloud Services for Big Data Analytics

So far we have focused on how to deal with large amounts of data and/or computationally demanding tasks on our local machines (desktop/laptop). A key aspect of this course has thus been in a first step to understand why our local machine is struggling with a data analysis task when there is a large amount of data to be processed. In a second step we have looked into practical solutions to these challenges. These solutions are in essence tools (in this course particularly tools provided in the R environment) to use the key components of our computing environment (CPU, RAM, mass storage) most efficiently:

- Computationally intense tasks (but not pushing RAM to the limit): parallelization, using several CPU cores (nodes) in parallel.
- Memory-intense tasks (data still fits into RAM): efficient memory allocation (`data.table`-package).
- Memory-intense tasks (data does not fit into RAM): efficient use of virtual memory (use parts of mass storage device as virtual memory).
- Big Data storage: efficient storage (avoid redundancies) and efficient access (speed) with RDBMSs (here: SQLite).

In practice, data sets might be too large for our local machine even if we take all of the techniques listed above into account. That is, a parallelized task might still take ages to complete because our local machine has too few cores available, a task involving virtual memory would use up way too much space on our hard-disk, running a large SQL database locally would use up too much resources, etc.

In such situations, we have to think about horizontal and vertical scaling beyond our local machine. That is, we outsource tasks to a bigger machine (or a cluster of machines) to which our local computer is connected over the Internet (or over a local network). While a decade or two ago most organizations had their own large centrally hosted

machines (database servers, cluster computers) for such tasks, today they often rely on third-party solutions *'in the cloud'*. That is, specialized companies provide flexible access to computing resources that can be easily accessed via a broadband Internet-connection and rented on an hourly (or even minutes and seconds) basis. Given the obvious economies of scale in this line of business, a few large players have emerged who practically dominate most of the global market:

- Amazon Web Services (AWS)[1].
- Microsoft Azure[2]
- Google Cloud Platform[3]
- IBM Cloud[4]
- Alibaba Cloud[5]
- Tencent Cloud[6]
- ...

For details on what some of the largest platforms provide, see the overview in the online chapter to Walkowiak (2016) 'Pushing R Further'[7].

When we use such cloud services to *scale up* (vertical scaling) the computing resources, the transition from our local implementation of a data analytics task to the cloud implementation is often rather simple. Once we have set up a cloud instance and figured out how to communicate with it, we typically can even run the exact same R-script locally or in the cloud. This is usually the case for parallelized tasks (simply run the same script on a machine with more cores), in-memory tasks (rent a machine with more RAM but still use `data.table()` etc.), and working with an SQL database (simply set up the database in the cloud instead of locally).

However, for really memory-intense tasks, the cloud provides options

[1] https://aws.amazon.com/

[2] https://azure.microsoft.com/en-us/

[3] https://cloud.google.com/

[4] https://www.ibm.com/cloud/

[5] https://www.alibabacloud.com/

[6] https://intl.cloud.tencent.com/

[7] https://www.packtpub.com/sites/default/files/downloads/5396_6457OS_PushingRFurther.pdf

to *scale out* (horizontal scaling). Meaning, a task is distributed among a cluster of computing instances/servers. The implementation of such data analytics tasks is based on a paradigm that we rather do not encounter when working locally: *Map/Reduce* (implemented in the *Hadoop* framework).

In the following we look first at scaling up more familiar approaches with the help of the cloud and then look at the Map/Reduce concept and how it can be applied in *Hadoop* running on cloud instances.

9.1 Scaling up in the Cloud

In the following examples we use different cloud services provided by AWS. See the online chapter to Walkowiak (2016) 'Pushing R Further'[8] for how to set up an AWS account and the basics for how to set up AWS instances. The examples below are based on the assumption that the EC2 instance and RStudio Server have been set up essentially as explained in 'Pushing R Further'[9], pages 22-38. However, AWS changed a few things regarding the way their linux machines are set up since the online chapter was published first. In order to install core R on your ec2 instance, use the following command in the terminal (instead of `sudo yum install R`):

```
sudo amazon-linux-extras install R3.4
```

and confirm the installation of additional dependencies with `y`. In a second step, we install the latest CentOS 6-7 version of RStudio Server with the following commands.

```
# April 2020
```

[8] https://www.packtpub.com/sites/default/files/downloads/5396_6457OS_PushingRFurther.pdf

[9] https://www.packtpub.com/sites/default/files/downloads/5396_6457OS_PushingRFurther.pdf

```
wget https://download2.rstudio.org/server/centos6/x86_64/rstudio-server-rhel-1.2.5033-x86_64.r
sudo yum install rstudio-server-rhel-1.2.5033-x86_64.rpm
```

9.1.1 Parallelization with an EC2 instance

This short tutorial illustrates how to scale the computation of clustered standard errors shown in Lecture 3 up by running it on an AWS EC2 instance. Below we use the same source code as in the original example (see `03_computation_memory.Rmd`[10]). Note that there are a few things that we need to keep in mind in order to make the script run on an AWS EC2 instance in RStudio Server.

First, our EC2 instance is a Linux machine. Most of you are probably rather used to running R on a Mac or Windows PC. When running R on a Linux machine, there is an additional step to install R packages (at least for most of the packages): R packages need to be compiled before they can be installed. The command to install packages is exactly the same (`install.packages()`) and normally you only notice a slight difference in the output shown in the R console during installation (and the installation process takes a little longer than what you are used to). Apart from that, using R via RStudio Server in the cloud looks/feels very similar if not identical as when using R/RStudio locally.

Now, let's go through the bootstrap example. First, let's run the non-parallel implementation of the script. When executing the code below line-by-line, you will notice that essentially all parts of the script work exactly as on your local machine. This is one of the great advantages of running R/RStudio Server in the cloud. You can implement your entire data analysis locally (based on a small sample), test it locally, and then move it to the cloud and run it on a larger scale in exactly the same way (even with the same GUI).

```
# CASE STUDY: PARALLEL ---------------------------
```

[10] https://github.com/umatter/BigData/blob/master/materials/notes/03_computation_memory.Rmd

```
# install packages
install.packages("data.table")
install.packages("doSNOW")

# load packages
library(data.table)


## ------------------------------------------------------------------------
stopdata <- read.csv("https://vincentarelbundock.github.io/Rdatasets/csv/carData/MplsStops.csv

## ------------------------------------------------------------------------
# remove incomplete obs
stopdata <- na.omit(stopdata)
# code dependent var
stopdata$vsearch <- 0
stopdata$vsearch[stopdata$vehicleSearch=="YES"] <- 1
# code explanatory var
stopdata$white <- 0
stopdata$white[stopdata$race=="White"] <- 1

## ------------------------------------------------------------------------
model <- vsearch ~ white + factor(policePrecinct)

## ------------------------------------------------------------------------
fit <- lm(model, stopdata)
summary(fit)


# bootstrapping: normal approach

## ----message=FALSE-------------------------------------------------------

# set the 'seed' for random numbers (makes the example reproducible)
set.seed(2)
```

```
# set number of bootstrap iterations
B <- 50
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)
# draw bootstrap samples, estimate model for each sample
for (i in 1:B) {

  # draw sample of precincts (cluster level)
  precincts_i <- sample(precincts, size = 5, replace = TRUE)
  # get observations
  bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
  bs_i <- rbindlist(bs_i)

  # estimate model and record coefficients
  boot_coefs[i,] <- coef(lm(model, bs_i))[1:2] # ignore FE-coefficients
}

## ------------------------------------------------------------------------
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

So far, we have only demonstrated that the simple implementation (non-parallel) works both locally and in the cloud. The real purpose of using an EC2 instance in this example is to make use of the fact that we can scale up our instance to have more CPU cores available for the parallel implementation of our bootstrap procedure. Recall that running the script below on our local machine will employ all cores available to an compute the bootstrap resampling in parallel on all these cores. Exactly the same thing happens when running the code below on our simple `t2.micro` instance. However this type of EC2 instance only has one core. You can check this when running the following line of code in RStudio Server (assuming the `doSNOW` package is installed and loaded):

```
parallel::detectCores()
```

When running the entire parallel implementation below, you will thus notice that it won't compute the bootstrap SE any faster than with the non-parallel version above. However, by simply initiating another EC2 type with more cores, we can distribute the workload across many CPU cores, using exactly the same R-script.

```
# bootstrapping: parallel approaach

## ----message=FALSE------------------------------------------------------------
# install.packages("doSNOW", "parallel")
# load packages for parallel processing
library(doSNOW)

# get the number of cores available
ncores <- parallel::detectCores()
# set cores for parallel processing
ctemp <- makeCluster(ncores) #
registerDoSNOW(ctemp)


# set number of bootstrap iterations
B <- 50
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)

# bootstrapping in parallel
boot_coefs <-
  foreach(i = 1:B, .combine = rbind, .packages="data.table") %dopar% {

    # draw sample of precincts (cluster level)
    precincts_i <- sample(precincts, size = 5, replace = TRUE)
    # get observations
```

```
    bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
    bs_i <- rbindlist(bs_i)

    # estimate model and record coefficients
    coef(lm(model, bs_i))[1:2] # ignore FE-coefficients

  }


# be a good citizen and stop the snow clusters
stopCluster(cl = ctemp)



## ------------------------------------------------------------------------
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

9.1.2 Mass Storage: MariaDB on an EC2 instance

Once we have set up RStudio Server on an EC2 instance, we can run the SQLite examples demonstrated locally in Lecture 7 on it. There are no additional steps needed to install SQLite. However, when using RDBMSs in the cloud, we typically have a more sophisticated implementation than SQLite in mind. Particularly, we want to set up an actual RDBMS-server running in the cloud to which several clients can connect (via RStudio Server). The following example, based on Walkowiak (2016), guides you through the first step to set up such a database in the cloud. To keep things simple, the example sets up a database of the same data set as shown in the first SQLite example in Lecture 7, but this time with MariaDB on an EC2 instance in the cloud. For most of the installation steps you are referred to the respective pages in Walkowiak (2016) (Chapter 5: 'MariaDB with R on a Amazon EC2 instance, pages 255ff). However, since some of the steps shown

in the book are outdated, the example below hints to some alternative/additional steps needed to make the database run on an Ubuntu 18.04 machine.

After launching the EC2 instance on AWS, use the following terminal commands to install R:

```
# update ubuntu packages
 sudo apt-get update
 sudo apt-get upgrade
```

```
sudo apt-get install r-base
```

and to install RStudio Server (on Ubuntu 18.04, as of April 2020):

```
sudo apt-get install gdebi-core
wget https://download2.rstudio.org/server/bionic/amd64/rstudio-server-1.2.5033-amd64.deb
sudo gdebi rstudio-server-1.2.5033-amd64.deb
```

Following Walkowiak (2016) (pages 257f), we first set up a new user and give it permissions to `ssh` directly to the EC2 instance (this way we can then more easily upload data 'for this user').

```
# create user
sudo adduser umatter
```

When prompted for additional information just hit enter (for default). Now we can grant the user the permissions

```
sudo cp -r /home/ubuntu/.ssh /home/umatter/
cd /home/umatter/
sudo chown -R umatter:umatter .ssh
```

Then install MariaDB as follows.

```
sudo apt update
sudo apt install mariadb-server
sudo apt install libmariadbclient-dev
sudo apt install libxml2-dev # needed later (dependency for some R packages)
```

If prompted to set a password for the root database user (user with all database priviledges), type in and confirm the chosen password.[11]

9.1.2.1 Data import

With the permissions set above, we can send data from the local machine directly to the instance via `ssh`. We use this to first transfer the raw data to the instance and then import it to the database.

The aim is to import the same simple data set `economics.csv` used in the local SQLite examples of Lecture 7. Following the instructions of Walkowiak (2016), pages 252 to 254, we upload the `economics.csv` file (instead of the example data used in Walkowiak (2016)). Note that in all the code examples below, the username is `umatter`, and the IP-address will have to be replaced with the public IP-address of your EC2 instance.

Open a new terminal window and send the `economics.csv` data as follows to the instance.

```
# from the directory where the key-file is stored...
scp -r -i "mariadb_ec2.pem" ~/Desktop/economics.csv umatter@ec2-184-72-202-166.compute-1.amazo
```

Then switch back to the terminal connected to the instance and start the MariaDB server.

```
# start the MariaDB server
sudo service mysql start
# log into the MariaDB client as root
sudo mysql -uroot
```

[11] Below it is shown how to do this 'manually', if not promted at this step.

If not prompted to do so when installing MariaDB (see above), add a new root user in order to login to MariaDB without the `sudo` (here we simply set the password to 'Password1').

```
GRANT ALL PRIVILEGES on *.* to 'root'@'localhost' IDENTIFIED BY 'Password1';
FLUSH PRIVILEGES;
```

Restart the mysql server and log in with the database root user.

```
# start the MariaDB server
sudo service mysql restart
# log into the MariaDB client as root
mysql -uroot -p
```

Now we can initiate a new database called `data1`.

```
CREATE database data1;
```

To work with the newly created database, we have to 'select' it.

```
USE data1;
```

Then, we create the first table of our database and import data into it. Note that we only have to slightly adjust the former SQLite syntax to make this work (remove double quotes for field names). In addition, note that we can use the same field types as in the SQLite DB.[12]

```
-- Create the new table
CREATE TABLE econ(
date DATE,
pce REAL,
pop INTEGER,
psavert REAL,
```

[12] However, MariaDB is a much more sophisticated RDBMS than SQLite and comes with many more field types, see the official list of supported data types[13].

```
uempmed REAL,
unemploy INTEGER
);
```

After following the steps in Walkowiak (2016), pages 259-262, we can import the `economics.csv`-file to the `econ` table in MariaDB (again, assuming the username is `umatter`). Note that the syntax to import data to a table is quite different from the SQLite example in Lecture 7.

```
LOAD DATA LOCAL INFILE
'/home/umatter/economics.csv'
INTO TABLE econ
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Now we can start using the newly created database from within RStudio Server running on our EC2 instance (following Walkowiak (2016), pages 263ff).

As in the SQLite examples in Lecture 7, we can now query the database from within the R console (this time using `RMySQL` instead of `RSQLite`, and using R from within RStudio Server in the cloud!).

First, we need to connect to the newly created MariaDB database.

```
# install package
#install.packages("RMySQL")
# load packages
library(RMySQL)

# connect to the db
con <- dbConnect(RMySQL::MySQL(),
                 user = "root",
                 password = "Password1",
```

```
                host = "localhost",
                dbname = "data1")
```

In our first query, we select all (*) variable values of the observation of January 1968.

```
# define the query
query1 <-
"
SELECT * FROM econ
WHERE date = '1968-01-01';
"
# send the query to the db and get the result
jan <- dbGetQuery(con, query1)
jan
```

```
#        date   pce    pop psavert uempmed unemploy
# 1 1968-01-01 531.5 199808    11.7     5.1     2878
```

Now let's select all year/months in which there were more than 15 million unemployed, ordered by date.

```
query2 <-
"
SELECT date FROM econ
WHERE unemploy > 15000
ORDER BY date;
"

# send the query to the db and get the result
unemp <- dbGetQuery(con, query2)
head(unemp)
```

```
#         date
# 1 2009-09-01
# 2 2009-10-01
```

```
# 3 2009-11-01
# 4 2009-12-01
# 5 2010-01-01
# 6 2010-02-01
```

When done working with the database, we close the connection to the MariaDB database with `dbDisconnect(con)`.

9.2 Distributed Systems/MapReduce

9.2.1 Map/Reduce Concept: Illustration in R

In order to better understand the basic concept behind the MapReduce-Framework on a distributed system, let's look at how we can combine the basic functions `map()` and `reduce()` in R to implement the basic MapReduce example shown in Walkowiak (2016), Chapter 4, pages 132-134 (this is just to illustrate the underlying idea, *not* to suggest that MapReduce actually is simply an application of the classical `map` and `reduce (fold)` functions in functional programming).[14] The overall aim of the program is to count the number of times each word is repeated in a given text. The input to the program is thus a text, the output is a list of key-value pairs with the unique words occurring in the text as keys and their respective number of occurrences as values.

In the code example, we will use the following text as input.

```
input_text <-
"Simon is a friend of Becky.
Becky is a friend of Ann.
Ann is not a friend of Simon."
```

[14] For a more detailed discussion of what `map` and `reduce` have *actually* to do with MapReduce see this post[15].

9.2.1.1 Mapper

The Mapper first splits the text into lines, and then splits the lines into key-value pairs, assigning to each key the value `1`. For the first step we use `strsplit()` that takes a character string as input and splits it into a list of substrings according to the matches of a substring (here `"\n"`, indicating the end of a line).

```
# Mapper splits input into lines
lines <- as.list(strsplit(input_text, "\n")[[1]])
lines
```

```
## [[1]]
## [1] "Simon is a friend of Becky."
##
## [[2]]
## [1] "Becky is a friend of Ann."
##
## [[3]]
## [1] "Ann is not a friend of Simon."
```

In a second step, we apply our own function (`map_fun()`) to each line of text via `Map()`. `map_fun()` splits each line into words (keys) and assigns a value of `1` to each key.

```
# Mapper splits lines into Key-Value pairs
map_fun <-
     function(x){

          # remove special characters
          x_clean <- gsub("[[:punct:]]", "", x)
          # split line into words
          keys <- unlist(strsplit(x_clean, " "))
          # initiate key-value pairs
          key_values <- rep(1, length(keys))
          names(key_values) <- keys
```

```
          return(key_values)
     }

kv_pairs <- Map(map_fun, lines)

# look at the result
kv_pairs
```

```
## [[1]]
##  Simon     is      a friend     of  Becky
##      1      1      1      1      1      1
##
## [[2]]
##  Becky     is      a friend     of    Ann
##      1      1      1      1      1      1
##
## [[3]]
##    Ann     is    not      a friend     of  Simon
##      1      1      1      1      1      1      1
```

9.2.1.2 Reducer

The Reducer first sorts and shuffles the input from the Mapper and then reduces the key-value pairs by summing up the values for each key.

```
# order and shuffle
kv_pairs <- unlist(kv_pairs)
keys <- unique(names(kv_pairs))
keys <- keys[order(keys)]
shuffled <- lapply(keys,
                   function(x) kv_pairs[x == names(kv_pairs)])
shuffled
```

```
## [[1]]
## a a a
## 1 1 1
```

```
##
## [[2]]
## Ann Ann
##   1   1
##
## [[3]]
## Becky Becky
##     1     1
##
## [[4]]
## friend friend friend
##      1      1      1
##
## [[5]]
## is is is
##  1  1  1
##
## [[6]]
## not
##   1
##
## [[7]]
## of of of
##  1  1  1
##
## [[8]]
## Simon Simon
##     1     1
```

Now we can sum up the keys in order to get the word count for the entire input.

```
sums <- sapply(shuffled, sum)
names(sums) <- keys
sums
```

```
##      a    Ann  Becky friend     is    not     of
##      3      2      2      3      3      1      3
```

```
##  Simon
##      2
```

9.2.1.3 Simpler example: Compute the total number of words

```
# assigns the number of words per line as value
map_fun2 <-
     function(x){
          # remove special characters
          x_clean <- gsub("[[:punct:]]", "", x)
          # split line into words, count no. of words per line
          values <- length(unlist(strsplit(x_clean, " ")))
          return(values)
     }
# Mapper
mapped <- Map(map_fun2, lines)
mapped
```

```
## [[1]]
## [1] 6
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 7
```

```
# Reducer
reduced <- Reduce(sum, mapped)
reduced
```

```
## [1] 19
```

9.3 Hadoop Word Count

Example adapted from this tutorial[16] by Melissa Anderson and Hanif Jetha.

9.3.1 Install Hadoop (on Ubuntu/Pop!_OS Linux)

```
# download binary
wget https://downloads.apache.org/hadoop/common/hadoop-2.10.0/hadoop-2.10.0.tar.gz
# download checksum
wget https://www.apache.org/dist/hadoop/common/hadoop-2.10.0/hadoop-2.10.0.tar.gz.sha512

# run the verification
shasum -a 512 hadoop-2.10.0.tar.gz
# compare with value in mds file
cat hadoop-2.10.0.tar.gz.sha512

# if all is fine, unpack
tar -xzvf hadoop-2.10.0.tar.gz
# move to proper place
sudo mv hadoop-2.10.0 /usr/local/hadoop



# then point to this version from hadoop
# open the file /usr/local/hadoop/etc/hadoop/hadoop-env.sh
# in a text editor and add (where export JAVA_HOME=...)
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")

# clean up
rm hadoop-2.10.0.tar.gz
rm hadoop-2.10.0.tar.gz.sha512
```

[16] https://www.digitalocean.com/community/tutorials/how-to-install-hadoop-in-stand-alone-mode-on-ubuntu-18-04#step-2-%E2%80%94-installing-hadoop

9.3.2 Run Hadoop

```
# check installation
/usr/local/hadoop/bin/hadoop
```

9.3.3 Run example

The basic Hadoop installation comes with a few examples for very typical map/reduce programs.[17] Below we replicate the same word-count example as shown in simple R code above.

In a first step, we create an input directory where we store the input file(s) to feed to Hadoop.

```
# create directory for input files (typically text files)
mkdir ~/input
```

Then we add a textfile containing the same text as in the example as above (to make things simpler, we already remove special characters).

```
echo "Simon is a friend of Becky
Becky is a friend of Ann
Ann is not a friend of Simon" >>  ~/input/text.txt
```

Now we can run the MapReduce/Hadoop word count as follows, storing the results in a new directory called `wordcount_example`.

```
# run mapreduce word count
/usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-exa
```

Show the output:

```
cat ~/wc_example/*
```

[17] More sophisticated programs need to be custom made, written in Java.

```
## Ann  2
## Becky    2
## Simon    2
## a    3
## friend   3
## is   3
## not  1
## of   3
```

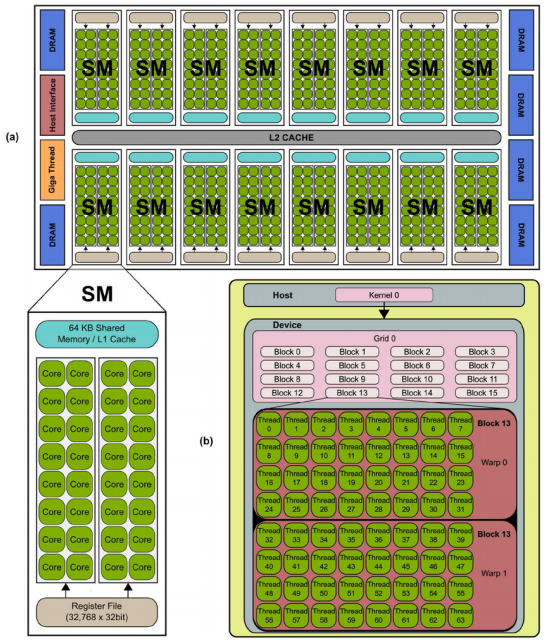
10

GPUs for Scientific Computing

The success of the computer games industry in the late 1990s/early 2000s led to an interesting positive externality for scientific computing. The ever more demanding graphics of modern computer games and the huge economic success of the computer games industry set incentives for hardware producers to invest in research and development of more powerful 'graphic cards', extending a normal PC/computing environment with additional computing power solely dedicated to graphics. At the heart of these graphic cards are so-called GPUs (Graphic Processing Units), microprocessors specifically optimized for graphics processing. The image below depicts a modern graphics card with NVIDIA GPUs, which is quite common in today's 'gaming' PCs.

Why did the hardware industry not simply invest in the development of more powerful CPUs to deal with the more demanding PC games? The main reason is that the architecture of CPUs is designed not only for efficiency but also flexibility. That is, a CPU needs to perform well in all kind of computations, some parallel, some sequential, etc. Computing graphics is a comparatively narrow domain of computation and designing a processing unit architecture that is custom-made to excel just at this one task is thus much more cost efficient. Interestingly, this graphics-specific architecture (specialized on highly paral-

lel numerical [floating point] workloads) turns out to be also very useful in some core scientific computing tasks. In particular, matrix multiplications (see Fatahalian et al. (2004) for a detailed discussion of why that is the case). A key aspect of GPUs is that they are composed of several multiprocessor units, of which each has in turn several cores. GPUS thus can perform computations with hundreds or even thousands of threads in parallel. The figure below illustrates this point.



While, initially, programming GPUs for scientific computing required a very good understanding of the hardware. Graphics card producers have realized that there is an additional market for their products (in particular with the recent rise of deep learning), and provide several high-level APIs to use GPUs for other tasks than graphics processing. Over the last few years more high-level software has been developed, which makes it much easier to use GPUs in parallel computing tasks. The following subsections shows some examples of such software in the R environment.[1]

[1]Note that while these examples are easy to implement and run, setting up a GPU for scientific computing still can involve many steps and some knowledge of your

10.1 GPUs in R

10.1.1 Example I: Matrix multiplication comparison (gpuR)

The gpuR package provides basic R functions to compute with GPUs from within the R environmnent. In the following example we compare the performance of the CPU with the GPU based on a matrix multiplication exercise. For a large $N \times P$ matrix X, we want to compute $X^t X$.

In a first step, we load the gpuR-package.[2] Note the output to the console. It shows the type of GPU identified by gpuR. This is the platform on which gpuR will compute the GPU examples. In order to compare the performances, we also load the bench package used in previous lectures.

```
# load package
library(bench)
library(gpuR)
```

```
## Number of platforms: 1
## - platform: NVIDIA Corporation: OpenCL 3.0 CUDA 11.4.158
##   - context device index: 0
##     - NVIDIA GeForce GTX 1650
## checked all devices
## completed initialization
```

Next, we initiate a large matrix filled with pseudo random numbers, representing a dataset with N observations and P variables.

computer's system. The examples presuppose that all installation and configuration steps (GPU drivers, CUDA, etc.) have already been completed successfully.

[2] As with the setting up of GPUs on your machine in general, installing all prerequisites to make gpuR work on your local machine can be a bit of work and can depend a lot on your system.

```
# initiate dataset with pseudo random numbers
N <- 10000  # number of observations
P <- 100 # number of variables
X <- matrix(rnorm(N * P, 0, 1), nrow = N, ncol =P)
```

For the GPU examples to work, we need one more preparatory step. GPUs have their own memory, which they can access faster than they can access RAM. However, this GPU memory is typically not very large compared to the memory CPUs have access to. Hence, there is a potential trade-off between losing some efficiency but working with more data or vice versa.[3] Here, we show both variants. With `gpuMatrix()` we create an object representing matrix `X` for computation on the GPU. However, this only points the GPU to the matrix and does not actually transfer data to the GPU's memory. The latter is done in the other variant with `vclMatrix()`.

```
# prepare GPU-specific objects/settings
gpuX <- gpuMatrix(X, type = "float")  # point GPU to matrix (matrix stored in non-GPU memory)
vclX <- vclMatrix(X, type = "float")  # transfer matrix to GPU (matrix stored in GPU memory)
```

Now we run the three examples: first, based on standard R, using the CPU. Then, computing on the GPU but using CPU memory. And finally, computing on the GPU and using GPU memory. In order to make the comparison fair, we force `bench::mark()` to run at least 20 iterations per benchmarked variant.

```
# compare three approaches
(gpu_cpu <- bench::mark(

  # compute with CPU
  cpu <- t(X) %*% X,

  # GPU version, GPU pointer to CPU memory (gpuMatrix is simply a pointer)
```

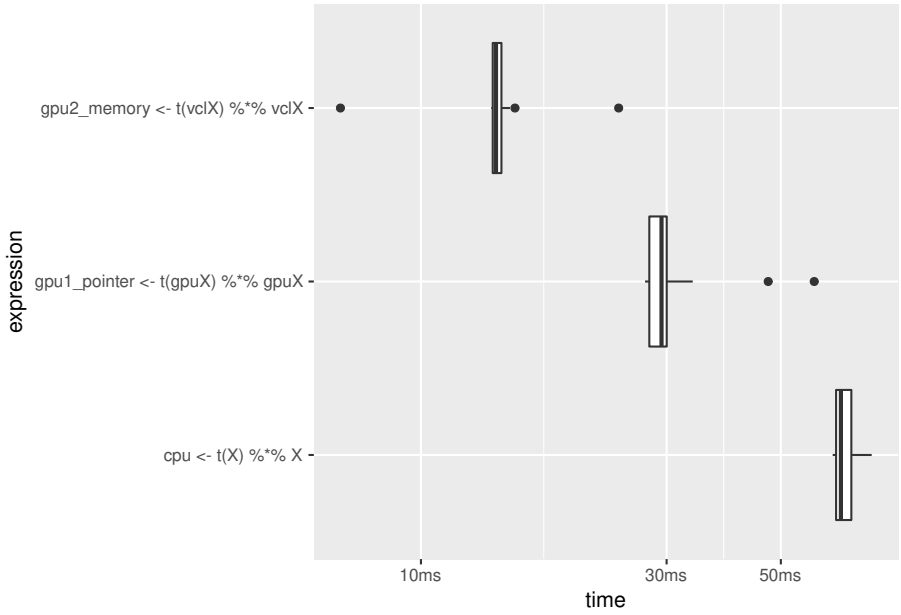
[3] If we instruct the GPU to use the own memory, but the data does not fit in it, the program will result in an error.

```
  gpu1_pointer <- t(gpuX) %*% gpuX,

  # GPU version, in GPU memory (vclMatrix formation is a memory transfer)
  gpu2_memory <- t(vclX) %*% vclX,

check = FALSE, memory = FALSE, min_iterations = 20))
```

```
## # A tibble: 3 x 6
##   expression                              min   median
##   <bch:expr>                          <bch:tm> <bch:tm>
## 1 cpu <- t(X) %*% X                    63.05ms   65.4ms
## 2 gpu1_pointer <- t(gpuX) %*% gpuX     27.24ms     29ms
## 3 gpu2_memory <- t(vclX) %*% vclX       6.98ms     14ms
## # ... with 3 more variables: itr/sec <dbl>,
## #   mem_alloc <bch:byt>, gc/sec <dbl>
```

The performance comparison is visualized with boxplots.

```
plot(gpu_cpu, type = "boxplot")
```



10.2 GPUs and Machine Learning

A most common application of GPUs for scientific computing is machine learning, in particular deep learning (machine learning based on artificial neural networks). Training deep learning models can be very computationally intense and to an important part depends on tensor (matrix) multiplications. This is also an area where you might come across highly parallelized computing based on GPUs without even noticing it, as the now commonly used software to build and train deep neural nets (tensorflow[4], and the high-level Keras[5] API) can easily be run on a CPU or GPU without any further configuration/preparation (apart from the initial installation of these programs). The example below is a simple illustration of how such techniques can be used in an econometrics context.

10.2.1 Tensorflow/Keras example: predict housing prices

In this example we train a simple sequential model with two hidden layers in order to predict the median value of owner-occupied homes (in USD 1,000) in the Boston area (data are from the 1970s). The original data and a detailed description can be found here[6]. The example follows closely this keras tutorial[7] published by RStudio. See RStudio's keras installation guide[8] for how to install keras (and tensorflow) and the corresponding R package `keras`.[9] While the purpose of the example here is to demonstrate a typical (but very simple!) usage case of GPUs in machine learning, the same code should also run on a normal machine (without using GPUs) with a default installation of keras.

Apart from `keras`, we load packages to prepare the data and visualize

[4] https://www.tensorflow.org/

[5] https://keras.io/

[6] https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html

[7] https://keras.rstudio.com/articles/tutorial_basic_regression.html#the-boston-housing-prices-dataset

[8] https://keras.rstudio.com/index.html

[9] This might involve the installation of additional packages and software outside the R environment.

the output. Via `dataset_boston_housing()`, we load the dataset (shipped with the keras installation) in the format preferred by the `keras` library.

```
# load packages
library(keras)
library(tibble)
library(ggplot2)
library(tfdatasets)


# load data
boston_housing <- dataset_boston_housing()
str(boston_housing)
```

```
## List of 2
##  $ train:List of 2
##   ..$ x: num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
##  ..$ y: num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
##  $ test :List of 2
##   ..$ x: num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
##  ..$ y: num [1:102(1d)] 7.2 18.8 19 27 22.2 24.5 31.2 22.9 20.5 23.2 ...
```

In a first step, we split the data into a training set and a test set. The latter is used to monitor the out-of-sample performance of the model fit. Testing the validity of an estimated model by looking at how it performs out-of-sample is of particular relevance when working with (deep) neural networks, as they can easily lead to over-fitting. Validity checks based on the test sample are, therefore, often an integral part of modelling with tensorflow/keras.

```
# assign training and test data/labels
c(train_data, train_labels) %<-% boston_housing$train
c(test_data, test_labels) %<-% boston_housing$test
```

In order to better understand and interpret the dataset we add the original variable names, and convert it to a `tibble`.

```
library(dplyr)

column_names <- c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
                  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT')

train_df <- train_data %>%
  as_tibble(.name_repair = "minimal") %>%
  setNames(column_names) %>%
  mutate(label = train_labels)

test_df <- test_data %>%
  as_tibble(.name_repair = "minimal") %>%
  setNames(column_names) %>%
  mutate(label = test_labels)
```

Next, we have a close look at the data. Note the usage of the term 'label' for what is usually called the 'dependent variable' in econometrics.[10] As the aim of the exercise is to predict median prices of homes, the output of the model will be a continuous value ('labels').

```
# check example data dimensions and content
paste0("Training entries: ", length(train_data), ", labels: ", length(train_labels))
```

```
## [1] "Training entries: 5252, labels: 404"
```

```
summary(train_data)
```

```
##        V1               V2               V3
##  Min.   : 0.01   Min.   :  0.0   Min.   : 0.46
##  1st Qu.: 0.08   1st Qu.:  0.0   1st Qu.: 5.13
##  Median : 0.27   Median :  0.0   Median : 9.69
##  Mean   : 3.75   Mean   : 11.5   Mean   :11.10
```

[10] Typical textbook examples in machine learning deal with classification (e.g. a logit model), while in microeconometrics the typical example is usually a linear model (continuous dependent variable).

```
##  3rd Qu.: 3.67   3rd Qu.: 12.5   3rd Qu.:18.10
##  Max.   :88.98   Max.   :100.0   Max.   :27.74
##        V4               V5              V6
##  Min.   :0.0000   Min.   :0.385   Min.   :3.56
##  1st Qu.:0.0000   1st Qu.:0.453   1st Qu.:5.88
##  Median :0.0000   Median :0.538   Median :6.20
##  Mean   :0.0619   Mean   :0.557   Mean   :6.27
##  3rd Qu.:0.0000   3rd Qu.:0.631   3rd Qu.:6.61
##  Max.   :1.0000   Max.   :0.871   Max.   :8.72
##        V7              V8              V9
##  Min.   :  2.9   Min.   : 1.13   Min.   : 1.00
##  1st Qu.: 45.5   1st Qu.: 2.08   1st Qu.: 4.00
##  Median : 78.5   Median : 3.14   Median : 5.00
##  Mean   : 69.0   Mean   : 3.74   Mean   : 9.44
##  3rd Qu.: 94.1   3rd Qu.: 5.12   3rd Qu.:24.00
##  Max.   :100.0   Max.   :10.71   Max.   :24.00
##       V10            V11             V12
##  Min.   :188   Min.   :12.6   Min.   :  0.3
##  1st Qu.:279   1st Qu.:17.2   1st Qu.:374.7
##  Median :330   Median :19.1   Median :391.2
##  Mean   :406   Mean   :18.5   Mean   :354.8
##  3rd Qu.:666   3rd Qu.:20.2   3rd Qu.:396.2
##  Max.   :711   Max.   :22.0   Max.   :396.9
##       V13
##  Min.   : 1.73
##  1st Qu.: 6.89
##  Median :11.39
##  Mean   :12.74
##  3rd Qu.:17.09
##  Max.   :37.97
```

```
summary(train_labels) # Display first 10 entries
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     5.0    16.7    20.8    22.4    24.8    50.0
```

As the dataset contains variables ranging from per capita crime rate

to indicators for highway access, the variables are obviously measured in different units and hence displayed on different scales. This is not per se a problem for the fitting procedure. However, fitting is more efficient when all features (variables) are normalized.

```
spec <- feature_spec(train_df, label ~ . ) %>%
  step_numeric_column(all_numeric(), normalizer_fn = scaler_standard()) %>%
  fit()

layer <- layer_dense_features(
  feature_columns = dense_features(spec),
  dtype = tf$float32
)
layer(train_df)
```

```
## tf.Tensor(
## [[ 0.81205493  0.44752213 -0.2565147  ... -0.1762239  -0.59443307
##   -0.48301655]
##  [-1.9079947   0.43137115 -0.2565147  ...  1.8920003  -0.34800112
##    2.9880793 ]
##  [ 1.1091131   0.2203439  -0.2565147  ... -1.8274226   1.563349
##   -0.48301655]
##  ...
##  [-1.6359899   0.07934052 -0.2565147  ... -0.3326088  -0.61246467
##    0.9895695 ]
##  [ 1.0554279  -0.98642045 -0.2565147  ... -0.7862657  -0.01742171
##   -0.48301655]
##  [-1.7970455   0.23288251 -0.2565147  ...  0.47467488 -0.84687555
##    2.0414166 ]], shape=(404, 13), dtype=float32)
```

We specify the model as a linear stack of layers: The input (all 13 explanatory variables), two densely connected hidden layers (each with a 64-dimensional output space), and finally the one-dimensional output layer (the 'dependent variable').

```
# Create the model
# model specification
```

```
input <- layer_input_from_dataset(train_df %>% select(-label))

output <- input %>%
  layer_dense_features(dense_features(spec)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1)

model <- keras_model(input, output)
```

In order to fit the model, we first have to compile it (configure it for training). At this step we set the configuration parameters that will guide the training/optimization procedure. We use the mean squared errors loss function (`mse`) typically used for regressions. We chose the RMSProp[11] optimizer to find the minimum loss.

```
# compile the model
model %>%
  compile(
    loss = "mse",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_absolute_error")
  )
```

Now we can get a summary of the model we are about to fit to the data.

```
# get a summary of the model
model
```

```
## Model
## Model: "model"
## _________________________________________________________
## Layer (type)      Output Shap Param Connected to
## =========================================================
```

[11] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

```
## AGE (InputLayer)  [(None,)]   0
## _________________________________________________________
## B (InputLayer)    [(None,)]   0
## _________________________________________________________
## CHAS (InputLayer) [(None,)]   0
## _________________________________________________________
## CRIM (InputLayer) [(None,)]   0
## _________________________________________________________
## DIS (InputLayer)  [(None,)]   0
## _________________________________________________________
## INDUS (InputLayer [(None,)]   0
## _________________________________________________________
## LSTAT (InputLayer [(None,)]   0
## _________________________________________________________
## NOX (InputLayer)  [(None,)]   0
## _________________________________________________________
## PTRATIO (InputLay [(None,)]   0
## _________________________________________________________
## RAD (InputLayer)  [(None,)]   0
## _________________________________________________________
## RM (InputLayer)   [(None,)]   0
## _________________________________________________________
## TAX (InputLayer)  [(None,)]   0
## _________________________________________________________
## ZN (InputLayer)   [(None,)]   0
## _________________________________________________________
## dense_features_1  (None, 13)  0     AGE[0][0]
##                                     B[0][0]
##                                     CHAS[0][0]
##                                     CRIM[0][0]
##                                     DIS[0][0]
##                                     INDUS[0][0]
##                                     LSTAT[0][0]
##                                     NOX[0][0]
##                                     PTRATIO[0][0]
##                                     RAD[0][0]
##                                     RM[0][0]
```

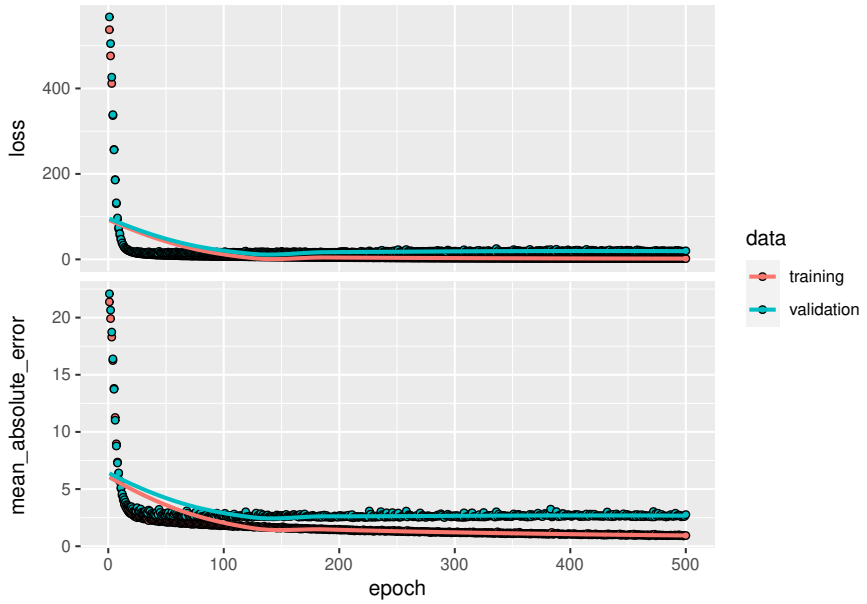
```
##                                     TAX[0][0]
##                                     ZN[0][0]
## ____________________________________________________
## dense_2 (Dense)   (None, 64)  896   dense_features_1[0]
## ____________________________________________________
## dense_1 (Dense)   (None, 64)  4160  dense_2[0][0]
## ____________________________________________________
## dense (Dense)     (None, 1)   65    dense_1[0][0]
## ====================================================
## Total params: 5,121
## Trainable params: 5,121
## Non-trainable params: 0
## ____________________________________________________
```

Given the relatively simple model and small dataset, we set the maximum number of epochs to 500 and allow for early stopping in case the validation loss (based on test data) is not improving for a while.

```
# Set max. number of epochs
epochs <- 500
```

Finally, we fit the model while preserving the training history, and visualize the training progress.

```
# Fit the model and store training stats

history <- model %>% fit(
  x = train_df %>% select(-label),
  y = train_df$label,
  epochs = epochs,
  validation_split = 0.2,
  verbose = 0
)


plot(history)
```



10.3 A word of caution

From just comparing the number of threads of a modern CPU with the number of threads of a modern GPU, one might get the impression that parallel tasks should always be implemented for GPU computing. However, whether one approach or the other is faster can depend a lot on the overall task and the data at hand. Moreover, the parallel implementation of tasks can be done more or less well on either system. Really efficient parallel implementation of tasks can take a lot of coding time (particularly when done for GPUs).[12].

[12] For a more detailed discussion of the relevant factors for well done parallelization (either on CPUs or GPUs), see Matloff (2015)

11

Applied Econometrics with Apache Spark

11.1 Spark basics

Building on the MapReduce model discussed in the previous lecture, Apache Spark[1] is a cluster computing platform particularly made for data analytics. From the technical perspective (in very simple terms), Spark improves some shortcomings of the older Hadoop[2] platform, further improving efficiency/speed. More importantly for our purposes, in contrast to Hadoop, Spark is specifically made for analytics tasks and therefore more easily accessible for people with an applied econometrics background but no substantial knowledge in MapReduce cluster computing. In particular, it comes with several high-level operators that make it rather easy to implement analytics tasks. Moreover (in contrast to basic Hadoop), its very easy to use interactively from R, Python, Scala, and SQL. This makes the platform much more accessible and worth the while for empirical economic research, even for relatively simple econometric analyses.

The following figure illustrates the basic components of Spark. The main functionality, including memory management, task scheduling, and the implementation of Spark's capabilities to handle and manipulate data distributed across many nodes in parallel. Several built-in libraries extend the core implementation, covering specific domains of practical data analytics tasks (querying structured data via SQL, processing streams of data, machine learning, and network/graph analysis). The latter two provide various common functions/algorithms frequently used in data analytics/applied econometrics, such as general-

[1] https://spark.apache.org/
[2] https://hadoop.apache.org/

ized linear regression, summary statistics, and principal component analysis.

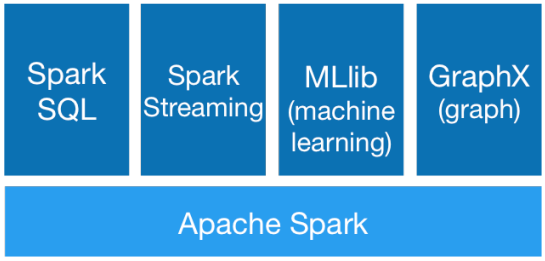


FIGURE 11.1: Basic Spark stack (source: https://spark.apache.org/images/spark-stack.png)

At the heart of big data analytics with Spark is the fundamental data structure called 'resilient distributed dataset' (RDD). When loading/importing data into Spark, the data is automatically distributed across the cluster in RDDs (~ as distributed collections of elements) and manipulations are then executed in parallel in these RDDs. However, the entire Spark framework also works locally on a simple laptop or desktop computer. This is of great advantage when learning Spark and testing/debugging an analytics script on a small sample of the real dataset.

11.2 Spark in R

There are two prominent packages to use Spark in connection to R: `SparkR` and RStudio's `sparklyr`, the former is in some ways closer to Spark's Python API, the latter is closer to the `dplyr`-type of data handling (and is 'compatible' with the '`tidyverse`').[3] For the very simple introductory examples below, either package could have been used equally well. For the general introduction we focus on `SparkR` and later have a look at a simple regression example based on `sparklyr`.

To install and use Spark from the R shell, only a few preparatory steps

[3] See this blog post[4] for a more detailed comparison and discussion of advantages of either package.

are needed. The following examples are based on installing/running Spark on a Linux machine with the `SparkR` package. `SparkR` depends on Java (version 8). Thus, we first should make sure the right Java version is installed. If several Java versions are installed, we might have to select version 8 manually via the following terminal command (Linux).

With the right version of Java running, we can install `SparkR` as usual with `install.packages("SparkR")`. After installing `SparkR`, the call `SparkR::install.spark()` will download and install Apache Spark to a local directory. No we can start an interactive Spark session from within R.

```
# install.packages("SparkR")
# or, if temporarily not available on CRAN:
#if (!require('devtools')) install.packages('devtools')
#devtools::install_github('apache/spark@v2.x.x', subdir='R/pkg') # replace x.x with the versio

# load packages
library(SparkR)

# start session
sparkR.session()
```

By default this starts a local standalone session (no connection to a cluster computer needed). While the examples below are all intended to run on a local machine, it is straightforward to connect to a remote Spark cluster and run the same examples there.[5]

11.2.1 Data import and summary statistics

First, we want to have a brief look at how to perform the first few steps of a typical econometric analysis: import data and compute summary statistics. We analyze the already familiar `flights.csv` dataset. The basic Spark installation provides direct support to import common data formats such as CSV and JSON via the `read.df()` function

[5] Simply set the `master` argument of `sparkR.session()` to the URL of the Spark master node of the remote cluster. Importantly, the local Spark and Hadoop versions should match the corresponding versions on the remote cluster.

(for many additional formats, specific Spark libraries are available). To import `flights.csv`, we set the `source`-argument to `"csv"`.

```
# Import data and create a SparkDataFrame (a distributed collection of data, RDD)
flights <- read.df(path="data/flights.csv", source = "csv", header="true")


# inspect the object
class(flights)
```

```
## [1] "SparkDataFrame"
## attr(,"package")
## [1] "SparkR"
```

```
head(flights)
```

```
##   year month day dep_time sched_dep_time dep_delay
## 1 2013     1   1      517            515         2
## 2 2013     1   1      533            529         4
## 3 2013     1   1      542            540         2
## 4 2013     1   1      544            545        -1
## 5 2013     1   1      554            600        -6
## 6 2013     1   1      554            558        -4
##   arr_time sched_arr_time arr_delay carrier flight
## 1      830            819        11      UA   1545
## 2      850            830        20      UA   1714
## 3      923            850        33      AA   1141
## 4     1004           1022       -18      B6    725
## 5      812            837       -25      DL    461
## 6      740            728        12      UA   1696
##   tailnum origin dest air_time distance hour minute
## 1  N14228    EWR  IAH      227     1400    5     15
## 2  N24211    LGA  IAH      227     1416    5     29
## 3  N619AA    JFK  MIA      160     1089    5     40
## 4  N804JB    JFK  BQN      183     1576    5     45
## 5  N668DN    LGA  ATL      116      762    6      0
## 6  N39463    EWR  ORD      150      719    5     58
```

```
##              time_hour
## 1 2013-01-01T10:00:00Z
## 2 2013-01-01T10:00:00Z
## 3 2013-01-01T10:00:00Z
## 4 2013-01-01T10:00:00Z
## 5 2013-01-01T11:00:00Z
## 6 2013-01-01T10:00:00Z
```

By default, all variables have been imported as type `character`. For several variables this is, of course, not the optimal data type to compute summary statistics. We thus first have to convert some columns to other data types with the `cast` function.

```
flights$dep_delay <- cast(flights$dep_delay, "double")
flights$dep_time <- cast(flights$dep_time, "double")
flights$arr_time <- cast(flights$arr_time, "double")
flights$arr_delay <- cast(flights$arr_delay, "double")
flights$air_time <- cast(flights$air_time, "double")
flights$distance <- cast(flights$distance, "double")
```

Suppose we only want to compute average arrival delays per carrier for flights with a distance over 1000 miles. Variable selection and filtering of observations is implemented in `select()` and `filter()` (as in the `dplyr` package).

```
# filter
long_flights <- select(flights, "carrier", "year", "arr_delay", "distance")
long_flights <- filter(long_flights, long_flights$distance >= 1000)
head(long_flights)
```

```
##   carrier year arr_delay distance
## 1      UA 2013        11     1400
## 2      UA 2013        20     1416
## 3      AA 2013        33     1089
## 4      B6 2013       -18     1576
## 5      B6 2013        19     1065
## 6      B6 2013        -2     1028
```

Now we summarize the arrival delays for the subset of long flights by carrier. This is the 'split-apply-combine' approach applied in `SparkR`.

```
# aggregation: mean delay per carrier
long_flights_delays<- summarize(groupBy(long_flights, long_flights$carrier),
                      avg_delay = mean(long_flights$arr_delay))
head(long_flights_delays)
```

```
##   carrier avg_delay
## 1      UA    3.2622
## 2      AA    0.4958
## 3      EV   15.6876
## 4      B6    9.0364
## 5      DL   -0.2394
## 6      OO   -2.0000
```

Finally, we want to convert the result back into a usual `data.frame` (loaded in our current R session) in order to further process the summary statistics (output to LaTex table, plot, etc.). Note that as in the previous aggregation exercises with the `ff` package, the computed summary statistics (in the form of a table/df) are obviously much smaller than the raw data. However, note that converting a `SparkDataFrame` back into a native R object generally means all the data stored in the RDDs constituting the `SparkDataFrame` object are loaded into local RAM. Hence, when working with actual big data on a Spark cluster, this type of operation can quickly overflow local RAM.

```
# Convert result back into native R object
delays <- collect(long_flights_delays)
class(delays)
```

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

```
delays
```

```
##    carrier avg_delay
## 1       UA    3.2622
```

```
## 2       AA    0.4958
## 3       EV   15.6876
## 4       B6    9.0364
## 5       DL   -0.2394
## 6       OO   -2.0000
## 7       F9   21.9207
## 8       US    0.5567
## 9       MQ    8.2331
## 10      HA   -6.9152
## 11      AS   -9.9309
## 12      VX    1.7645
## 13      WN    9.0842
## 14      9E    6.6730
```

11.2.2 Regression analysis with `sparklyr`

Suppose we want to conduct a correlation study of what factors are associated with more or less arrival delay. Spark provides via its built-in 'MLib' library several high-level functions to conduct regression analyses. When calling these functions via `sparklyr` (or `SparkR`), their usage is actually very similar to the usual R packages/functions commonly used to run regressions in R.

As a simple point of reference, we first estimate a linear model with the usual R approach (all computed in the R environment). First, we load the data as a common `data.table`. We could also convert a copy of the entire `SparkDataFrame` object to a `data.frame` or `data.table` and get essentially the same outcome. However, collecting the data from the RDD structure would take much longer than parsing the csv with `fread`. In addition, we only import the first 300 rows. Running regression analysis with relatively large datasets in Spark on a small local machine might fail or be rather slow.[6]

[6] Again, it is important to keep in mind that running Spark on a small local machine is only optimal for learning and testing code (based on relatively small samples). The whole framework is not optimized to be run on a small machine but for cluster computers.

```
# flights_r <- collect(flights) # very slow!
flights_r <- data.table::fread("data/flights.csv", nrows = 300)
```

Now we run a simple linear regression (OLS) and show the summary output.

```
# specify the linear model
model1 <- arr_delay ~ dep_delay + distance
# fit the model with ols
fit1 <- lm(model1, flights_r)
# compute t-tests etc.
summary(fit1)
```

```
##
## Call:
## lm(formula = model1, data = flights_r)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -42.39  -9.96  -1.91   9.87  48.02
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.182662   1.676560   -0.11     0.91
## dep_delay    0.989553   0.017282   57.26   <2e-16 ***
## distance     0.000114   0.001239    0.09     0.93
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.5 on 297 degrees of freedom
## Multiple R-squared:  0.917,  Adjusted R-squared:  0.917
## F-statistic: 1.65e+03 on 2 and 297 DF,  p-value: <2e-16
```

Now we aim to compute essentially the same model estimate in `spark-`

lyr.[7] In order to use Spark via the sparklyr package we need to first load the package and establish a connection with Spark (similar to SparkR::sparkR.session()).

```
library(sparklyr)

# connect with default configuration
sc <- spark_connect(master = "local",
                    version = "2.4.5")
```

We then copy the data.table flights_r (previously loaded into our R session) to Spark. Again, working on a normal laptop this seems trivial, but the exact same command would allow us (when connected with Spark on a cluster computer in the cloud) to properly load and distribute the data.table on the cluster. Finally, we then fit the model with ml_linear_regression() and compute

```
# load data to spark
flights3 <- copy_to(sc, flights_r, "flights3")
# fit the model
fit1_spark <- ml_linear_regression(flights3, formula = model1)
# compute summary stats
summary(fit1_spark)
```

```
## Deviance Residuals:
##    Min     1Q Median     3Q    Max
## -42.39  -9.96  -1.91   9.87  48.02
##
## Coefficients:
## (Intercept)   dep_delay    distance
##   -0.182662    0.989553    0.000114
##
## R-Squared: 0.9172
## Root Mean Squared Error: 15.42
```

[7] Most regression models commonly used in traditional applied econometrics are in some form provided in sparklyr or SparkR. See the package documentations for more details.

Alternatively, we can do essentially the same with the `SparkR` package:

```
# create SparkDataFrame
flights3 <- createDataFrame(flights_r)
# fit the model
fit2_spark <- spark.glm( formula = model1, data = flights3 , family="gaussian")
# compute t-tests etc.
summary(fit2_spark)
```

Appendix A

.1 GitHub

.1.1 Initiate a new repository

1. Log in to your GitHub account and click on the plus-sign in the upper right corner. From the drop-down-menu select `New repository`.
2. Give your repository a name, for example `bigdatastat`. Then, click on the big green button `Create repository`. You have just created a new repository.
3. Open Rstudio and and navigate to a place on your hard-disk where you want to have the local copy of your repository.
4. Then create the local repository as suggested by GitHub (see the page shown right after you have clicked on `Create repository`: "...or create a new repository on the command line"). In order to do so, you have to switch to the Terminal window in RStudio and type (or copy paste) the commands as given by GitHub. This should look similar to

```
echo "# bigdatastat" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/umatter/bigdatastat.git
git push -u origin master
```

5. Refresh the page of your newly created GitHub repository. You should now see the result of your first commit.
6. Open `README.md` in RStudio and add a few words describing what this repository is all about.

.1.2 Clone this course's repository

1. In RStudio, navigate to a folder on your hard-disk where you want to have a local copy of this course's GitHub repository.
2. Open a new browser window and go to `www.github.com/umatter/BigData`[8].
3. Click on `Clone or download` and copy the link.
4. In RStudio, switch to the Terminal, and type the following command (pasting the copied link).

```
git clone https://github.com/umatter/BigData.git
```

You have now a local copy of the repository which is linked to the one on GitHub. You can see this by changing to the newly created directory, containing the local copy of the repository:

```
cd BigData
```

Whenever there are some updates to the course's repository on GitHub, you can update your local copy with:

```
git pull
```

(Make sure you are in the `BigData` folder when running `git pull`.)

.1.3 Fork this course's repository

1. Go to `https://github.com/umatter/BigData`[9], click on the 'Fork' button in the upper-right corner (follow the instructions).
2. Clone the forked repository (see the cloning of a repository above for details). Assuming you called your forked repository `BigData-forked`, you run the following command in the terminal (replacing `<yourgithubusername>`:

[8] www.github.com/umatter/BigData
[9] https://github.com/umatter/BigData

```
git clone https://github.com/`<yourgithubusername>`/BigData-
forked.git
```

3. Switch into the newly created directory:

```
cd BigData-forked
```

4. Set a remote connection to the *original* repository

```
git remote add upstream https://github.com/umatter/BigData.git
```

You can verify the remotes of your local clone of your forked repository as follows

```
git remote -v
```

You should see something like

```
origin https://github.com/<yourgithubusername>/BigData-
forked.git (fetch)
origin https://github.com/<yourgithubusername>/BigData-
forked.git (push)
upstream https://github.com/umatter/BigData.git (fetch)
upstream https://github.com/umatter/BigData.git (push)
```

5. Fetch changes from the original repository. New material has been added to the original course repository and you want to merge it with your forked repository. In order to do so, you first fetch the changes from the original repository:

```
git fetch upstream
```

6. Make sure you are on the master branch of your local repository:

```
git checkout master
```

7. Merge the changes fetched from the original repo with the master of your (local clone of the) forked repo.

```
git merge upstream/master
```

8. Push the changes to your forked repository on GitHub.

```
git push
```

Now your forked repo on GitHub also contains the commits (changes) in the original repository. If you make changes to the files in your forked repo. you can add, commit, and push them as in any repository. Example: open `README.md` in a text editor (e.g. RStudio), add `# HELLO WORLD` to the last line of `README.md`, and save the changes. Then:

```
git add README.md
git commit -m "hello world"
git push
```

Appendix B

Bibliography

Dhillon, P., Lu, Y., Foster, D. P., and Ungar, L. (2013). New subsampling algorithms for fast least squares regression. In *Advances in Neural Information Processing Systems 26*, pages 360–368.

Fatahalian, K., Sugerman, J., and Hanrahan, P. (2004). Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, page 133–137, New York, NY, USA. Association for Computing Machinery.

Karami, A., Gangopadhyay, A., Zhou, B., and Kharrazi, H. (2017). Fuzzy approach topic discovery in health and medical corpora. *International Journal of Fuzzy Systems*, 20:1334–1345.

Matloff, N. (2015). *Parallel Computing for Data Science*. CRC Press, Boca Raton, FL.

Murrell, P. (2009). *Introduction to Data Technologies*. CRC Press, London, UK.

Walkowiak, S. (2016). *Big Data Analytics with R*. PACKT Publishing, Birmingham, UK.

Wickham, H. (2011). The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40.