Ulrich Matter

Big Data Analytics

A guide to data science practitioners making the transition to Big Data

To the students who have motivated and inspired this work.

Contents

List of Figures

List of Tables

Preface

This textbook provides an introduction to Big Data Analytics 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, analysis).

Prerequisites

This book builds extensively on programming in R. The reader is expected to already be familiar with R and basic programming concepts such as loops, control statements and functions. In addition, the book presupposes some familiarity with undergraduate and basic graduate statistics/econometrics.

FIGURE 1: Creative Commons License

Ulrich Matter St. Gallen, Switzerland

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

Part I

Setting the Scene: Analyzing Big Data

1

Introduction

Over the last decade, 'Big Data' has often been discussed as the new 'most valuable' resource in highly developed economies, driving the development of new products and services in various industries. Extracting knowledge from large data sets is increasingly seen as a strategic asset for firms, governments, and NGOs. In a similar vein, the increasing size of data sets in empirical economic research (both in number of observations and number of variables) offers new opportunities and poses new challenges for economists and business leaders.

Successfully navigating the data-driven economy presupposes a certain understanding of the technologies and methods to gain insights from Big Data. This textbook introduces the reader to the basic concepts of Big Data Analytics to gain insights from large and complex data sets. Thereby, the focus of the book is on the practical application of econometrics, given large/complex data sets, and all the steps involved before actually analyzing data (data storage, data import, data preparation). The book combines conceptual/theoretical material with the practical application of the concepts with R and SQL. Thereby, the reader will acquire the basic skill set to analyze large data sets both locally and in the cloud. Various code examples and tutorials, focused on empirical economic and business research, illustrate practical techniques to handle and analyze Big Data.

1.1 What is *big* in "Big Data"?

Generally, we can think of Big Data as data that is (a) expensive to handle and (b) hard to get value from due to its size and complexity. The handling of Big Data is expensive as the data is often gath-

ered from unorthodox sources, providing poorly structured data (e.g., raw text, web pages, images, etc.) as well as because of the infrastructure needed to store and load/process large amounts of data. Getting value/insights from Big Data is related to two distinct properties that render the analysis of large amounts of data difficult:

- The *big P* problem: a data set has more variables than observations, which renders the search for a good predictive model with traditional econometric techniques difficult or illusive. For example, suppose you run an e-commerce business that sells hundreds of thousands of products to tens of thousands of customers. You want to figure out from which product category a customer is most likely to buy an item from, based on her previous product page visits. That is, you want to (in simple terms) regress an indicator of purchasing from a specific category on indicators for previous product page visits. Given this set up, you would potentially end up with hundreds of thousands of explanatory indicator variables (and potentially even linear combinations of those), while you "only" have tens of thousands of observations (one per user/customer) to estimate your model. These sorts of problems are at the core of the domain of modern predictive econometrics, which shows how machine learning approaches like the LASSO can be applied in order to get reasonable estimates of such a predictive model.
- The *big N* problem: a data set has massive amounts of observations (rows) such that it cannot be handled with standard data analytics techniques and/or on a standard desktop computer. For example, suppose you want to segment your e-commerce customers based on the traces they leave on your website's server. Specifically, you plan to use the server log files (when does a customer visit the site from where, etc.) in combination with purchase records as well as written product reviews by the users. You focus on 50 variables that you measure on a daily basis over five years for all users. The resulting data set has $50,000 \times 365 \times 5 = 91,250,000$ rows and with 50 variables (50 columns) over 4.5 billion cells. Such a data set can easily take up dozens of Gigabytes on the hard disk. Hence it will either not fit into the memory of a standard computer to begin with (import fails), or the standard programs to process and analyze the

data will likely be very inefficient and take ages to finish when used on such a large data set. There are both econometric techniques as well as various specialized software and hardware tools to handle such a situation.

While covering some aspects of both problem domains, most of this book focuses on practical challenges and solutions related to the *big N* problem in Big Data Analytics.

1.2 Approaches to analyzing Big Data

Throughout the book, we consider four approaches on how to solve challenges related to analyzing Big Data. Those approaches should not be understood as mutually exclusive categories for Big Data tools, rather they should help us to look at a specific problem from different angles in order to find the most efficient tool/approach to proceed.

1. *Statistics/econometrics and machine learning*: During the initial hype surrounding Big Data/Data Science about a decade ago, statisticians prominently (and justifiably) pointed out that statistics has always been a very useful tool when analyzing "all the data" (the entire population) is too costly.[1] In simple terms, when confronted with the challenge of answering an empirical question based on a *big N* data set (which is too large to process on a normal computer), one might ask "why not simply take a random sample"? In some situations this might actually be a very reasonable question, and we should be sure to have a good answer for it before we rent a cluster computer with specialized software for distributed computing. After all, statistical inference is there to help us answer empirical questions in situations where collecting data on the entire population would be practically impossible or simply way too costly. In today's world, digital data is abun-

[1] David Donoho has nicely summarized this critique in a paper titled "50 Years of Data Science"[2] (Donoho (2017)), which I warmly recommend.

dant in many domains and the collection is not so much the problem anymore but our standard data analytics tools are not made to analyze such amounts of data. Depending on the question and data at hand, it might thus make sense to simply use well-established "traditional" statistics/econometrics in order to properly address the empirical question. Note, though, that there are also various situations in which this would not work well. For example, consider online advertising. If you want to figure out which user characteristics make a user significantly more likely to click on a specific type of ad, you likely need hundreds of millions of data points because the expected probability that a specific user clicks on an ad is likely generally very low. That is, in many practical big data analytics settings you might expect rather small effects. Consequently, you need to rely on a big-N data set in order to get the statistical power to distinguish an actual effect from a zero effect. However, even then, it might make sense to first look at newer statistical procedures that are specifically made for big-N data before renting a cluster computer. Similarly, traditional statistical/econometric approaches might help to deal with big-p data, but they are usually rather inefficient or have rather problematic statistical properties in such a situation. However, there are also well-established machine learning approaches to better address these problems. In sum, before focusing on specialized software like Hadoop and scaling up hardware resources, make sure to use the adequate statistical tools for a big-data situation. This can save a lot of time and money. Once you have found the most efficient statistical procedure for the problem at hand, you can focus on how to compute it.

2. *Writing efficient code*: no matter how suitable a statistical procedure theoretically is to analyze a large data set, there are always various ways of how this procedure can be implemented in software. Some ways will be less efficient than others. When working with small or moderately sized data sets you might not even notice whether your data analytics script is written in an efficient way. However, it might get uncom-

fortable to run your script once you confront it with a large data set. Hence the question you should ask yourself when taking this perspective is, "can I write this script in a different way to make it faster (but achieve the same result)?" Before introducing you to specialized R-packages to work with large data sets, we thus look at a few important aspects of how to write efficient/fast code in R.

3. *Use limited local computing resources more efficiently*: there are several strategies to use the available local computing resources (your PC) more efficiently, and many of those have been around for a while. In simple terms, these strategies are based on the idea of more explicitly telling the computer how to allocate and use the available hardware resources as part of a data analytics task (something that is usually automatically taken care of by the PC's operating system). We will touch upon several of these strategies, such as multi-core processing and the efficient use of virtual memory and then practically implement these strategies with the help of specialized R packages. Unlike writing more efficient R code, these packages/strategies usually come with an overhead. That is, they help you save time only after a certain threshold. In other words, not using these approaches can be faster if the data set is not "too big". In addition, there can be trade-offs between using one vs the other hardware component more efficiently. Hence, using these strategies can be tricky and the best approach might well depend on the specific situation. The aim is thus to make you comfortable with answering the question "how can I use my local computing environment more efficiently to further speed up this specific analytics task"?

4. *Scale up and scale out*: once you have properly considered all of the above, but the task still cannot be done in a reasonable amount of time, you will need to either *scale up* or *scale out* the available computing resources. *Scaling up* refers to enlarging your machine (e.g., add more random access memory) or to switching to a more powerful machine altogether. Techni-

cally, this can mean literally building an additional hardware-device into your PC, today it usually means renting a virtual server in the cloud. Instead of using a "bigger machine", *scaling out* means using several machines in concert (cluster computer, distributed systems). While this also has often been done locally (connecting several PCs to a cluster of PCs in order to combine all their computing power), today this too is usually done in the cloud (due to the much easier set up and maintenance). Practically, a key difference between scaling out and scaling up is that by-and-large scaling up does not require you to get familiar with specialized software. You can simply run the exact same script you tested locally on a larger machine in the cloud. Although most of the tools and services available to scale out your analyses are by now quite easy to use, you will have to get familiar with some additional software components and programming paradigms to really make use of the latter.[3] In addition, in some situations, scaling up might be perfectly sufficient while in others only scaling out makes sense (particularly if you need massive amounts of memory). In any event, you should be comfortable dealing with the question "does it make sense to scale up or scale out?" and "if yes, how can it be done?" in a given situation.[4]

[3] Not thought, that this aspect of scaling out has recently become much more user-friendly and is likely to get even more so over the next few years. Therefore, we will focus primarily on how to set up such a solution with the help of high-level/easy-to-use interfaces.

[4] Importantly, the perspective on scaling up and scaling out provided in this book is solely focused on Big Data Analytics in the context of economic/business research. There is a large array of practical problems and corresponding solutions/tools to deal with "Big Data Analytics" in the context of application development (e.g. tools related to data streams) which this book does not cover.

1.3 Content overview

The book is organized in three main parts. The first part introduces the reader to the topic of Big Data Analytics from the perspective of a practitioner in empirical economic and business research. It covers the differences between *Big P* and *Big N* problems and shows avenues of how to practically address either.

The second part focuses on the tools and platforms to work with Big Data. Initially, this part introduces a set of software tools primarily used throughout the book: (advanced) R and SQL. It then covers the conceptual basis of modern computing environments and discusses how different hardware components matter in practical local Big Data Analytics as well as how virtual servers in the cloud help to scale up and scale out analyses when the local hardware does not have enough computing resources.

Building on the previous part, the third part of this book visits all: data collection and data storage, data import/ingestion, data cleaning/transformation, data aggregation, and finally, data analysis (with a particular focus on modern applied econometrics) and explorative data visualization (with a particular focus on GIS). The chapters in this part of the book discuss basic concepts such as the split-apply-combine approach and demonstrate the practical application of these concepts when working with large data sets in R. Many tutorials and code examples illustrate how a specific task can be implemented with comparatively simple tools locally as well as with a broader set of tools in the cloud. Thereby, the reader re-visits key concepts regarding computing resources and econometric methods.

The code examples, illustrations, and tutorials provided throughout the book focus on data analytics contexts in empirical economics as well as business data science/business analytics. However, the basic concepts and tools covered in the book are not domain-specific and could easily be transferred to other fields of modern data analytics/data science.

2

Two domains of Big Data Analytics

Data analytics in the context of Big Data can be broadly categorized into two domains: techniques/estimators to address *big P* problems and techniques/estimators to address *big N* problems. While this book predominantly focuses on how to handle Big Data for applied economics and business analytics settings in the context of *big N* problems, it is useful to set the stage for the following chapters with two practical examples concerning *big P* and *big N* methods.

2.1 A practical *big P* problem

Due to the abundance of digital data on all kinds of human activities, both empirical economists and business analysts are increasingly confronted with high-dimensional data (many signals, many variables). While having a lot of variables to work with sounds kind of like a good thing, it introduces new problems in coming up with useful predictive models. In the extreme case of having more variables in the model than observations, traditional methods cannot be used at all. In the less extreme case of just having dozens or hundreds of variables in a model (and plenty of observations), we risk "falsely" discovering seemingly influential variables and consequently coming up with a model with potentially very misleading out-of-sample predictions. So how can we find a reasonable model?[1]

[1] Note that finding a model with good in-sample prediction performance is trivial when you have a lot of variables: simply adding more variables will improve the performance. However, that will inevitably result in a nonsensical model as even highly significant variables might not have any actual predictive power when looking at out-of-sample predictions. Hence, in this kind of exercise we should *exclusively focus on out-of-sample predictions* when assessing the performance of candidate models.

Let us look at a real-life example. Suppose you work for Google's e-commerce platform www.googlemerchandiseshop.com[2], and you are in charge of predicting purchases (i.e., a user is actually buying something from your store in a given session) based on user and browser-session characteristics.[3] The dependent variable `purchase` is an indicator equal to `1` if the corresponding shop visit leads to a purchase and equal to `0` otherwise. All other variables contain information about the user and the session (Where is the user located? Which browser is (s)he using? etc.). As the dependent variable is binary, we will first estimate a simple logit model, in which we use the origins of the store visitors (how did a visitor end up in the shop) as explanatory variables. Note that many of these variables are categorical, and the model matrix thus contains a lot of "dummies" (indicator variables). The plan in this (intentionally naive) first approach is to simply add a lot of explanatory variables to the model, run logit, and then select the variables with statistically significant coefficient estimates as the final predictive model. The following code snippet covers the import of the data, the creation of the model matrix (with all the dummy-variables), as well as the logit estimation.

```
# import/inspect data
ga <- read.csv("data/ga.csv")
head(ga[, c("source", "browser", "city", "purchase")])
```

```
##     source browser          city purchase
## 1   google  Chrome      San Jose        1
## 2 (direct)    Edge     Charlotte        1
## 3 (direct)  Safari San Francisco        1
## 4 (direct)  Safari   Los Angeles        1
## 5 (direct)  Chrome       Chicago        1
## 6 (direct)  Chrome     Sunnyvale        1
```

[2] https://shop.googlemerchandisestore.com

[3] We will in fact be working with a real-life Google Analytics dataset from www.googlemerchandiseshop.com; see here for details about the dataset: https://www.blog.google/products/marketingplatform/analytics/introducing-google-analytics-sample/.

```
# create model matrix (dummy vars)
mm <- cbind(ga$purchase,
            model.matrix(purchase~source, data=ga,)[,-1])
mm_df <- as.data.frame(mm)
# clean variable names
names(mm_df) <- c("purchase",
                  gsub("source", "", names(mm_df)[-1]))

# run logit
model1 <- glm(purchase ~ .,
              data=mm_df, family=binomial)
```

Now we can perform the t-tests and filter out the "relevant" variables.

```
model1_sum <- summary(model1)

# select "significant" variables for final model
pvalues <- model1_sum$coefficients[,"Pr(>|z|)"]
vars <- names(pvalues[which(pvalues<0.05)][-1])
vars
```

```
##  [1] "bing"
##  [2] "dfa"
##  [3] "docs.google.com"
##  [4] "facebook.com"
##  [5] "google"
##  [6] "google.com"
##  [7] "m.facebook.com"
##  [8] "Partners"
##  [9] "quora.com"
## [10] "siliconvalley.about.com"
## [11] "sites.google.com"
## [12] "t.co"
## [13] "youtube.com"
```

Finally, we re-estimate our "final" model

```
# specify and estimate the final model
finalmodel <- glm(purchase ~.,
                  data = mm_df[, c("purchase", vars)],
                  family = binomial)
```

The first problem with this approach is that we should not trust the coefficient t-tests based on which we have selected the covariates too much. The first model contains 62 explanatory variables (plus the intercept). With that many hypothesis tests, we quite likely reject the NULL of no predictive effect although there is actually no predictive effect. In addition, this approach turns out to be unstable. There might be correlation between some variables in the original set of variables, and adding/removing even one variable might substantially affect the predictive power of the model (and the apparent relevance of other variables). We see this already from the summary of our final model estimate (generated in the next code chunk). One of the apparently relevant predictors (`dfa`) is not at all significant anymore in this specification. Thus, we might be tempted to further change the model, which in turn would again change the apparent relevance of other covariates, and so on.

```
summary(finalmodel)$coef[,c("Estimate", "Pr(>|z|)")]
```

```
##                           Estimate   Pr(>|z|)
## (Intercept)               -1.3831  0.000e+00
## bing                      -1.4647  4.416e-03
## dfa                       -0.1865  1.271e-01
## docs.google.com           -2.0181  4.714e-02
## facebook.com              -1.1663  3.873e-04
## google                    -1.0149 6.321e-168
## google.com                -2.9607  3.193e-05
## m.facebook.com            -3.6920  2.331e-04
## Partners                  -4.3747  3.942e-14
## quora.com                 -3.1277  1.869e-03
## siliconvalley.about.com   -2.2456  1.242e-04
## sites.google.com          -0.5968  1.356e-03
```

```
## t.co                    -2.0509  4.316e-03
## youtube.com             -6.9935  4.197e-23
```

An alternative approach would be to estimate models based on all possible combinations of covariates and then use that sequence of models to select the final model based on some out-of-sample prediction performance measure. Clearly such an approach would take a long time to compute.

Instead, the *lasso estimator* provides a convenient and efficient way to get a sequence of candidate models. The key idea behind the lasso is to penalize model complexity (the cause of instability) during the estimation procedure.[4] In a second step, we can then select a final model from the sequence of candidate models based on, for example, "out-of-sample" prediction in a k-fold cross validation.

The `gamlr` package provides both parts of this procedure (lasso for the sequence of candidate models, and selection of the "best" model based on k-fold cross-validation).

```
# load packages
library(gamlr)

# create the model matrix
mm <- model.matrix(purchase~source, data = ga)
```

In cases with both many observations and many candidate explanatory variables, the model matrix might get very large. Even simply generating the model matrix might be a computational burden, as we might run out of memory to hold the model matrix object. If this large model matrix is sparse (i.e, has a lot of `0` entries), there is a much more memory efficient way to store it in an R object. R provides ways to represent such sparse matrices in a compressed way in specialized R objects (such as `CsparseMatrix` provided in the `Matrix` package). Instead of containing all $n \times m$ cells of the matrix, these objects only explicitly store the cells with non-zero values and the corresponding

[4] In simple terms, this is done by adding $\lambda \sum_k |\beta_k|$ as a "cost" to the optimization problem.

index. Below, we make use of the high-level `sparse.model.matrix` function to generate the model matrix and store it in a sparse matrix object. To illustrate the point of a more memory-efficient representation, we show that the traditional matrix object is about 7.5 times larger than the sparse version.

```
# create the sparse model matrix
mm_sparse <- sparse.model.matrix(purchase~source, data = ga)
# the sparse representation needs less than 15% of
# the memory needed for the standard matrix representation:
as.numeric(object.size(mm)/object.size(mm_sparse))
```

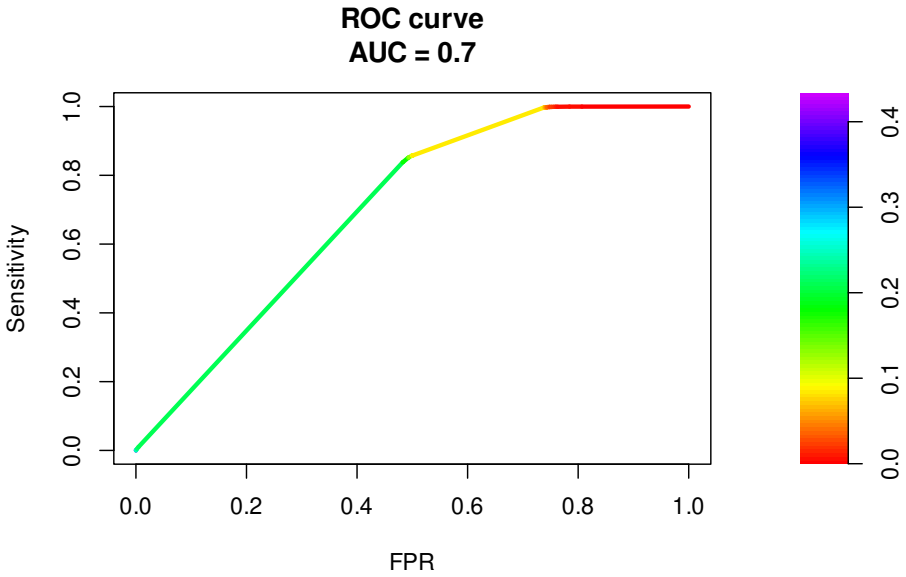
```
## [1] 7.525
```

Finally, we run the lasso estimation with k-fold cross-validation.

```
# run k-fold cross-validation lasso
cvpurchase <- cv.gamlr(mm_sparse, ga$purchase, family="binomial")
```

We then can illustrate the performance of the selected final model, for example, with an ROC curve. Note that both the `coef` method and the `predict` method for `gamlr` objects automatically select the 'best' model.

```
# load packages
library(PRROC)

# use "best" model for prediction
# (model selection based on average OSS deviance, here CV1se rule
pred <- predict(cvpurchase$gamlr, mm_sparse, type="response")

# compute tpr, fpr, plot ROC
comparison <- roc.curve(scores.class0 = pred,
                        weights.class0=ga$purchase,
                        curve=TRUE)
plot(comparison)
```



Hence, econometrics techniques such as the lasso help deal with *big P* problems by providing reasonable ways to select a good predictive model (in other words, decide which of the many variables should be included).

2.2 A practical *big N* problem

Big N problems are situations in which we know what type of model we want to use but the *number of observations* is too big to run the estimation (the computer crashes or slows down significantly). The simplest statistical solution to such a problem is usually to just estimate the model based on a smaller sample. However, we might not want to do that for other reasons (see introduction above). As an illustration of how an alternative statistical procedure can speed up the analysis of big N datasets, we look at a procedure to estimate linear models when the classical OLS estimator is computationally too demanding when analyzing large datasets: The *Uluru* algorithm (Dhillon et al., 2013).

2.2.1 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.[5] If our dataset 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` (the number of observations in our pseudo-sample) and `p` (the number of variables describing each of these observations) and initiate the dataset `X`.

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

[5] 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.

```
# generate sample based on Monte Carlo
# generate a design matrix (~ our 'dataset')
# with 4 variables and 10,000 observations
X <- matrix(rnorm(n*p, mean = 10), ncol = p)
# add column for intercept
X <- cbind(rep(1, n), X)
```

Now we define what the real linear model that we have in mind looks like and compute the output `y` of this model, given the input `X`.[6]

```
# 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,]  1.992
## [2,]  1.500
## [3,]  4.001
## [4,] -3.500
## [5,]  0.500
```

2.2.2 The *Uluru* algorithm as an alternative to OLS

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

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

[6] 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.

, where

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

, and

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

, 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 dataset. However, the remainder of the dataset 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)
     }
```

We then test it with the same input as above:

```
# set size of sub-sample
n_subs <- 1000
# select sub-sample 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,]  2.188
## [2,]  1.489
## [3,]  3.998
## [4,] -3.503
## [5,]  0.498
```

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 sub-samples
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 sub-sample
     n_subs <- n_subs_sizes[i]
     # select sub-sample 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] # (1 is the intercept)
    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]
```

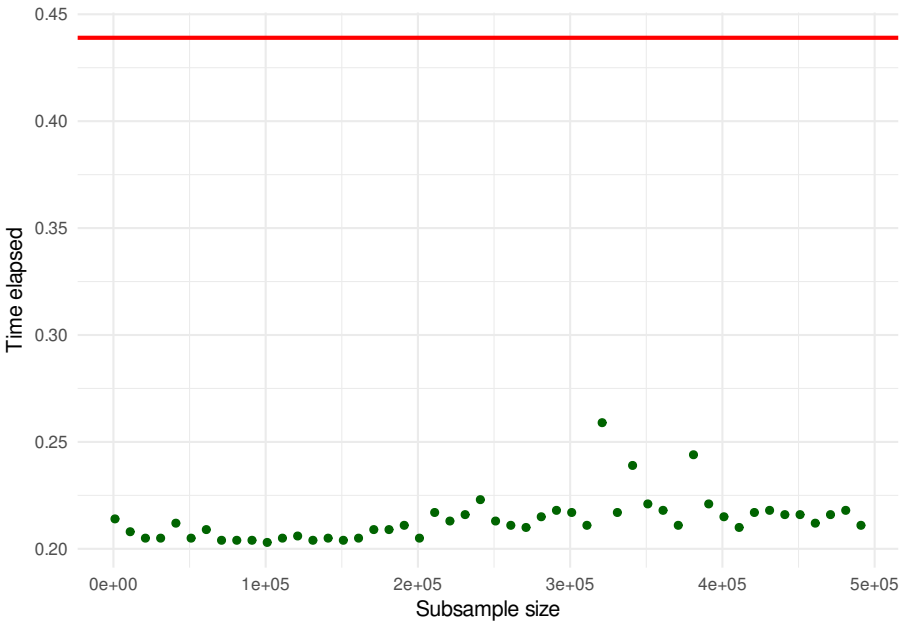
Let's visualize the comparison with OLS.

```
# 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")
```

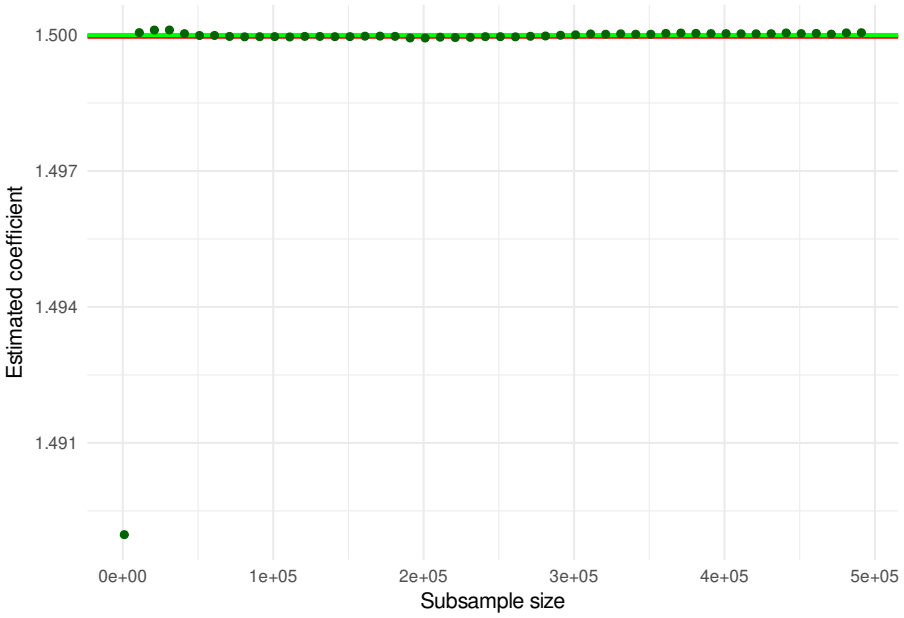


The horizontal red line indicates the computation time for estimation via OLS; the green points indicate the computation time for the estimation via the *Ulruru* algorithm. Note that even for large sub-samples, the computation time is substantially lower than for OLS.

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.

Taken together, the example illustrates that alternative statistical methods, optimized for large amounts of data, can deliver results very close to traditional approaches. Yet, they can deliver these results much more efficiently.

2.3 Conclusion

When applying econometrics to "large" datasets, two types of problems dominate: a) the number of covariates is 'too large' to handle with traditional estimators (*big P* problem) and/or b) the number of observations is 'too large' to handle with traditional estimators (*big N* problem). The first problem is being addressed by an interesting new strain of econometrics literature, which builds on previously known machine learning algorithms such as lasso and random forest and clarifies the statistical properties of predictions coming from these algorithms. While the following chapters will repeatedly link to these contributions in specific code examples, the main focus of the following chapters lies on the *big N* problem and the key underlying concepts (in terms of software, hardware, and faster new estimators) to productively perform data analytics tasks when confronted with large amounts of data. Several of the following chapters will also focus on something that is often ignored in the applied econometrics literature: how do we get from the raw data to the cleaned and filtered analytic dataset to run regressions. While these topics are generally highly relevant for proper data science and data analytics workflows, you will see that these topics are absolutely crucial when working with large datasets.

Part II

Platform: Software and Computing Resources

3

Software: Programming with (Big) Data

The programming language and computing environment *R* is particularly made for writing code in a data analytics context. However, the language was developed at a time when data analytics was primarily focused on moderately sized data sets that can easily be loaded/imported and worked with on a common PC. Depending on the field or industry you work in, this is not anymore the case today. In this chapter, we will explore some of R's (potential) weaknesses as well as learn how to avoid them and how to exploit some of R's strengths when it comes to working with large data sets. The first part of this chapter is primarily focused on understanding code profiling and improving code with the aim of making computationally intense data analytics scripts in R run faster. This chapter presupposes basic knowledge of R data structures and data types as well as experience with basic programming concepts such as loops.[1]

While R is a very useful tool for many aspects of big data analytics that we will cover in the following chapters, R alone is not enough for a basic big data analytics toolbox. The second part of this chapter introduces the reader to the *Structured Query Language (SQL)*, a programming language designed for managing data in relational databases. Although the type of databases where SQL is traditionally encountered would not necessarily be considered part of big data analytics today, some versions of SQL are now used with systems particularly designed for big data analytics (such as Amazon Athena and Google BigQuery). Hence, with a good knowledge of R in combination with basic SQL skills, you will be able to productively engage with a large array of practical big data analytics problems.

[1] Appendix B reviews the most relevant concepts regarding data types and data structures in R.

3.1 Domains of programming with (big) data

Programming tasks in the context of data analytics 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]
2. How can we exploit/avoid some of R's lower-level characteristics in order to write more efficient code?
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 writing of code for *statistical analysis*: Independent of *how* we write a statistical procedure in R (or in any other language, for that matter), keep in mind that there might be an *alternative statistical procedure/algorithm* that is faster but delivers approximately the same result (as long as we use a sufficiently large sample).

[2] Throughout the rest of this book, I will point to specialized R packages and functions that are particularly designed to work with large amounts of data. Where necessary, we will also look more closely at the underlying concepts that explain why these specialized packages work better with large amounts of data than the standard approaches.

3.2 Measuring R performance

When writing a data analysis script in R to process large amounts of data, it generally makes sense to first test each crucial part of the script with a small sub-sample. In order to nevertheless quickly recognize potential bottle necks, there are a couple of R packages that help you keep track of how long exactly each component of your script needs to process as well as how much memory it uses. The table below lists some of the packages and functions that you should keep in mind when "*profiling*" and testing your code.

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`	`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.).

Most of these functions are used in an interactive way in the R console. They serve either of two purposes that are central to profiling and improving your code's performance. First, in order to assess the performance of your R code you probably want to know how long it

takes to run your entire script or a specific part of your script. The `system.time()`-function provides an easy way to check this. This function is loaded by default with R, there is no need to install an additional package. Simply wrap it around the line(s) of code that you want to assess.

```
# how much time does it take to run this loop?
system.time(for (i in 1:100) {i + 5})
```

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

Note that each time you run this line of code, the returned amount of time varies slightly. This has to do with the fact that the actual time needed to run a line of code can depend on various other processes happening at the same time on your computer.

The `microbenchmark` and `bench` packages provide additional functions to measure execution time in more sophisticated ways. In particular, they account for the fact that the processing time for the same code might vary and automatically run the code several times in order to return statistics about the processing time. In addition, `microbenchmark()` provides highly detailed and highly accurate timing of R expression evaluation. The function is particularly useful to accurately find even "minor" room for improvement when testing a data analysis script on a smaller sub-sample (which might scale when working on a large data set). For example, suppose you need to run a for-loop over millions of iterations and there are different ways to implement the loop of the body (which does not take too much time to process in one iteration). Note that the function actually evaluates the R expression in question many times and returns a statistical summary of the timings.

```
# load package
library(microbenchmark)
# how much time does it take to run this loop (exactly)?
microbenchmark(for (i in 1:100) {i + 5})
```

```
## Unit: milliseconds
##                            expr   min    lq mean
##  for (i in 1:100) {     i + 5 } 1.129 1.228 1.36
##  median   uq   max neval
##   1.278 1.35 4.859   100
```

Second, a key aspect to improving the performance of data analysis scripts in R is to detect inefficient memory allocation as well as avoiding that an R-object is either growing too much or is generally too large to handle in memory. To this end, you might want to monitor how much memory R occupies at different points in your script as well as how much memory is taken up by individual R objects. For example, `object.size()` returns the size of an R object, that is the amount of memory it takes up in the R environment in bytes (`pryr::object_size()` counts slightly more accurately).

```
hello <- "Hello, World!"
object.size(hello)
```

```
## 120 bytes
```

This is useful to implementing your script with a generally less memory-intense approach. For example, for a specific task it might not matter whether a particular variable is stored as a `character` vector or a `factor`. But storing it as `character` turns out to be more memory intense (why?).

```
# initiate a large string vector containing letters
large_string <- rep(LETTERS[1:20], 1000^2)
head(large_string)
```

```
## [1] "A" "B" "C" "D" "E" "F"
```

```
# store the same information as a factor in a new variable
large_factor <- as.factor(large_string)
```

```
# is one bigger than the other?
object.size(large_string) - object.size(large_factor)
```

```
## 79999456 bytes
```

`pryr::mem_change()` is useful to track how different parts of your script affect the overall memory occupied by R.

```
# load package
library(pryr)

# initiate a vector with 1000 (pseudo)-random numbers
mem_change(
        thousand_numbers <- runif(1000)
        )
```

```
## 9.64 kB
```

```
# initiate a vector with 1M (pseudo)-random numbers
mem_change(
        a_million_numbers <- runif(1000^2)
        )
```

```
## 8 MB
```

`bench::mark()` allows you to easily compare the performance of several different implementations of a code chunk both regarding timing and memory usage. The following code example illustrates this in a comparison of two approaches to computing the product of each element in a vector `x` with a factor `z`.

```
# load packages
library(bench)

# initiate variables
x <- 1:10000
```

```
z <- 1.5
# approach 1: loop
multiplication <-
        function(x,z) {
                result <- c()
                for (i in 1:length(x)) {result <- c(result, x[i]*z)}
                return(result)
        }
result <- multiplication(x,z)
head(result)
```

```
## [1] 1.5 3.0 4.5 6.0 7.5 9.0
```

```
# approach II: "R-style"
result2 <- x * z
head(result2)
```

```
## [1] 1.5 3.0 4.5 6.0 7.5 9.0
```

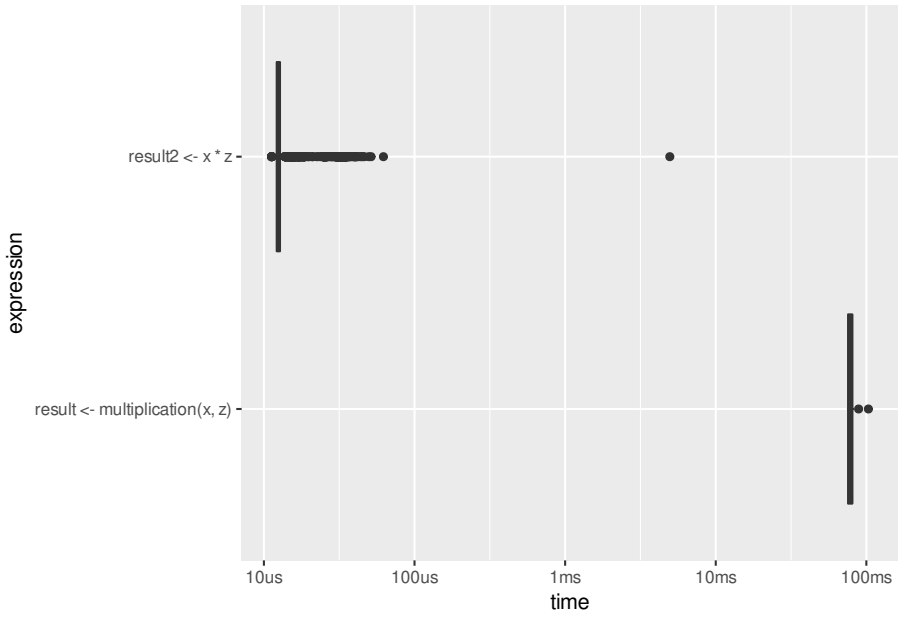
```
# comparison
benchmarking <-
        mark(
        result <- multiplication(x,z),
        result2 <- x * z,
        min_iterations = 50
)
benchmarking[, 4:9]
```

```
## # A tibble: 2 x 3
##   `itr/sec` mem_alloc `gc/sec`
##       <dbl> <bch:byt>    <dbl>
## 1      13.1     382MB    11.2
## 2   75944.     78.2KB     7.60
```

In addition, the bench package provides a simple way to visualize these outputs:

```
plot(benchmarking, type = "boxplot")
```

```
## Loading required namespace: tidyr
```



Finally, to analyze the performance of your entire script/program the `profvis` package provides visual summaries to quickly detect the most prominent bottle necks. You can either call this via the `profvis()` function with the code section to be profiled as argument, or via the RStudio user interface by clicking on the Code Tools menu in the editor window and select "Profile selected lines".

```
# load package
library(profvis)

# analyse performance of several lines of code
profvis({
        x <- 1:10000
        z <- 1.5
        # approach 1: loop
multiplication <-
```

```
        function(x,z) {
                result <- c()
                for (i in 1:length(x)) {result <- c(result, x[i]*z)}
                return(result)
        }
result <- multiplication(x,z)

# approach II: "R-style"
result2 <- x * z
head(result2)
})
```

3.3 Writing efficient R code

This subsection touches upon several prominent aspects of writing efficient/fast R code.[3]

3.3.1 Memory allocation and growing objects

R tends to "grow" already initiated objects in memory when they are modified. At the initiation of the object a small amount of memory is occupied at some location in memory. In simple terms, once the object grows, it might not have enough space where it is currently located. Hence, it needs to be "moved" to another location in memory with more space available. This moving, or "re-allocation" of memory, needs time and slows down the overall process.

This potential is most practically illustrated with a `for`-loop in which each iteration's result is stored as an element of a vector (the object in question). To avoid growing this object, you need to instruct R to pre-allocate the memory necessary to contain the final result. If we don't do

[3] This is not intended to be a definitive guide to writing efficient R code in every aspect. Instead the subsection aims at covering most of the typical pitfalls to avoid and to provide an easy-to-remember number of tricks to keep in mind when writing R code for computationally intense tasks.

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 functions take a numeric vector as input and return 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)
     }
```

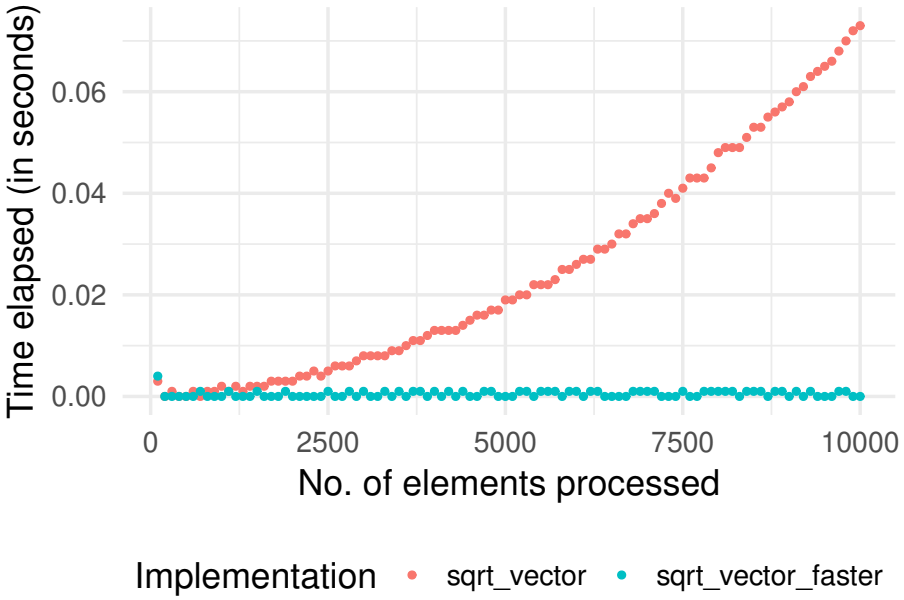
As a proof of concept we use `system.time()` to measure the difference in speed for various input sizes.[4]

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

```
# 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)

# compute outputs 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) +
     theme(legend.position = "bottom") +
     ylab("Time elapsed (in seconds)") +
     xlab("No. of elements processed")
```



Clearly, the version with pre-allocation of memory (avoiding growing an object) is overall much faster. In addition, we see that the problem with the growing object in the naive implementation tends to get worse with each iteration. The take-away message for the practitioner: if possible, always initiate the "container" object (list, matrix, etc.) for iteration results as an empty object of the final size/dimensions.

The attentive reader and experienced R coder will have noticed by this point, that both of the functions implemented above are not really smart practice to solve the problem at hand. If you consider yourself part of this group the next subsection will make you more comfortable.

3.3.2 Vectorization in basic R functions

We can further improve the performance of this function by exploiting a particular characteristic of R: in R 'everything is a vector' and many of the most 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.

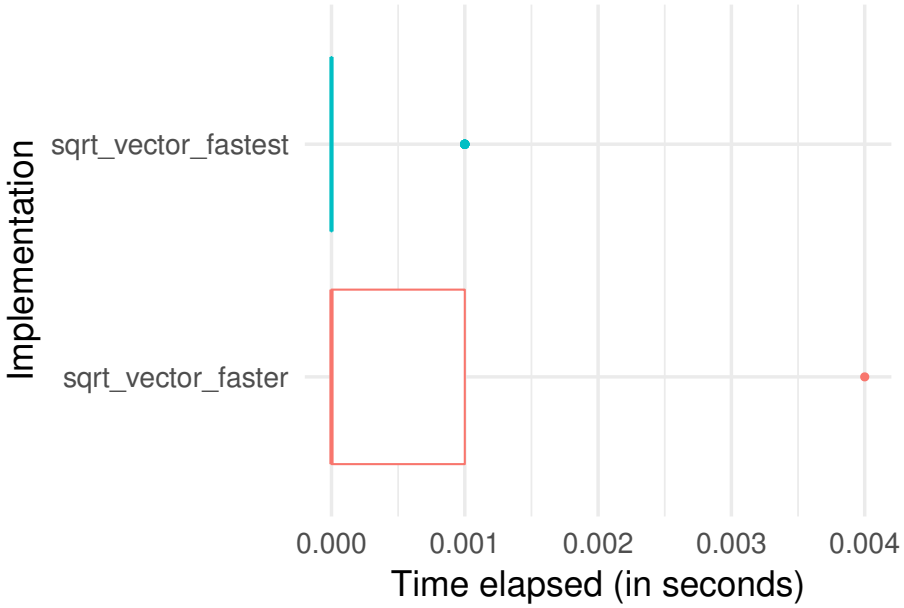
Following up on the problem from the previous subsection, we implement an additional function called `sqrt_vector_fastest` that exploits the fact that math operators in R are vectorized functions. We then re-run the same speed test as above with this function.

```
# 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=time_elapsed, y=Implementation)) +
```

```
    geom_boxplot(aes(colour=Implementation),
                         show.legend = FALSE) +
    theme_minimal(base_size = 18) +
    xlab("Time elapsed (in seconds)")
```



Clearly, the vectorized implementation is even faster. The take-away message: make use of vectorized basic R functions where possible. At this point you might wonder: why not always use vectorization over loops, when working with R? This question (and closely related similar questions) have been fiercely debated in the R online community over the last few years. Also the debate contains and has contained several (in my view) slightly misleading arguments. A simple answer to this question is: it is in fact not that simple to use *actual* vectorization for every kind of problem in r. There are a number of functions often mentioned to achieve "vectorization" easily in R, however, they do not actually implement actual vectorization in its original technical sense (the type just demonstrated here with the R math operators). Since this point is very prominent in debates about how to improve R code, the next subsection attempts to summarize the most important aspects to keep in mind.

3.3.3 `apply`-type functions and vectorization

There are basically two ways to make use of some form of "vectorization" instead of writing loops.

One approach is to use an `apply`-type function instead of loops. Note though, that the `apply`-type functions primarily make the writing of code more efficient. They still run a loop under the hood. Nevertheless, some `apply`-type functions might still outperform an explicit loops as they are might be better implemented.[5]

Consider, for example, `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 (and it automatically takes care of the "growing objects" problem discussed above). 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[6] 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.

```
# 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.

[5] If you know how to implement efficient for-loops in R (as you are certainly expected at this point), there is not much to gain from using an `apply`-type function instead of a loop, apart from making your code easier to read (and faster to write).

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

```
# 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.

```
# use lapply instead of 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" our own import function (written for this example). Again, this does not make use of vectorization in its original technical sense.

```
# 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>
```

The take-away message: instead of writing simple loops, use `apply`-type functions to save time writing code (and make the code easier to read) and automatically avoid memory-allocation problems.

3.3.4 Avoid unnecessary copying

The "growing objects" problem discussed above is only one aspect that can lead to inefficient use of memory when working with R. Another potential problem of using up more memory than necessary during an execution of an R-script, is how R handles objects/variables and their names.

Consider the following line of code.

```
a <- runif(10000)
```

what is usually said to describe what is happening here is something along the lines of "we initiate a variable called `a` and assign a numeric vector with 10,000 random numbers. What in fact happens is that the name `a` is assigned to the integer vector (which in turn exists at a specific memory address). Thus values do not have names but *names have values*. This has important consequences for memory allocation and performance. For example, because `a` is in fact just a name attached to a value, the following does not involve any copying of values. It simply"binds" another name, `b`, to the same value to which `a` is already bound.

```
b <- a
```

We can prove this in two ways. First, if what I just stated was not true, the line above would actually lead to more memory being occupied by the current R session. However, this is not the case:

```
object_size(a)
```

```
## 80.05 kB
```

```
mem_change(c <- a)
```

```
## -635 kB
```

Second, we can see that the values to which `a` and `b` are bound are stored at the same memory address. Hence, they are the same values.

```
# load packages
library(lobstr)
```

```
##
## Attaching package: 'lobstr'
```

```
## The following objects are masked from 'package:pryr':
##
##     ast, mem_used
```

```
# check memory addresses of objects
obj_addr(a)
```

```
## [1] "0x555ade63c190"
```

```
obj_addr(b)
```

```
## [1] "0x555ade63c190"
```

Now you probably wonder, what happens to `b` if we modify `a`. After all, if the values to which `b` is bound are changed when we write code concerning `a`, we might end up with very surprising output. The answer is, and this is key (!), once we modify `a`, the values need to be *copied* in order to ensure the integrity of `b`. Only at this point, our program will require more memory.

```
# check the first element's value
a[1]
```

```
## [1] 0.234
```

```
b[1]
```

```
## [1] 0.234
```

```
# modify a, check memory change
mem_change(a[1] <- 0)
```

```
## 79 kB
```

```
# check memory addresses
obj_addr(a)
```

```
## [1] "0x555b179e2e50"
```

```
obj_addr(b)
```

```
## [1] "0x555ade63c190"
```

Note that the entire vector needed to be copied for this. There is, of course, a lesson from all this regarding writing efficient code. Knowing how actual copying of values does occur helps avoiding unnecessary copying. The larger an object, the more time it will take to copy it in memory. Objects with a single binding get modified in place (no copying):

```
mem_change(d <- runif(10000))
```

```
## 80.2 kB
```

```
mem_change(d[1] <- 0)
```

```
## 584 B
```

3.3.5 Releasing memory

Closely related to the issue of copy-upon-modify, is the issue of "releasing" memory via "garbage collection". If your program uses up a lot of (too much) memory (typical for working with large data sets), all processes on your computer might substantially slow down (we will look more closely into why this is the case in the next chapter). Hence, you

might want to remove/delete an object once you do not need it anymore. This can be done with the `rm()` function.

```
mem_change(large_vector <- runif(10^8))
```

```
## 800 MB
```

```
mem_change(rm(large_vector))
```

```
## -800 MB
```

`rm()` removes objects that are currently accessible in the global R environment. However, some objects/values might technically not be visible/accessible anymore (for example, objects that have been created in a function which has since returned the function output). To also release memory occupied by these objects you can call `gc()` (the garbage collector). While R will automatically collect the garbage once it is close to running out of memory, explicitly calling `gc` can still improve the performance of your script when working with large data sets. This is in particular the case when R is not the only data-intense process running on your computer. For example, when running an R script involving the repeated querying of data from a local SQL database and the subsequent memory-intense processing of this data in R, you can avoid using up too much memory by running `rm` and `gc` explicitly.[7]

3.3.6 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[8], meaning that when we execute R code, it is processed (statement by statement) by an 'in-

[7] Note that running `gc()` takes some time, so you should not overdo it. As a rule of thumb, run `gc()` after removing a really large object.

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

terpreter' 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)
##      }
## <bytecode: 0x555ad8639c18>
```

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 par-

ticularly prominent in the context of functions/packages made to deal with large amounts of data (such as the `data.table` package).

3.4 SQL basics

> Are tomorrow's bigger computers going to solve the problem? For some people, yes—their data will stay the same size and computers will get big enough to hold it comfortably. For other people it will only get worse—more powerful computers means extraordinarily larger datasets. If you are likely to be in this latter group, you might want to get used to working with databases now.

(Burns, 2011)

The Structured Query Language (SQL) has become a bread-and-butter tool for data analysts and data scientists due to its broad application in systems used to store large amounts of data. While traditionally only encountered in the context of structured data stored in relational database management systems, some versions of it are now also used to query data from data warehouse systems (e.g. Amazon Redshift) and even to query massive amounts (terabytes or even petabytes) of data stored in data lakes (e.g., Amazon Athena). In all of these applications, SQL's purpose (from the data analytics' perspective) is to provide a convenient and efficient way to query data from mass storage for analysis. Instead of importing a CSV file into R and then filtering it in order to get to the analytic data set, we use SQL to express how the analytic data set should look like (which variables and rows should be included).

The latter point is very important to keep in mind when already having experience with a language like R and learning SQL for the first time. In R we write code to instruct the computer what to do with the data. For example, we tell it to import a csv file called `economics.csv` as a

`data.table`, then we instruct it to remove observations which are older than a certain date according to the `date` column, then we instruct it to compute the average of the `unemploy` column values for each year based on the `date` column and then return the result as a separate data frame.

```
# import data
econ <- read.csv("data/economics.csv")

# filter
econ2 <- econ["1968-01-01"<=econ$date,]

# compute yearly averages (basic R approach)
econ2$year <- lubridate::year(econ2$date)
years <- unique(econ2$year)
averages <- sapply(years, FUN = function(x) mean(econ2[econ2$year==x,"unemploy"]))
output <- data.frame(year=years, average_unemploy=averages)

# inspect the first few lines of the result
head(output)
```

```
##   year average_unemploy
## 1 1968             2797
## 2 1969             2830
## 3 1970             4127
## 4 1971             5022
## 5 1972             4876
## 6 1973             4359
```

In contrast, when using SQL we write code that describes how the final result is supposed to look like. The SQL engine processing the code then takes care of the rest and returns the result in the most efficient way.[9]

[9] In particular, the user does not need to explicitly instruct SQL at which point in the process which part (filtering, selecting variables, aggregating, creating new variables etc.) of the query should be processed. SQL will automatically find the most efficient way to process the query.

TABLE 3.2: 6 records

year	average_unemploy
1968	2797
1969	2830
1970	4127
1971	5022
1972	4876
1973	4359

```
SELECT
strftime('%Y', `date`)  AS year,
AVG(unemploy) AS average_unemploy
FROM econ
WHERE "1968-01-01"<=`date`
GROUP BY year LIMIT 6;
```

For the moment, we will only focus on the code and ignore the underlying hardware and database concepts (those will be discussed in more detail in chapter 5).

3.4.1 First steps in SQL(ite)

In order to get familiar with coding in SQL, we work with a free and easy-to-use version of SQL called *SQLite*. SQLite[10] is a free full-featured SQL database engine widely used across platforms. It comes usually pre-installed with Windows and Mac/OSX distributions and has (from the user's perspective) all the core features of more sophisticated SQL versions. Unlike the more sophisticated SQL systems, SQLite does not rely explicitly on a client/server-model. That is, there is no need to set up your database on a server and then query it from a client interface. In fact, setting it up is straightforward. In the terminal, we can directly call SQLite as a command-line tool (on most modern computers the command is now `sqlite3`, SQLite version 3).

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

In this first code example, we set up an SQLite database using the command line. In the file structure of the book repository, we first switch to the data directory.

```
cd data
```

With one simple command, we start up SQLite, create a new database called `mydb.sqlite` and connect to the newly created database.[11]

```
sqlite3 mydb.sqlite
```

This created a new file `mydb.sqlite` in our `data` directory which contains the newly created database. And, we are now running `sqlite` in the terminal (indicated with the `sqlite>`. This means we can now type SQL code in the terminal to run queries and other SQL commands.

At this point, the newly created database does not contain any data yet. There are no tables in it. We can see this by running the `.tables` command.

```
.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` (similar to `character` in R) by default. Setting the right data type for each variable follows essentially the same logic as setting the data types of a data frame's columns in R (with the difference that in SQL this also affects how the data is stored on disk).[12]

In a first step, we thus create a new table called `econ`.

[11] If there is already a database called `mydb.sqlite` in this folder, the same command would simply start up SQLite and connect to the existing database.

[12] The most commonly used data types in SQL all have a very similar R equivalent: `DATE` is like `Date` in R, `REAL` like `numeric/double`, `INTEGER` like `integer`, and `TEXT` like `character`.

```
-- Create the new table
CREATE TABLE econ(
"date" DATE,
"pce" REAL,
"pop" REAL,
"psavert" REAL,
"uempmed" REAL,
"unemploy" INTEGER
);
```

Then, we can import the data from the csv file, by first switching to CSV mode via the command `.mode csv` and then importing the data to `econ` with `.import`. The `.import` command expects as a first argument the path to the CSV file on disk and as a second argument the name of the table to import the data to.

```
-- prepare import
.mode csv
-- import data from csv
.import --skip 1 economics.csv econ
```

Now we can have a look at the new database table in SQLite. `.tables` shows that we now have 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" REAL,
# "psavert" REAL,
# "uempmed" REAL,
# "unemploy" INTEGER
```

TABLE 3.3: 1 records

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

```
# );
```

With this, we can 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
```

```
.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';
```

3.4.1.1 Simple queries

Now let's select all dates and unemployment values of observations with more than 15 million unemployed, ordered by date.

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

3.4.2 Joins

So far, we have only considered queries involving one table of data. However, SQL provides a very efficient way to join data from various tables. Again, the way of writing SQL code is the same: you describe

TABLE 3.4: 9 records

date	unemploy
2009-09-01	15009
2009-10-01	15352
2009-11-01	15219
2009-12-01	15098
2010-01-01	15046
2010-02-01	15113
2010-03-01	15202
2010-04-01	15325
2010-11-01	15081

how the final table should look like and from where the data is to be selected.

Let's extend the previous example by importing an additional table to our `mydb.sqlite`. The additional data is stored in the file `inflation.csv` in the book's data folder and contains information on the US yearly inflation rate measured in percent.[13]

```
-- Create the new table
CREATE TABLE inflation(
"date" DATE,
"inflation_percent" REAL
);

-- prepare import
.mode csv
-- import data from csv
.import --skip 1 inflation.csv inflation
-- switch back to column mode
.mode columns
```

Note that the data stored in `econ` contains monthly observations, while

[13] Like the data stored in `economics.csv`, the data stored in `inflation.csv` is provided by the Federal Reserve Bank's (FRED)[https://fred.stlouisfed.org/] website.

`inflation` contains yearly observations. We can thus only meaningfully combine the two data sets at the level of years. Again using the combination of data sets in R as a reference point, here is what we would like to achieve expressed in R. The aim is to get a table that serves as basis for a Phillips curve[14] plot, with yearly observations and the variables `year`, `average_unemp_percent`, and `inflation_percent`.

```
# import data
econ <- read.csv("data/economics.csv")
inflation <- read.csv("data/inflation.csv")

# prepare variable to match observations
econ$year <- lubridate::year(econ$date)
inflation$year <- lubridate::year(inflation$date)

# create final output
years <- unique(econ$year)
averages <- sapply(years, FUN = function(x) {
        mean(econ[econ$year==x,"unemploy"]/econ[econ$year==x,"pop"])*100

} )
unemp <- data.frame(year=years,
                    average_unemp_percent=averages)

# combine via the year column
# keep all rows of econ
output<- merge(unemp, inflation[, c("year", "inflation_percent")], by="year")


# inspect output
head(output)
```

```
##   year average_unemp_percent inflation_percent
## 1 1967                 1.512             2.773
## 2 1968                 1.394             4.272
```

[14] https://en.wikipedia.org/wiki/Phillips_curve

TABLE 3.5: Displaying records 1 - 6

year	average_unemp_percent	inflation_percent
1967	1.512	2.773
1968	1.394	4.272
1969	1.397	5.462
1970	2.013	5.838
1971	2.419	4.293
1972	2.324	3.272

```
## 3 1969                1.396              5.462
## 4 1970                2.013              5.838
## 5 1971                2.419              4.293
## 6 1972                2.324              3.272
```

Now let's look at how the same table can be created in SQLite.

```
SELECT
strftime('%Y', econ.date)  AS year,
AVG(unemploy/pop)*100 AS average_unemp_percent,
inflation_percent
FROM econ INNER JOIN inflation ON year = strftime('%Y', inflation.date)
GROUP BY year
```

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

4

Hardware: Computing Resources

In order to better understand how we can use the available computing resources most efficiently in an analytics task, this chapter first briefly reviews the most basic hardware components and how they matter for computation. We then look at each of these components (and additional specialized components) through the lens of Big Data. That is, for each component, we look at how this component becomes a crucial bottleneck when processing large amounts of data and what we can do about it in R.

4.1 Components of a standard computing environment

Figure **??** 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 dataset as input.

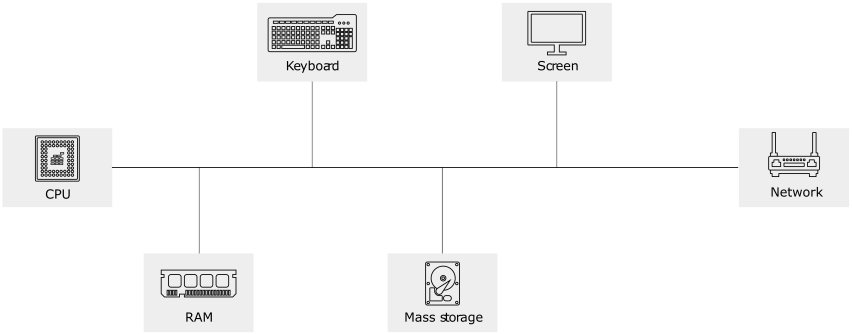


FIGURE 4.1: Basic components of a standard computing environment.

- *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 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 for as long as we are working with it.
- 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 sets of basic operations, which the *CPU* then executes.

For this and the later chapters, consider the main difference between common 'data analytics' and 'Big Data analytics' to be the following: in a Big Data analytics context, the standard usage of one or several of the standard hardware components in your local computer fails to work or works very inefficiently because the amount of data overwhelms its normal capacity.

From the 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 a large amount of 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 following sub-sections, we first get an idea of what we mean by capacity and *big* regarding the most important hardware components. First we focus on mass storage and memory, then on the CPU, and finally on new alternatives to the CPU.

4.2 Mass storage

In a simple computing environment, the mass storage device (hard disk) is where the data is stored to be analyzed. So, in what units do we measure the size of datasets and consequently the mass storage capacity of a computer? 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). When thinking of a given dataset in its raw/binary representation, we can simply think of it as a row of `0`s and `1`s.

Bigger units for storage capacity usually build on bytes, for example:

- 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

Currently, a common laptop or desktop computer has several hundred GBs of mass storage capacity. The problems related to a lack of mass storage capacity in Big Data analytics are likely the easiest to understand. Suppose you collect large amounts of data from an online source such as the Twitter API (application programming interface). At some point, R will throw an error and stop the data collection procedure as the operating system will not allow R to use up more disk space. The simplest solution to this problem is to clean up your hard disk: empty the trash, archive files in the cloud or on an external drive and delete them on the main disk, etc. In addition, there are some easy-to-learn tricks to use from within R to save some disk space.

4.2.1 Avoid redundancies

Different formats for structuring data stored on disk use up more or less space. A simple example is the comparison of JSON (JavaScript Ob-

ject Notation) and CSV (Comma Separated Values) – both data structures that are widely used to store data for analytics purposes. JSON is much more flexible in that it allows the definition of arbitrarily complex hierarchical data structures (and even allows for hints at data types). However, this flexibility comes with some overhead in the usage of special characters to define the structure. Consider the following JSON excerpt of an economic time series fetched from the Federal Reserve's FRED API[1].

```
{
    "realtime_start": "2013-08-14",
    "realtime_end": "2013-08-14",
    "observation_start": "1776-07-04",
    "observation_end": "9999-12-31",
    "units": "lin",
    "output_type": 1,
    "file_type": "json",
    "order_by": "observation_date",
    "sort_order": "asc",
    "count": 84,
    "offset": 0,
    "limit": 100000,
    "observations": [
        {
            "realtime_start": "2013-08-14",
            "realtime_end": "2013-08-14",
            "date": "1929-01-01",
            "value": "1065.9"
        },
        {
            "realtime_start": "2013-08-14",
            "realtime_end": "2013-08-14",
            "date": "1930-01-01",
            "value": "975.5"
        },
```

[1] https://fred.stlouisfed.org/docs/api/fred/series_observations.html#example_json

```
        ...,
        {
            "realtime_start": "2013-08-14",
            "realtime_end": "2013-08-14",
            "date": "2012-01-01",
            "value": "15693.1"
        }
    ]
}
```

The JSON format is very practical here in separating metadata (such as what time frame is covered by this dataset, etc.) in the first few lines on top from the actual data in `"observations"` further down. However, note that due to this structure, the key names like `"date"`, and `"value"` occur for each observation in that time series. In addition, `"realtime_start"` and `"realtime_end"` occur both in the metadata section and again in each observation. Each of those occurrences costs some bytes of storage space on your hard disk but does not add any information once you have parsed and imported the time series into R. The same information could also be stored in a more efficient way on your hard disk by simply storing the metadata in a separate text file and the actual observations in a CSV file (in a table-like structure):

```
"date","value"
"1929-01-01", "1065.9"
"1930-01-01", "975.5"

...,

"2012-01-01", 15693.1"
```

In fact, in that particular example, storing the data in JSON format would take up more than double the hard-disk space as CSV. Of course, this is not to say that one should generally store data in CSV files. In many situations, you might really have to rely on JSON's flexibility to represent more complex structures. However, in practice it is very much worthwhile to think about whether you can improve storage efficiency by simply storing raw data in a different format.

Another related point to storing data in CSV files is to remove redundancies by splitting the data into several tables/CSV files, whereby each table contains the variables exclusively describing the type of observation in it. For example, when analyzing customer data for marketing purposes, the dataset stored in one CSV file might be at the level of individual purchases. That is, each row contains both information on what has been purchased on which day by which customer as well as additional variables describing the customer (such as customer id, name, address, etc.). Instead of keeping all of this data in one file, we could split it into two files, where one only contains the order ids and corresponding customer ids as well as attributes of individual orders (but not additional attributes of the customers themselves) and the other contains the customer ids and all customer attributes. Thereby, we avoid redundancies in the form of repeatedly storing the same values of customer attributes (like name and address) for each order.[2]

4.2.2 Data compression

Data compression essentially follows from the same basic idea of avoiding redundancies in data storage as the simple approaches discussed above. However, it happens on a much more fundamental level. Data compression algorithms encode the information contained in the original representation of the data with fewer bits. In the case of lossless compression, this results in a new data file containing the exact same information but taking up less space on disk. In simple terms, compression replaces repeatedly occurring sequences with shorter expressions and keeps track of replacements in a table. Based on the table, the file can then be de-compressed to recreate the original representation of the data. For example, consider the following character string.

```
"xxxxxyyyyyzzzz"
```

The same data could be represented with fewer bits as:

[2] This concept of organizing data into several tables is the basis of relational database management systems, which we will look at in more detail in Chapter 5. However, the basic idea is also very useful for storing raw data efficiently even if there is no intention to later build a database and run SQL queries on it.

```
"5x6y4z"
```

which needs fewer than half the number of bits to be stored (but contains the same information).

There are several easy ways to use your mass storage capacity more efficiently with data compression in R. Most conveniently, some functions to import/export data in R directly allow for reading and writing of compressed formats. For example, the `fread()`/`fwrite()` functions provided in the `data.table` package will automatically use the GZIP (de-)compression utility when writing to (reading from) a CSV file with a `.gz` file extension in the file name.

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

# load example data from basic R installation
data("LifeCycleSavings")

# write data to normal csv file and check size
fwrite(LifeCycleSavings, file="lcs.csv")
file.size("lcs.csv")
```

```
## [1] 1441
```

```
# write data to a GZIPped (compressed) csv file and check size
fwrite(LifeCycleSavings, file="lcs.csv.gz")
file.size("lcs.csv.gz")
```

```
## [1] 744
```

```
# read/import the compressed data
lcs <- data.table::fread("lcs.csv.gz")
```

Alternatively, you can also use other types of data compression as follows.

```
# common ZIP compression (independent of data.table package)
write.csv(LifeCycleSavings, file="lcs.csv")
file.size("lcs.csv")
```

```
## [1] 1984
```

```
zip(zipfile = "lcs.csv.zip", files =  "lcs.csv")
file.size("lcs.csv.zip")
```

```
## [1] 1205
```

```
# unzip/decompress and read/import data
lcs_path <- unzip("lcs.csv.zip")
lcs <- read.csv(lcs_path)
```

Note that data compression is subject to a time–memory trade-off. Compression and de-compression is computationally intense and needs time. When using compression in order to make more efficient use of the available mass storage capacity, think about how frequently you expect the data to be loaded into R as part of the data analysis tasks ahead and for how long your will need to keep the data stored on your hard disk. Importing GBs of compressed data can be uncomfortably slower than importing from a decompressed file.

So far, we have only focused on data size in the context of mass storage capacity. But what happens once you load a large dataset into R (e.g., by means of `read.csv()`)? A program called a "parser" is executed that reads the raw data from the hard disk and creates a representation of that data in the R environment, that is, in random access memory (RAM). All common computers have more GBs of mass storage available than GBs of RAM. Hence, new issues of hardware capacity loom at the stage of data import, which brings us to the next subsection.

4.3 Random access memory (RAM)

Currently, a common laptop or desktop computer has 8–32 GB of RAM capacity. These are more-or-less the numbers you should keep in the back of your mind for the examples/discussions that follow. That is, we will consider a dataset as "big" if it takes up several GBs in RAM (and therefore might overwhelm a machine with 8GB RAM capacity).

There are several types of problems that you might run into in practice when attempting to import and analyze a dataset of the size close to or larger than your computer's RAM capacity. First, importing the data might take much longer than expected, your computer might freeze during import (or later during the analysis), R/Rstudio might crash, or you might get an error message hinting at a lack of RAM. How can you anticipate such problems, and what can you do about them?

Many of the techniques and packages discussed in the following chapters are in one way or another solutions to these kinds of problems. However, there are a few relatively simple things to keep in mind before we go into the details.

1. The same data stored on the mass storage device (e.g., in a CSV file) might take up more or less space in RAM. This is due to the fact that the data is (technically speaking) structured differently in a CSV or JSON file than in, for example, a data table or a matrix in R. For example, it is reasonable to anticipate that the example JSON file with the economic time series data will take up less space as a time series object in R (in RAM) than it does on the hard disk (for one thing just simply due to the fact that we will not keep the redundancies mentioned before).
2. The import might work well, but some parts of the data analysis script might require much more memory to run through even without loading additional data from disk. A classic example of this is regression analysis performed with, for example, `lm()` in R. As part of the OLS estimation procedure,

`lm` will need to create the model matrix (usually denoted X). Depending on the model you want to estimate, the model matrix might actually be larger than the data frame containing the dataset. In fact, this can happen quite easily if you specify a fixed effects model in which you want to account for the fixed effects via dummy variables (for example, for each country except for one).[3] Again, the result can be one of several: an error message hinting at a lack of memory, a crash, or the computer slowing down significantly. Anticipating these types of problems is very tricky since memory problems are often caused at a lower level of a function from the package that provides you with the data analytics routine you intend to use. Accordingly, error messages can be rather cryptic.

3. Keep in mind that you have some leeway to guide how much space imported data takes up in R by considering data structures and data types. For example, you can use factors instead of character vectors when importing categorical variables into R (the default in `read.csv`), and for some operations it makes sense to work with matrices instead of data frames.

Finally, recall the lessons regarding memory usage from the section "Writing efficient R code" in Chapter 1.

4.4 Combining RAM and hard disk: virtual memory

What if all the RAM in our computer is not enough to store all the data we want to analyze?

Modern operating systems (OSs) have a way of 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*. Figure 4.2 illustrates this point.

[3] For example, if you specify something like `lm(y~x1 + x2 + country, data=mydata)` and `country` is a categorical variable (factor).

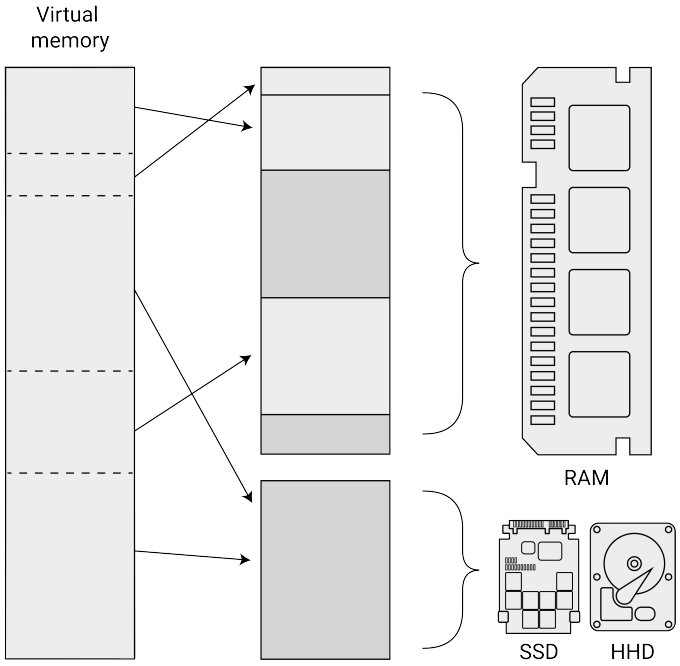


FIGURE 4.2: Virtual memory. Overall memory is mapped to RAM and parts of the hard disk.

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 will 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 for running 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 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 ba-

sic idea behind these packages is to map a dataset 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 dataset is loaded into R.

4.5 CPU and parallelization

The actual processing of the data is done in the computer's central processing unit (CPU). Consequently, the performance of the CPU has a substantial effect on how fast a data analytics task runs. A CPU's performance is usually denoted by its *clock rate* measured in gigaherz (GHz). In simple terms, a CPU with a clock rate of 4.8 GHz can execute 4.8 billion basic operations per second. Holding all other aspects constant, you can thus expect an analytics task to run faster if it runs on a computer with higher CPU clock rate. Alternatively to scaling up the CPU, we can exploit the fact that modern CPUs have several *cores*. In the normal usage of a PC, the operating system makes use of these cores to run several applications smoothly *in parallel* (e.g., you listen to music on Spotify while browsing the web and running some analytics script in RStudio in the background).

Modern computing environments such as R allow us to explicitly run parts of the same analytics task in parallel, that is, on several CPU cores at the same time. Following the same logic, we can also connect several computers (each with several CPU cores) in a cluster computer and run the program in parallel on all of these computing nodes. Both of these approaches are generally referred to as *parallelization* and both are supported in several R packages.

An R program run in parallel typically involves the following steps

- First, several instances of R are running at the same time (across one machine with multiple CPU cores or across a cluster computer). One of the instances (i.e., the *master* instance) breaks the computation into batches and sends those to the other instances.
- Second, each of the instances processes its batch and sends the results back to the master instance.

- Finally, the master instance combines the partial results into the final result and returns it to the user.

To illustrate this point, consider the following econometric problem: you have a customer dataset[4] with detailed data on customer characteristics, past customer behavior, and information on online marketing campaigns. Your task is to figure out which customers are more likely to react positively to the most recent online marketing campaign. The aim is to optimize personalized marketing campaigns in the future based on insights gained from this exercise. In a first step you take a computationally intense "brute force" approach: you run all possible regressions with the dependent variable `Response` (equal to 1 if the customer took the offer in the campaign and 0 otherwise). In total you have 21 independent variables, thus you need to run $2^20 = 1,048,576$ logit regressions (this is without considering linear combinations of covariates etc.). Finally, you want to select the model with the best fit according to deviance.

A simple sequential implementation to solve this problem could look like this (for the sake of time, we cap the number of regression models to N=10).

```
# you can download the dataset from
# https://www.kaggle.com/jackdaoud/marketing-data?
# select=marketing_data.csv

# PREPARATION ------------------------------

# packages
library(stringr)

# import data
marketing <- read.csv("data/marketing_data.csv")
# clean/prepare data
marketing$Income <- as.numeric(gsub("[[:punct:]]",
                                    "",
```

[4] https://www.kaggle.com/jackdaoud/marketing-data?select=marketing_data.csv

```
                                    marketing$Income))
marketing$days_customer <-
     as.Date(Sys.Date())-
     as.Date(marketing$Dt_Customer, "%m/%d/%y")
marketing$Dt_Customer <- NULL

# all sets of independent vars
indep <- names(marketing)[ c(2:19, 27,28)]
combinations_list <- lapply(1:length(indep),
                            function(x) combn(indep, x,
                                              simplify = FALSE))
combinations_list <- unlist(combinations_list,
                            recursive = FALSE)
models <- lapply(combinations_list,
                 function(x) paste("Response ~",
                                   paste(x, collapse="+")))

# COMPUTE REGRESSIONS --------------------------

N <- 10 #  N <- length(models) for all
pseudo_Rsq <- list()
length(pseudo_Rsq) <- N
for (i in 1:N) {
  # fit the logit model via maximum likelihood
  fit <- glm(models[[i]], data=marketing, family = binomial())
  # compute the proportion of deviance explained by
  # the independent vars (~R^2)
  pseudo_Rsq[[i]] <- 1-(fit$deviance/fit$null.deviance)
}

# SELECT THE WINNER ---------------
models[[which.max(pseudo_Rsq)]]
```

```
## [1] "Response ~ MntWines"
```

4.5.1 Naive multi-session approach

There is actually a simple way of doing this "manually" on a multi-core PC, which intuitively illustrates the point of parallelization (although it would not be a very practical approach): you write an R script that loads the dataset, runs the fist n of the total of N regressions and stores the result in a local text file. Next, you run the script in your current RStudio session, open an additional RStudio session and run the script with the next n regressions, and so on until all cores are occupied with one RStudio session. At the end you collect all of the results from the separate text files and combine them to get the final result. Depending on the problem at hand, this could indeed speed up the overall task, and it is technically speaking a form of "multi-session" approach. However, as you have surely noticed, this is unlikely to be a very practical approach.

4.5.2 Multi-core and multi-node approach

A more practical approach is to write one R script (with the help of some specialized packages) that instructs R to automatically distribute the batches to different cores (or different computing nodes in a cluster computer), control and monitor the progress in all cores, and then automatically collect and combine the results from all cores. There are several approaches to achieve this in R.

4.5.2.1 Parallel for-loops using socket

Likely the most intuitive approach to parallelizing a task in R is the `foreach` package. It allows you to write a `foreach` statement that is very similar to the for-loop syntax in R. Hence, you can straightforwardly "translate" an already implemented sequential approach with a common for-loop to a parallel implementation.

```
# COMPUTE REGRESSIONS IN PARALLEL (MULTI-CORE) ---------------------------

# packages for parallel processing
library(parallel)
library(doSNOW)
```

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

# prepare loop
N <- 10000 #  N <- length(models) for all
# run loop in parallel
pseudo_Rsq <-
  foreach ( i = 1:N, .combine = c) %dopar% {
    # fit the logit model via maximum likelihood
    fit <- glm(models[[i]],
               data=marketing,
               family = binomial())
    # compute the proportion of deviance explained by
    # the independent vars (~R^2)
    return(1-(fit$deviance/fit$null.deviance))
}

# SELECT THE WINNER ---------------
models[[which.max(pseudo_Rsq)]]
```

```
## [1] "Response ~ Year_Birth+Teenhome+Recency+MntWines+days_customer"
```

With relatively few cases, this approach is not very fast due to the overhead of "distributing" variables/objects from the master process to all cores/workers. In simple terms, the socket approach means that the cores do not share the same variables/the same environment, which creates overhead. However, this approach is usually very stable and runs on all platforms.

4.5.2.2 Parallel lapply using forking

Finally, let us look at the implementation based on forking. In the fork approach, each core works with the same objects/variables in a shared environment, which makes this approach very fast. However, depend-

ing on what exactly is being computed, sharing an environment can cause problems.[5] If you are not sure whether your set up might run into issues with forking, it would be better to rely on a non-fork approach.

```
# COMPUTE REGRESSIONS IN PARALLEL (MULTI-CORE) ---------------


# prepare parallel lapply (based on forking,
# here clearly faster than foreach)
N <- 10000 #  N <- length(models) for all
# run parallel lapply
pseudo_Rsq <- mclapply(1:N,
                       mc.cores = ncores,
                       FUN = function(i){
                         # fit the logit model
                         fit <- glm(models[[i]],
                                    data=marketing,
                                    family = binomial())
                         # compute the proportion of deviance
                         # explained  by the independent vars (~R^2)
                         return(1-(fit$deviance/fit$null.deviance))
                         })

# SELECT THE WINNER, SHOW FINAL OUTPUT ---------------

best_model <- models[[which.max(pseudo_Rsq)]]
best_model
```

```
## [1] "Response ~ Year_Birth+Teenhome+Recency+MntWines+days_customer"
```

[5] Also, this approach does not work on Windows machines (see `?mclapply` for details).

4.6 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. Figure 4.3 depicts a modern graphics card similar to those commonly built into today's 'gaming' PCs.

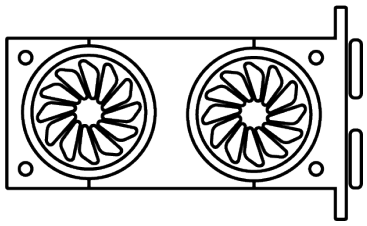


FIGURE 4.3: Illustration of a graphic processing unit (GPU).

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 kinds 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 in highly parallel numerical [floating point] workloads) turns out to also be 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 in turn has several cores.

GPUs can thus perform computations with hundreds or even thousands of threads in parallel. The figure below illustrates this point by showing the typical architecture of a NVIDIA GPU.

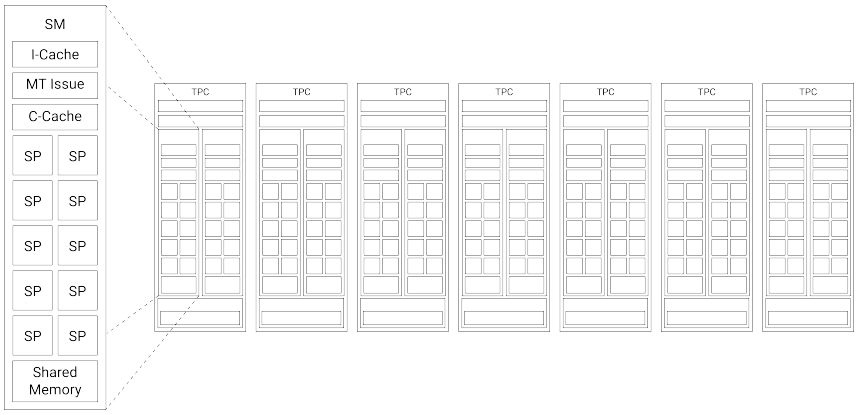


FIGURE 4.4: Illustration of a graphic processing unit (GPU).

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 now provide several high-level APIs to use GPUs for tasks other than graphics processing. Over the last few years, more high-level software has been developed that makes it much easier to use GPUs in parallel computing tasks. The following subsections show some examples of such software in the R environment.[6]

4.6.1 GPUs in R

4.6.1.1 GPU computing example: Matrix multiplication comparison (gpuR)

The gpuR package provides basic R functions to compute with GPUs from within the R environment. In the following example we compare the performance of a CPU with a GPU based on a matrix multiplication exercise. For a large $N \times P$ matrix X, we want to compute $X^t X$.

[6] 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 computer's system. The examples presuppose that all installation and configuration steps (GPU drivers, CUDA, etc.) have already been completed successfully.

In a first step, we load the `gpuR`-package.[7] 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.

```
# load package
library(bench)
library(gpuR)
```

```
## Number of platforms: 1
## - platform: NVIDIA Corporation: OpenCL 3.0 CUDA 11.6.134
##   - context device index: 0
##     - NVIDIA GeForce GTX 1650
## checked all devices
## completed initialization
```

Next, we initialize a large matrix filled with pseudo-random numbers, representing a dataset with N observations and P variables.

```
# initialize 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.[8] 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 actu-

[7] As with setting up 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.

[8] If we instruct the GPU to use its own memory but the data does not fit in it, the program will result in an error.

ally transfer data to the GPU's memory. The latter is done in the other variant, with `vclMatrix()`.

```
# prepare GPU-specific objects/settings

# point GPU to matrix (matrix stored in non-GPU memory)
gpuX <- gpuMatrix(X, type = "float")
# transfer matrix to GPU (matrix stored in GPU memory)
vclX <- vclMatrix(X, type = "float")
```

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.

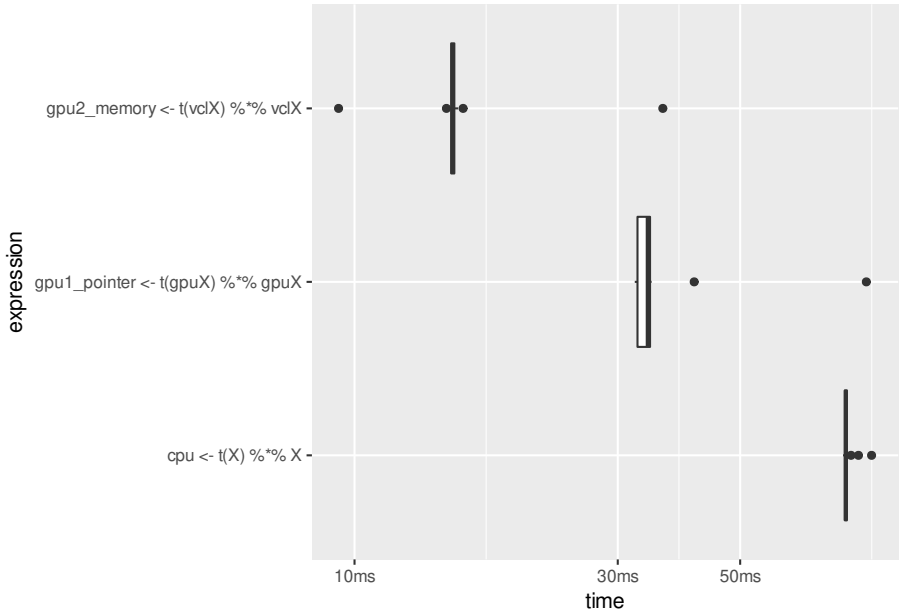
```
# compare three approaches
(gpu_cpu <- bench::mark(

  # compute with CPU
  cpu <- t(X) %*% X,

  # GPU version, GPU pointer to CPU memory
  # (gpuMatrix is simply a pointer)
  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                     76.8ms   77.7ms
## 2 gpu1_pointer <- t(gpuX) %*% gpuX  32.21ms     34ms
```

```
## 3 gpu2_memory <- t(vclX) %*% vclX    9.36ms   15.1ms
## # ... with 3 more variables: `itr/sec` <dbl>,
## #   mem_alloc <bch:byt>, `gc/sec` <dbl>
## # i Use `colnames()` to see all variable names
```

The performance comparison is visualized with boxplots.

```
plot(gpu_cpu, type = "boxplot")
```



The theoretically expected pattern becomes clearly visible. When using the GPU + GPU memory, the matrix operation takes on average less than 20 ms, with GPU + RAM over 30 ms, and with the common CPU operation close to 100 ms to finish. In the chapters on applied data analysis, we will look at some applications of GPUs in the domain of deep learning (which relies heavily on matrix multiplications).

4.7 The road ahead: Hardware made for machine learning

Due to the high demand for more computational power in the domain of training complex neural network models (for example, in computer vision), Google has recently developed a new hardware platform specifically designed to work with complex neural networks using TensorFlow: Tensor Processing Units (TPUs). TPUs were designed from the ground up to improve performance in dense vector and matrix computations with the aim of substantially increasing the speed of training deep learning models implemented with TensorFlow.

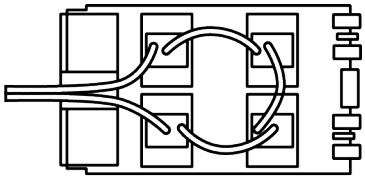


FIGURE 4.5: Illustration of a tensor processing unit (GPU).

While initially only used internally by Google, the Google Cloud platform now offers cloud TPUs to the general public.

4.8 Insufficient computing resources?

When working with very large datasets (i.e., terabytes of data), processing the data on one common computer might not work due to a lack of memory or would be way too slow due to a lack of computing power (CPU cores). The architecture or basic hardware set up of a common computer is subject to a limited amount of RAM and a limited number of CPUs/CPU cores. Hence simply scaling up might not be sufficient. Instead we need to scale out. In simple terms, this means connecting several computers (each with their own RAM, CPU, and mass storage) in a network, distributing the dataset across all computers ("nodes") in this network, and working on the data simultaneously across all nodes. In the next chapter, we look into how such "dis-

tributed systems'' basically work, what software frameworks are commonly used to work on distributed systems, and how we can interact with this software (and the distributed system) via R and SQL.

5

Distributed Systems

When we connect several computers in a network to jointly process large amounts of data, such a computing system is commonly referred to as a "distributed system". From a technical standpoint the key difference between a distributed system and the more familiar parallel system (e.g. our desktop computer with its multicore CPU) is that in distributed systems the different components do not share the same memory (and storage). Figure 5.1 illustrates this point.

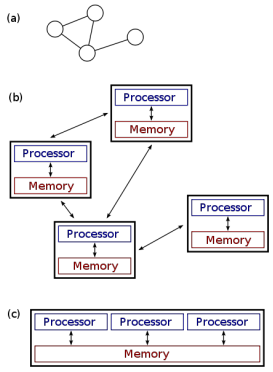


FIGURE 5.1: (a), (b): a distributed system. (c): a parallel system.I llustration by Miym (CC BY-SA 3.0).

In a distributed system, the data set is literally split up into pieces that then reside separately on different nodes. This requires an additional "layer" of software (that coordinates the distribution/loading of data as well as the simultaneous processing) and a different approach (a different "programming model") to defining computing/data analytics tasks. Below, we will look at both of these aspects in turn.

5.1 MapReduce

The most broadly used programming model and its implementation for processing big data on distributed systems is called MapReduce. It essentially consists of two procedures and is conceptually very close to the "split-apply-combine" strategy in data analysis. First, the Map function sorts/filters the data (on each node). Then, a Reduce function aggregates the sorted/filtered data. In the background, the MapReduce "framework" orchestrates these processes across all nodes, collects the results, and returns them to the user.

Let us illustrate the basic idea behind MapReduce with a simple example. Suppose you are working on a text mining task in which all the raw text in thousands of digitized books (stored as text files) need to be processed. In a first step, you want to compute word frequencies (countthe number of occurrences of specific words in all books combined).

For simplicity, let us only focus on the following excerpts from two exemplary (and rather cryptic) books on botany:

Text in book 1:

Apple Orange Mango

Orange Grapes Plum

Text in book 2:

Apple Plum Mango

Apple Apple Plum

The figure below illustrates the MapReduce process. First, the data is loaded from the original text files. Each line of text is then passed to individual mapper instances which separately split the lines of text into key-value pairs. Then the system sorts and shuffles all key-value pairs across all instances and finally. The reducer aggregates the sorted/shuffled key-value pairs (here: counts the number of word occurrences) and, finally, the master instance collects all the results and

returns the final output. From this simple example, a key aspect of MapReduce should become clear: for the key tasks of mapping and reducing the data processing on one node/instance can happen completely independent from the processing on the other instances. Note that not for every data anlytics task this is so easily conceivable as for computing word frequencies.

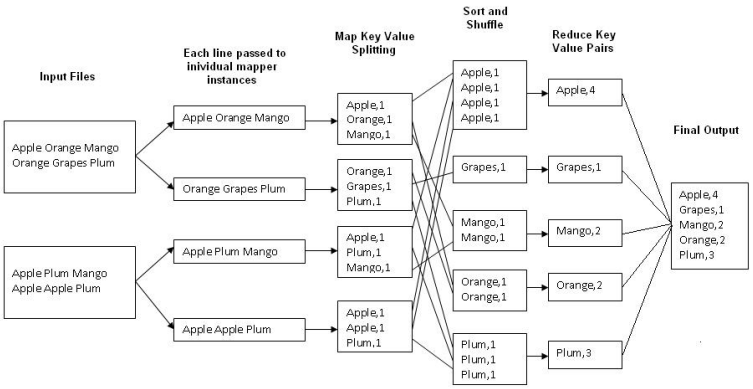


FIGURE 5.2: Illustration of the MapReduce programming model.

5.1.1 Map/Reduce Concept Illustrated 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 functions `map()` and `reduce()` in R to implement the basic MapReduce example shown above (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).[1] 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

[1]For a more detailed discussion of what `map` and `reduce` have *actually* to do with MapReduce see this post[2].

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.

```
# initiate the input text (for simplicity as one text string)

input_text <-

"Apple Orange Mango
Orange Grapes Plum
Apple Plum Mango
Apple Apple Plum"
```

5.1.2 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] "Apple Orange Mango"
##
## [[2]]
## [1] "Orange Grapes Plum"
##
## [[3]]
## [1] "Apple Plum Mango"
##
## [[4]]
## [1] "Apple Apple Plum"
```

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]]
##  Apple Orange  Mango
##      1      1      1
##
## [[2]]
## Orange Grapes   Plum
##      1      1      1
##
## [[3]]
## Apple  Plum Mango
##     1     1     1
##
## [[4]]
```

```
## Apple Apple  Plum
##     1     1     1
```

5.1.3 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]]
## Apple Apple Apple Apple
##     1     1     1     1
##
## [[2]]
## Grapes
##      1
##
## [[3]]
## Mango Mango
##     1     1
##
## [[4]]
## Orange Orange
##      1      1
##
## [[5]]
## Plum Plum Plum
##    1    1    1
```

Now we can sum up the keys in order to get the word count for the entire input.

```
sums <- lapply(shuffled, Reduce, f=sum)
names(sums) <- keys
sums
```

```
## $Apple
## [1] 4
##
## $Grapes
## [1] 1
##
## $Mango
## [1] 2
##
## $Orange
## [1] 2
##
## $Plum
## [1] 3
```

5.2 Hadoop

Hadoop is probably the most broadly known and used system to implement the MapReduce framework. A decade ago big data analytics with really large data sets often involved directly interacting with/ working in Hadoop to run MapReduce jobs. However, over the last few years various higher-level interfaces have been developed that make the usage of MapReduce/Hadoop for data analysts much more easily accessible. The purpose of this section is thus to give a lightweight introduction to the underlying basics that power some of the code examples and tutorials discussed in the data analytics chapters towards the end of this book.

5.2.1 Hadoop word count example

To get an idea of how running a Hadoop job looks like, we run the same simple word count example introduced above on a local Hadoop installation. The example presupposes a local installation of Hadoop version 2.10.1 (see Appendix C for details) and can easily be run on a completely normal desktop/laptop computer running Ubuntu Linux. As a side remark, this actually illustrates an important aspect of developing MapReduce scripts in Hadoop (and many of the software packages building on it): the code can be easily developed tested locally on a small machine and only later transferred to the actual Hadoop cluster to be run on the full data set.

The basic Hadoop installation comes with a few templates for very typical map/reduce programs.[3] 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 text file containing the same text as in the example above.

```
echo "Apple Orange Mango
Orange Grapes Plum
```

[3] More sophisticated programs need to be custom made, written in Java.

```
Apple Plum Mango
Apple Apple Plum" >>  ~/input/text.txt
```

Now we can run the MapReduce/Hadoop word count as follows, storing the results in a new directory called `wordcount_example`. This is where we use the already implemented Hadoop script to run a word count job Map/Reduce style. This is where we rely on the already implemented wordcount example provided with the Hadoop installation (located in `/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.10.1.jar`)

```
# run mapreduce word count
/usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-exa
```

What this line says is: run the Hadoop program called `wordcount` implemented in the jar-file `hadoop-mapreduce-examples-2.10.1.jar`, use the files in directory `~/input` containing the raw text as input and store the final output in directory `~/wc_example`.

```
cat ~/wc_example/*
```

```
## Apple    4
## Grapes   1
## Mango    2
## Orange   2
## Plum 3
```

What looks rather simple in this example can get very complex once you want to write an entire data analysis script with all kind of analysis for Hadoop. Also, Hadoop was designed for batch processing and does not offer a simple interface for interactive sessions. All of this makes it rather impractical for a usual analytics workflow as we know it from working with R. This is where Apache Spark[4] comes to the rescue.

[4] https://spark.apache.org/

5.3 Spark

Also building on the MapReduce model, Spark is a cluster computing platform particularly made for data analytics. It partially relies on Hadoop for handling storage and resource management, but offers way more easy-to-use high-level interfaces for typical analytics tasks. From the technical perspective (in very simple terms), Spark also addresses some shortcomings of the Hadoop platform, further improving efficiency/speed for many data analysis tasks. More importantly for our purposes, in contrast to Hadoop, Spark is specifically made for interactively developing and runnin data analytics scripts 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, as we will see in later chapters, it is very easy to use interactively from within R (and other languages like Python, SQL, and Scala). 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 generalized linear regression, summary statistics, and principal component analysis.

At the heart of big data analytics with Spark is the fundamental data structure called 'resilient distributed data set' (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. How-

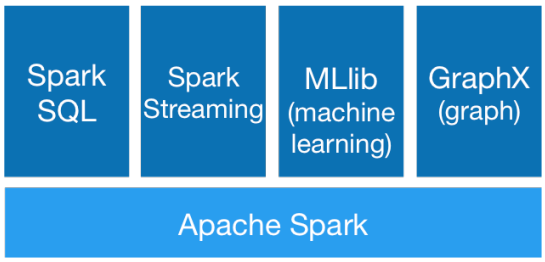


FIGURE 5.3: Basic Spark stack (source: https://spark.apache.org/images/spark-stack.png)

ever, 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 data set.

5.4 Spark with 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`').[5] 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 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

[5] See this blog post[6] for a more detailed comparison and discussion of advantages of either package.

`SparkR` from GitHub (needs the `devtools`-package) `devtools::install_github("cran/SparkR")`. After installing `SparkR`, the call `SparkR::install.spark()` will download and install Apache Spark to a local directory.[7] Now we can start an interactive SparkR session from the terminal with

```
$ SPARK-HOME/bin/sparkR
```

where `SPARK-HOME` is a placeholder for the path to your local Spark installation (printed to the console after running `SparkR::install.spark()`). Or simply run SparkR from within RStudio by loading `SparkR` and initiating Spark with `sparkR.session()`.

```
# to install use
# devtools::install_github("cran/SparkR")

# 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.[8]

5.4.1 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

[7] Note that after the installation, the location of Spark is printed to the R console. Alternatively, you can also first install the `sparklyr`-package and then run `sparklyr::spark_install()` to install Spark. In the data analysis examples later in the book we will work both with `SparkR` and `sparklyr`.

[8] 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.

statistics. We analyze the already familiar `flights.csv` data set. The basic Spark installation provides direct support to import common data formats such as CSV and JSON via the `read.df()` function (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("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
```

5.5 Spark with SQL

Instead of interacting with Spark via R, you can do the same via SQL. This can be very convenient at the stage of data exploration and data preparation. Also note that this is a very good example of how knowing some SQL can be very useful when working with Big Data even if you are not interacting with an actual relational database.[9]

To directly interact with Spark via SQL, open a terminal window, switch to the `SPARK-HOME` directory,

```
cd SPARK-HOME
```

[9] Importantly, this also means that we cannot use SQL commands related to configuring such databases such as `.tables` etc. Instead we use SQL commands to directly query data from JSON or CSV files.

and enter the following command,

```
$ bin/spark-sql
```

where `SPARK-HOME` is again the placeholder for the path to your local Spark installation (printed to the console after running `SparkR::install.spark()`). This will start up Spark and connect to it via Spark's sql interface. You will notice that the prompt in the terminal changes (similar to when you start `sqlite`).

Let's run some example queries. The Spark installation comes with several data and script examples. The example data sets are located at `SPARK-HOME/examples/src/main/resources`. For example, the file `employees.json` contains the following records in JSON format:

```
{"name":"Michael", "salary":3000}
{"name":"Andy", "salary":4500}
{"name":"Justin", "salary":3500}
{"name":"Berta", "salary":4000}
```

We can query this data directly via SQL commands by referring to the location of the original JSON file.

Select all observations

```
SELECT *
FROM json.`examples/src/main/resources/employees.json`
;
```

```
Michael 3000
Andy    4500
Justin  3500
Berta   4000
Time taken: 0.099 seconds, Fetched 4 row(s)
```

Filter observations

```
SELECT *
FROM json.`examples/src/main/resources/employees.json`
WHERE salary <4000
;
```

```
Michael 3000
Justin  3500
Time taken: 0.125 seconds, Fetched 2 row(s)
```

Compute the average salary

```
SELECT AVG(salary) AS mean_salary
FROM json.`examples/src/main/resources/employees.json`

;
```

```
3750.0
Time taken: 0.142 seconds, Fetched 1 row(s)
```

5.6 Spark with R + SQL

Most conveniently, you can combine the SQL query features of Spark and SQL with running R on Spark. First, initiate the Spark session in RStudio and import the data as Spark data frame.

```
# to install use
# devtools::install_github("cran/SparkR")

# load packages
library(SparkR)

# start session
sparkR.session()
```

```
## Java ref type org.apache.spark.sql.SparkSession id 1
```

```
# read data
flights <- read.df("data/flights.csv", source = "csv", header="true")
```

Now we can make the Spark data frame accessible for SQL queries by registering it as a temporary table/view with `createOrReplaceTempView()` and then run SQL queries on it from within the R session via the `sql()`-function. `sql()` will return the results as Spark data frame (this means the result is also located on the cluster and does hardly affect the master node's memory).

```
# register the data frame as a table
createOrReplaceTempView(flights, "flights" )

# now run SQL queries on it
query <-
"SELECT DISTINCT carrier,
year,
arr_delay,
distance
FROM flights
WHERE 1000 <= distance"

long_flights2 <- sql(query)
head(long_flights2)
```

```
##   carrier year arr_delay distance
## 1      DL 2013       -30     1089
## 2      UA 2013       -11     1605
## 3      DL 2013       -42     1598
## 4      UA 2013        -5     1585
## 5      AA 2013         6     1389
## 6      UA 2013       -23     1620
```

6

Cloud Computing

In this chapter, we first look at what cloud computing basically is and what platforms provide cloud computing services. We then focus on *scaling up* in the cloud. For the sake of simplicity, we will primarily focus on how to use cloud instances provided by one of the providers, Amazon Web Services (AWS). However, once you are familiar with setting things up on AWS, also using Google Cloud, Azure, etc. will be easy. Most of the core services are provided by all providers and once you understand the basics the different dashboards will look quite familiar. In a second step, we look at a prominent approach to *scaling out* by setting up a Spark cluster in the cloud.

6.1 Cloud computing basics and platforms

So far we have focused on the available computing resources on our local machines (desktop/laptop) and how to use them optimally when dealing with large amounts of data and/or computationally demanding tasks. A key aspect of this has been to understand why our local machine is struggling with a computing task when there is a large amount of data to be processed and then identify potential avenues to use the available resources more efficiently. For example, by using one of the following approaches:

- 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.
- Memory-intense tasks (data does not fit into RAM): efficient use of virtual memory (use parts of mass storage device as virtual memory).

- Storage: efficient storage (avoid redundancies).

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, 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 (typically, over the Internet). While only one or two decades 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 computing resources (usually, virtual servers) that can be easily accessed via a broadband Internet-connection and rented on an hourly (or even minutes or 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]
- and others.

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 run the exact same R-script locally and

[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/

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.)

In order to get started with the examples in the following subsections, go to `https://aws.amazon.com/` and create an account. You will only be charged for the time you use an AWS service. However, even when using some cloud instances, several of AWS' cloud products offer a free tier to test and try out products. The following examples rely whenever possible on free-tier instances, otherwise I explicitly that running the example in the cloud will generate some costs.

6.2 Scaling up in the cloud

6.2.1 Scaling up with AWS EC2 and R/RStudio

One of the easiest ways to set up an AWS EC2 instance for R/RStudio is to use Louis Aslett's Amazon Machine Image (AMI)[7]. This way you do not need to install R/Rstudio yourself. Simply follow these five steps:

- Depending on the region in which you want to initiate your EC2 instance, click on the corresponding AMI link in `https://www.louisaslett.com/RStudio_AMI/`. For example, if you want to initiate the instance in Frankfurt click on ami-076abd591c4335092[8]. You will be automatically directed to the AWS page where you can select the type of EC2 instance you want to initiate. Per default the free tier T2.micro instance is selected (I recommend using this type of instance, if you simply want to try out the examples below).
- After selecting the instance type, click on "Review and Launch". On the opened page, select "Edit security groups". There should be one entry with SSH selected in the drop-down menu. Click on this drop-down menu and select HTTP (instead of SSH). Click again on "Review and Launch" to confirm the change.

[7] `https://www.louisaslett.com/RStudio_AMI/`

[8] `https://console.aws.amazon.com/ec2/home?region=eu-central-1#launchAmi=ami-076abd591c4335092`

- Then, click "Launch" to initiate the instance. From the pop-up concerning the key pair, select "Proceed without a key pair" from the drop-down menu and check the box below ("I acknowledge ..."). Click "Launch" to confirm. A page opens. Click on "View" instances to see all of your instances and their status. Wait until "Status check" is "2/2 checks passed" (you might want to refresh the instance overview or browser window).
- Click on the instance ID of your newly launched instance and copy the public IPv4 address, open a new browser window/tab, type in `http://`, paste the IP address, and hit enter (the address in your browser bar will be something like `http://3.66.120.150`; `http`, not `https`!) .
- You should see the login-interface to RStudio on your cloud instance. The username is `rstudio` and the password is the instance ID of your newly launched instance (it might take a while to load R/Rstudio). Once RStudio is loaded, you are ready to go.

NOTE: the instructions above help you set up your own EC2 instance with R/RStudio to run some example scripts and tryout R on EC2. For more serious/professional (long-term) usage of an EC2 instance, I strongly recommend to set it up manually and improve the security settings accordingly! The above set up will theoretically result in your instance being accessible for anyone in the Web (something you might want to avoid).

6.2.1.1 Parallelization with an EC2 instance

This short tutorial illustrates how to scale the computation up by running it on an AWS EC2 instance. Thereby, we build on the techniques discussed in the previous chapter. Note that our EC2 instance is a Linux machine. When running R on a Linux machine, there is sometimes 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). In some cases you might also have to install additional dependencies di-

rectly in Linux. Apart from that, using R via RStudio Server in the cloud looks/feels very similar if not identical as when using R/RStudio locally.

6.2.1.1.1 Preparatory steps

If your EC2 instance with RStudio-Server is not running yet, do the following. In the AWS console, navigate to EC2, select your EC2 instance (with RStudio Server installed) , and click on "Instance state/Start instance". You will have to wait until you see "2/2 checks passed". Then, open a new browser window, enter the address of your EC2/RStudio-Server instance (see above, e.g. `http://3.66.120.150`), and log in to RStudio. First, we need to install the `parallel` and `doSNOW` packages. In addition we will rely on the `stringr` package.

```
# install packages for parallelization
install.packages("parallel", "doSNOW", "stringr")
```

Once the installations have finished, you can load the packages and verify the number of cores available on your EC2 instance as follows. In case you have chosen the free tier T2.micro instance type when setting up your EC2 instance, you will see that you only have one core available. Do not worry. It is reasonable practice to test your parallelization script with a few iterations on a small machine before bringing out the big guns. The specialized packages we use for parallelization here do not mind if you have one or 32 cores, the same code runs on either machine (obviously not very fast with only one core).

```
# load packages
library(parallel)
library(doSNOW)

# verify no. of cores available
n_cores <- detectCores()
n_cores
```

Finally, we have to upload the data that we want to process as part of the parallelization task. To this end, in RStudio-Server, navigate

to the file explorer in the lower-right corner. The graphical user interface of a local RStudio installation and RStudio-Server is almost identical. However, you will find in the file explorer pane a "Upload"-button to transfer files from your local machine to the EC2 instance. In this demonstration, we will work with the previously introduced `marketing_data.csv` dataset. You can thus click on "Upload" and upload it to the current target directory (the home directory of RStudio-Server). As soon as the file is uploaded you can work with it as usual (as on the local RStudio installation). To keep things like in the local examples, use the file explorer to create a new `data` folder and move `marketing_data.csv` in this new folder.

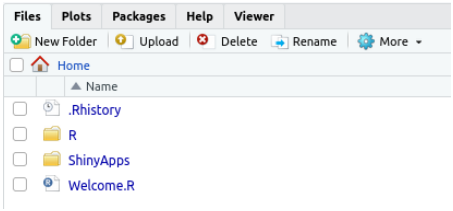


FIGURE 6.1: File explorer and Upload-button on Rstudio-Server.

To test if all is set up properly to run a in parallel on our EC2 instance, open a new R-script in RStudio-Server and copy/paste the preparatory steps and the simple parallelization example from Section 4.5 into the R-Script.

```
# PREPARATION ------------------------------

# packages
library(stringr)

# import data
marketing <- read.csv("data/marketing_data.csv")
# clean/prepare data
marketing$Income <- as.numeric(gsub("[[:punct:]]", "", marketing$Income))
marketing$days_customer <- as.Date(Sys.Date())- as.Date(marketing$Dt_Customer, "%m/%d/%y")
marketing$Dt_Customer <- NULL
```

```
# all sets of independent vars
indep <- names(marketing)[ c(2:19, 27,28)]
combinations_list <- lapply(1:length(indep),
                            function(x) combn(indep, x, simplify = FALSE))
combinations_list <- unlist(combinations_list, recursive = FALSE)
models <- lapply(combinations_list,
                 function(x) paste("Response ~", paste(x, collapse="+")))
```

6.2.1.1.2 *Test parallelized code*

Now, we can start testing the code on EC2 without registering the one core for cluster processing. This way `%dopart%` will automatically resort to running the code sequentially. Make sure to set `N` to 10 (or another small number) for this test.

```
# set cores for parallel processing
# ctemp <- makeCluster(ncores)
# registerDoSNOW(ctemp)

# prepare loop
N <- 10 # just for illustration, the actual code is N <- length(models)
# run loop in parallel
pseudo_Rsq <-
  foreach ( i = 1:N, .combine = c) %dopar% {
    # fit the logit model via maximum likelihood
    fit <- glm(models[[i]], data=marketing, family = binomial())
    # compute the proportion of deviance explained by the independent vars (~R^2)
    return(1-(fit$deviance/fit$null.deviance))
}
```

Once the test has run through successfully, we are ready to scale up and run the actual workload in parallel in the cloud.

6.2.1.2 Scale up and run in parallel

First, switch back to the AWS EC2 console and stop the instance by selecting the tick-mark in the corresponding row, and click on “In-

stance state/stop instance". Once the Instance state is "Stopped", click on "Actions/Instance settings/change instance type". You will be presented with a drop-down menu from which you can select the new instance type and confirm. The example bellow is based on selecting the `t2.2xlarge` (with 8 vCPUs and 32MB of RAM). Now you can start the instance again, log in to RStudio-Server (as above) and run the script again but this time with the following lines not commented out (in order to make use of all eight cores).

```
# set cores for parallel processing
ctemp <- makeCluster(ncores)
registerDoSNOW(ctemp)
```

In order to monitor the usage of computing resources on your instance, switch to the Terminal tab, enter `htop`, and hit enter. This will open the interactive process viewer called htop. With the default free tier T2.micro instance, you will again notice that only one core is available.

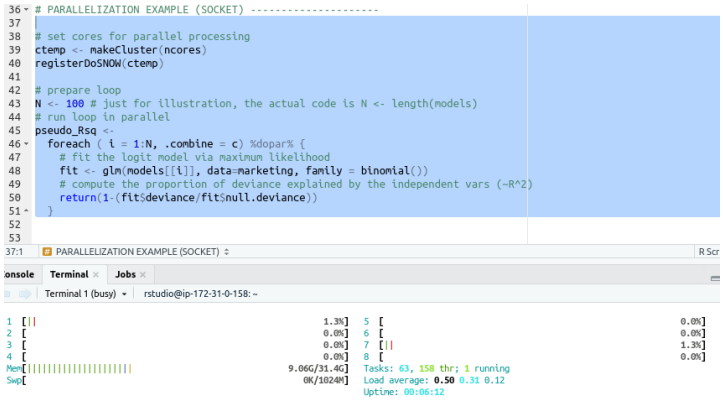


FIGURE 6.2: Monitor resources and proceses with htop.

6.2.2 Scaling up with GPUs

To start a ready-made EC2 instance with GPUs and RStudio installed, go to this service provided by RStudio on the AWS Marketplace: `https://aws.amazon.com/marketplace/pp/B0785SXYB2`. Click on Subscribe.

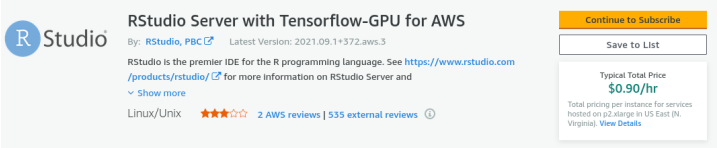


FIGURE 6.3: AWS Marketplace product provided by RStudio to run RStudio Server with Tensorflow-GPU on AWS EC2.

Then, click on Continue to Configuration and Continue to Launch. If you want to use the smallest/cheapest EC2 instance, select under "EC2 Instance Type" `g3s.xlarge`. If necessary, create a new key pair under 'Key Pair Settings', otherwise keep all the default settings as they are. Then, at the bottom, click on *Launch*. This will launch a new EC2 instance with a GPU and with RStudio server installed.[9]

Once the instance is launched go to the EC2 dashboard, click on the new instance, copy its public dns, open a new browser window and go to `http://<ec2_instance_public_dns>:8787`. You should see the RStudio server login. You can log in with username `rstudio-user` and the instance_id of your newly created instance as the password.[10]

6.3 GPUs on Google Colab

Google Colab provides a very easy way to run R code on GPUs from Google Cloud. All you need is a Google account. Open a new browser

[9] Note that you might not have the required vCPU limit to additionally launch such an instance. In that case you will get an error message `You have requested more vCPU capacity than your current vCPU limit of 0 allows for the instance bucket that the specified instance type belongs to. Please visit http://aws.amazon.com/contact-us/ec2-request to request an adjustment to this limit.`. Simply follow the indicated URL and send your request to increase the capacity. After your limit is adjusted, click again on Launch as per the procedure outlined above.

[10] It is strongly recommended to change the password afterwards. In o#rder to do that, click on "Tools" and select "shell". Then type "password" into the shell/terminal and enter the current password (the instance id) and then enter the new password, hit enter, and enter again the new password to confirm.

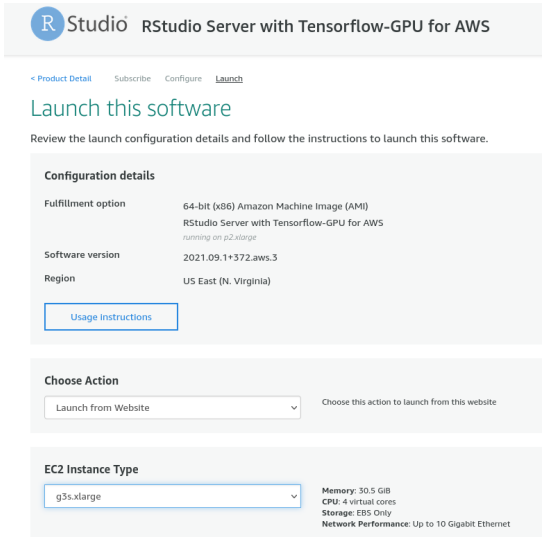


FIGURE 6.4: Launch RStudio Server with Tensorflow-GPU on AWS EC2.

window, go to `https://colab.to/r` and log in with your Google account if prompted to do so. Colab will open a Jupyter notebook[11] with an R runtime. Click on "Runtime/Change runtime type" and select in the drop-down menu under 'Hardware accelerator' the option 'GPU'.

Then, you can install the packages you need to work with GPU acceleration (e.g., `gpuR`, `keras` and `tensorflow`) and the code relying on GPU processing will be run on GPUs (or even TPUs[12]). Under the following link you find a Colab-notebook set up for running a simply image classification tutorial[13] with keras on TPUs: bit.ly/bda_colab[14].

[11] https://en.wikipedia.org/wiki/Project_Jupyter

[12] https://en.wikipedia.org/wiki/Tensor_Processing_Unit

[13] https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/tutorial_basic_classification/

[14] https://bit.ly/bda_colab

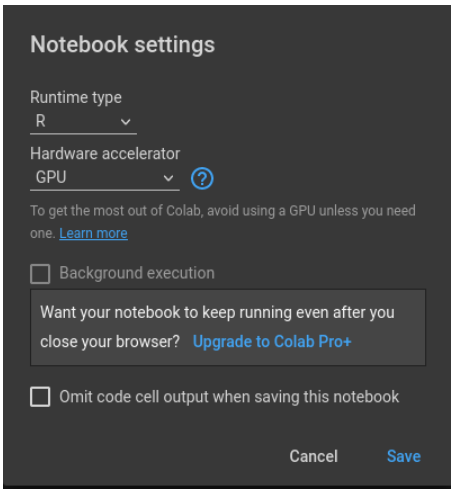


FIGURE 6.5: Colab notebook with R runtime and GPUs.

6.4 AWS EMR: MapReduce in the cloud

Many cloud computing providers offer specialized services for MapReduce tasks in the cloud. Here we look at a comparatively easy-to-use solution provided by AWS, called Elastic MapReduce (AWS EMR). It allows to set up a Hadoop cluster in the cloud within minutes and requires essentially no additional configuration if the cluster is being used for the kind of data analytics tasks discussed in this book.

Setting up a default AWS EMR cluster via the AWS console is straightforward. Simply go to `https://console.aws.amazon.com/elasticmapreduce/`, click on "Create cluster" and adjust the default selection of settings if necessary. Alternatively, we can set up an EMR cluster via the AWS command-line interface (CLI). In the following tutorials, we will work with AWS EMR via R/Rstudio (specifically, via the package `sparklyr`). Per default, RStudio is not part of the EMR cluster set up. However, AWS EMR offers a very flexible way to install/configure additional software on virtual EMR clusters via so-called "bootstrap" scripts. These scripts

can be shared on AWS S3 and used by others, which is what we do in the following cluster set up via the CLI.[15]

In order to run the cluster set up via AWS CLI shown below, you need an SSH key to later connect to the EMR cluster. If you do not have such an SSH key for AWS yet, follow these instructions to generate one: `https://docs.aws.amazon.com/cloudhsm/classic/userguide/generate_ssh_key.html`. In the example below, the key generated in this way is stored in a file called `sparklyr.pem`.[16]

The following command (`aws emr create-cluster`) initiates our EMR cluster with a specific set of options (all of these options can also be modified via the AWS console in the browser). `--applications Name=Hadoop Name=Spark Name=Hive Name=Pig Name=Tez Name=Ganglia` specifies which type of basic applications (that are essential to running different types of MapReduce tasks) should be installed on the cluster. Unless you really know what you are doing, do not change that setting. `--name "EMR 6.1 RStudio + sparklyr` simply specifies how the newly initiated cluster should be called (this name will then appear on your list of clusters in the AWS console). More relevant for what follows is the line specifying what type of virtual servers (EC2 instances) should be used as part of the cluster: `--instance-groups InstanceGroupType=MASTER,InstanceCount=1,InstanceType=m3.2xlarge` specifies that the one master node (the machine distributing tasks and coordinating the MapReduce procedure) is an instance of type `m3.2xlarge`; `InstanceGroupType=CORE,InstanceCount=2,InstanceType=m3.2xlarge` specifies that there are two slave nodes in this cluster, also of type `m1.medium`.[17] `--bootstrap-action Path=s3://aws-bigdata-blog/artifacts/aws-blog-emr-rstudio-sparklyr/rstudio_sparklyr_emr6.sh,Name="Install RStudio"` tells the set-up application to run the corresponding boot-

[15] Specifically, we will use the bootstrap script provided by the AWS Big Data Blog, which is stored here: s3://aws-bigdata-blog/artifacts/aws-blog-emr-rstudio-sparklyr/rstudio_sparklyr_emr6.sh

[16] If you simply copy-paste the CLI command below to set up an EMR cluster, make sure to name your key file also `sparklyr.pem`. Otherwise, make sure to change the part in the command referring to the key file accordingly.

[17] Working with one master node of type `m3.2xlarge` and two slave nodes of the same type only makes sense for test purposes. For an actual

strap script on the cluster in order to install the additional software (here RStudio).

Finally, there are two important aspects to note: First, in order to initiate the cluster in this way, you need to have an SSH key pair (for your EC2 instances) set up, which you then instruct the cluster to use with `KeyName=`. That is, `KeyName="sparklyr"` means that the user already has create an SSH key pair called `sparklyr` and that this is the key pair that will be used with the cluster nodes for SSH connections. Second, the `--region` argument defines in which AWS region the cluster should be created. Importantly, in this particular case, the bootstrap script used to install RStudio on the cluster is stored in the `us-east-1` region, hence we need to set up the cluster also in this region `--region us-east-1` (otherwise the set up will fail as the set-up application will not find the bootstrap script and terminate with an error!).

```
aws emr create-cluster \
--release-label emr-6.1.0 \
--applications Name=Hadoop Name=Spark Name=Hive Name=Pig Name=Tez Name=Ganglia   \
--name "EMR 6.1 RStudio + sparklyr"  \
--service-role EMR_DefaultRole \
--instance-groups InstanceGroupType=MASTER,InstanceCount=1,InstanceType=m3.2xlarge \
InstanceGroupType=CORE,InstanceCount=2,InstanceType=m3.2xlarge \
--bootstrap-action Path='s3://aws-bigdata-blog/artifacts/aws-blog-emr-rstudio-sparklyr/rstudio
--configurations '[{"Classification":"spark","Properties":{"maximizeResourceAllocation":"true"
--region us-east-1
```

Setting up this cluster with all the additional software and configurations from the bootstrap script will take around 40 minutes. You can always follow the progress in the AWS console on . Once the cluster is ready, you will see something like this:

In order to access RStudio on the EMR cluster's master node via a secure SSH connection, follow these steps:

- First, follow the prerequisites to connect to EMR via SSH: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-connect-ssh-prereqs.html.

FIGURE 6.6: AWS EMR console indicating the successful set up of the EMR cluster

- Then initiate the SSH tunnel to the EMR cluster as instructed here: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-ssh-tunnel.html.
- Protect your key-file (`sparklyr.pem`) by navigating to the location of the key-file on your computer in the terminal and run `chmod 600 sparklyr.pem` before connecting. Also make sure your IP address is still the one you have entered in the previous step (you can check your current IP address by visiting https://whatismyipaddress.com/).
- In a browser tab, navigate to the AWS EMR console, click on the newly created cluster and copy the "Master public DNS". In the terminal, connect to the EMR cluster via SSH by running `ssh -i sparklyr.pem -ND 8157 hadoop@master-node-dns` (if you have protected the key-file as super user, i.e. `sudo chmod`, you will need to use `sudo ssh` here; make sure to replace `master-node-dns` with the actual dns copied from the AWS EMR console). The terminal will be busy but you won't see any output (if all goes well).
- In your Firefox browser, install the FoxyProxy add on[18]. Follow these instructions to set up the proxy via FoxyProxy: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-connect-master-node-proxy.html.
- Select the newly created Socks5 proxy in FoxyProxy.
- Go to http://localhost:8787/ and enter with username `hadoop` and password `hadoop`.

Now you can run `sparklyr` on the AWS EMR cluster. After finishing working with the cluster, make sure to terminate it via the EMR con-

[18] https://addons.mozilla.org/en-US/firefox/addon/foxyproxy-standard/

sole. This will shot down all the EC2 instances that are part of the cluster (and hence AWS will stop charging you for this).

Once you have connected and logged in to RStudio on the EMR cluster's master node, you can connect the the Rstudio session to the Spark cluster as follows.

```
##
## Attaching package: 'sparklyr'
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
# load packages
library(sparklyr)
# connect rstudio session to cluster
sc <- spark_connect(master = "yarn")
```

After using the EMR Spark cluster, make sure to terminate the cluster in the AWS EMR console to avoid additional charges. This automatically terminates all the EC2 machines linked to the cluster.

Part III

Applied Big Data Analytics

Platform: Software and Computing Resources

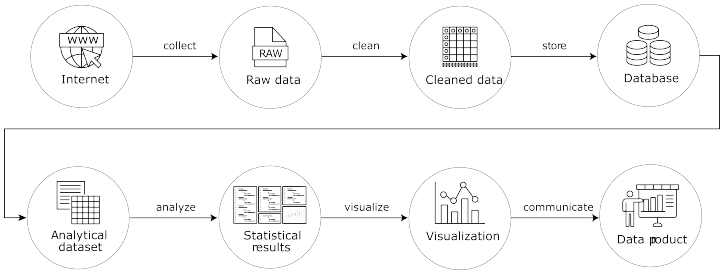
7

Forms of Big Data and the Data Pipeline

7.1 Unstructured, semi-structured, structured data

7.2 Data pipelines: a systematic approach to processing big data

Data (science) pipeline



8

Data Collection and Data Storage

8.1 Gathering and compilation of raw data

8.1.1 NYC taxi data

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 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).

The raw data consists of several monthly CSV-files and can be downloaded via the TLC's website[1]. 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
###################################
```

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

```
# 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 -----------

# 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)

}
```

8.2 Data import and 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 <- "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 noticeable 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 alloca-

tion. 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)


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

```
## 1.72 GB
```

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

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

```
## 4.04 MB
```

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

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

```
## 1.72 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   6535541  349.1   10692948  571.1  10692948
## Vcells 169834397 1295.8  409858047 3127.0 284501290
##          (Mb)
## Ncells  571.1
## Vcells 2170.6
```

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)
```

```
##
## Attaching package: 'data.table'

## The following object is masked from 'package:pryr':
##
##     address
```

```
# 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.
## Input contains no \n. Taking this to be a filename to open
## [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 00:00.066 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 (  5%) 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
##   0.062s ( 94%) Reading 30 chunks (0 swept) of 0.984MB (each chunk 11225 rows) using 6 threads
##             +             0.018s    (   27%)   Parse   to   row-
major thread buffers (grown 0 times)
##    +    0.028s ( 43%) Transpose
##    +    0.016s ( 24%) Waiting
```

```
##        0.000s  (    0%)  Rereading  0  columns  due  to  out-of-
sample type exceptions
##    0.066s        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   6593725  352.2   10692948  571.1  10692948
## Vcells 166820133 1272.8  409858047 3127.0 284501290
##          (Mb)
## Ncells  571.1
## Vcells 2170.6
```

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

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

```
## 35.8 MB
```

8.3 Efficient local data storage

In this section, 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)

8.3.1 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).

8.3.2 Efficient data access: indices and joins in SQLite

So far we have only had a look at the very basics of writing SQL code. Let us now further explore SQLite as an easy-to-use and easy-to-set-up relational database solution. In a second step we then look at how to connect to a local SQLite database from within R. First, we switch to the Terminal tab in RStudio, set up a new database called `air.sqlite`, and import the csv-file `flights.csv` (used in previous chapters) as a first table.

```
# switch to data directory
cd data
# create database and run sqlite
sqlite3 air.sqlite
```

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

We check if everything worked out well via the `.tables` and `.schema` commands.

```
.tables
.schema flights
```

In `flights`, each row describes a flight (the day it took place, its ori-

gin, 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. First 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`).
- `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 num-

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

TABLE 8.1: 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

ber. 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'
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.

TABLE 8.2: 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

```
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);
```

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,
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.

8.4 Connecting R to RDBMS

The R-package `RSQLite` embeds SQLite in R. That is, it provides functions that allow us to use SQLite directly from within R. You will see that the combinaiton of SQLite with R is a simple but very practical approach to work with very efficiently (and locally) stored data sets. In the following example, we explore how `RSQLite` can be used to set up and query the `air.sqlite` shown in the example above.

8.4.1 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")
```

8.4.2 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)
```

8.4.3 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
```

When done working with the database, we close the connection to the database with `dbDisconnect(con)`.

8.5 Cloud solutions for (big) data storage

As outlined in the previous section, RDBMSs are a very practical tool to store the structured data of an analytics project locally in a database. A local SQLite database can easily be set up and accessed via R, allowing to write the whole data pipeline from data gathering to filtering, aggregating and finally analyzing in R. In contrast to directly working with CSV files, using SQLite has the advantage of organizing the data access much more efficiently in terms of RAM. Only the final result of a query is really loaded fully into R's memory.

If mass storage space is too sparse or if RAM is nevertheless not sufficient, even when organizing data access via SQLite, several cloud solutions come to the rescue. Although you could also rent a traditional web server and host a SQL database there, this is usually not worth the while for a data analytics project. In the next section we thus look at three important cases of how to store data as part of an analytics project: *RDBMS in the cloud*, a serverless *data warehouse* solution for large datasets called *Google BigQuery*, and a simple storage service to use as a *data lake* called *AWS S3*. All of these solutions are discussed from a data analytics perspective, and for all of these solutions we will look at how to make use of them from within R.

8.5.1 Easy-to-use RDBMS in the cloud: AWS RDS

Once we have set up RStudio Server on an EC2 instance, we can run the SQLite examples shown above 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 (e.g., via RStudio Server).

AWS' Relational Database Service (RDS) provides an easy way to set up and run a SQL database in the cloud. The great advantage for users new to RDBMS/SQL is that you do not have to manually set up a server

(e.g. a EC2 instance) and install/configure the SQL server. Instead you can directly set up a fully functioning relational database in the cloud.

In a first step, open the AWS console and search for/select "RDS" in the search bar. Then, click on "Create database" in the lower part of the landing page.

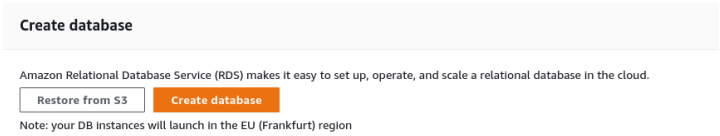


FIGURE 8.1: Create a managed relational database on AWS RDS.

On the next page, select "Easy create", "MySQL", and the "Free tier" DB instance size. Further down you will have to set the database instance identifier the user name and a password.

Once the database instance is ready, you will see it in the databases overview. Click on the DB identifier (the name of your database shown in the list of databases) and click on modify (button in the upper-right corner). In the "Connectivity" panel under "Additional configuration", select *Publicly accessible* (this is necessary to interact with the DB from your local machine), and save the settings. Back on the overview page of your database, under "Connectivity & security", click on the link under the VPC security groups, scroll down and select the "Inbound rules" tab. Edit the inbound rule to allow any IP4 inbound traffic.[4]

Now we can connect to the instance via the `RMySQL` package. Before loading data, we first have to initiate a new database (in contrast, this is done automatically when connecting to a SQLite database).

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

[4] Note that this is not generally recommendable. Only do this to get familiar with the service and to test some code.

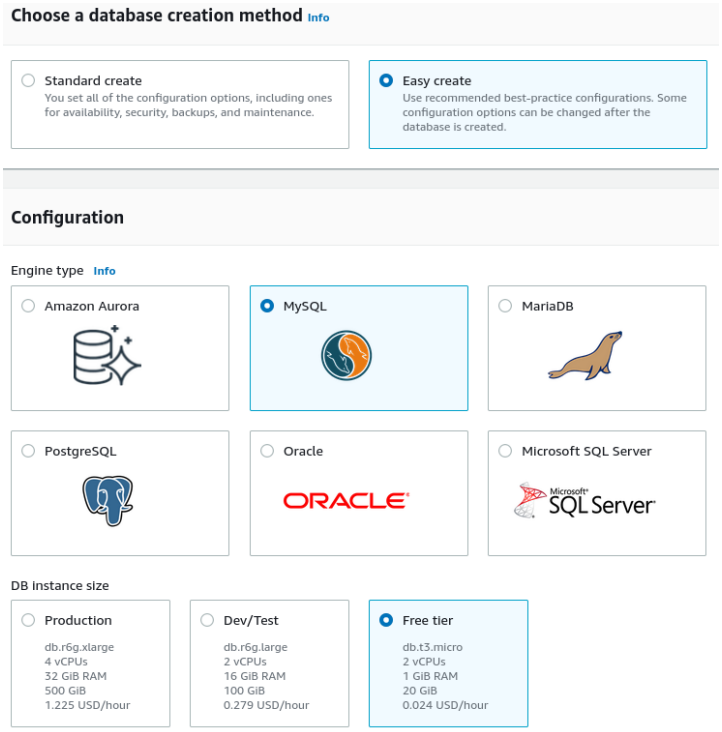


FIGURE 8.2: Easy creation of a RDS MySQL DB.

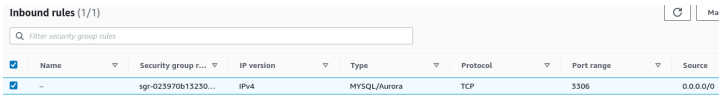


FIGURE 8.3: Allow all IP4 inbound traffic (set Source to `0.0.0.0/0`).

```
# fix vars
RDS_ENDPOINT <- "MY-ENDPOINT" # replace this with the Endpoint shown in the AWS RDS console
PW <- "MY-PW" # replace this with the password you have set when initiating the RDS DB on AWS

# connect to DB
con_rds <- dbConnect(RMySQL::MySQL(),
                     host=RDS_ENDPOINT,
                     port=3306,
                     username="admin",
```

```
                password=PW)

# initiate a new database on the MySQL RDS instance
dbSendQuery(con_rds, "CREATE DATABASE air")

# disconnect and re-connect directly to the new DB
dbDisconnect(con_rds)
con_rds <- dbConnect(RMySQL::MySQL(),
                host=RDS_ENDPOINT,
                port=3306,
                username="admin",
                dbname="air",
                password=PW)
```

`RMySQL` and `RSQLite` are both building on the `DBI` package, which generalizes how we can interact with SQL-type databases via R. This makes it straightforward to apply what we have learned so far by interacting with our local SQLite database to interactions with other databases. As soon as the connection to the new database is established, we can essentially use the same R functions as above to create new tables and import data.

```
# 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_rds, "flights", flights)
dbWriteTable(con_rds, "airports", airports)
dbWriteTable(con_rds, "carriers", carriers)
```

Finally, we can query our RDS MySQL database on AWS.

```
# 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_rds, delay_query)
delays_df
```

8.5.2 Database server in the cloud: MariaDB on an EC2 instance

Working with an SQL database in the cloud via AWS RDS is probably sufficient for most simple use cases. However, for some projects you might want to have more flexibility regarding settings and configurations. A practical solution to this is to set up "manually" a SQL-server on an EC2 instance (and then work with it 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. 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.[5]

8.5.2.1 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
```

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').

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

```
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.[6]

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

[6] 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[7].

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

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 data set. 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).

9.1 '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 data set, 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 data set (the data structure) and some metadata describing the data set. 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 data set 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`.

9.1.1 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`-data set as an example. When importing data via the `ff` package, we first have to set up a directory where `ff` can store the partitioned data set (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=0.404sec  ffdf-
write=0.055sec
## read.table.ffdf 100001..200000 (100000)  csv-read=0.372sec ffdf-
write=0.034sec
## read.table.ffdf 200001..300000 (100000)  csv-read=0.374sec ffdf-
write=0.033sec
## read.table.ffdf 300001..336776 (36776)  csv-read=0.149sec ffdf-
write=0.02sec
##  csv-read=1.299sec  ffdf-write=0.142sec  TOTAL=1.441sec

## -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`.

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

```
## [1] "clone10c9b3c430ec8.ff" "clone10c9b4977164.ff"
## [3] "clone10c9b56908fe0.ff" "clone10c9b77bbf204.ff"
## [5] "clone18ff55474d98.ff"  "clone18ff56cbf21c.ff"
```

9.1.2 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]

```
# 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 data set.

```
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
```

[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].

```
## 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
```

9.2 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)

9.2.1 Data Preparation with `ff`

9.2.1.1 Set up

The following examples are based on Walkowiak (2016), Chapter 3. You can download the original data sets 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 data sets will be stored.

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

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

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

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

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

9.2.1.2 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()
```

```
## 1.72 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=0.525sec  ffdf-
write=0.087sec
## read.table.ffdf 100001..200000 (100000)  csv-read=0.582sec ffdf-
write=0.074sec
## read.table.ffdf 200001..300000 (100000)  csv-read=0.555sec ffdf-
write=0.064sec
## read.table.ffdf 300001..400000 (100000)  csv-read=0.567sec ffdf-
write=0.075sec
## read.table.ffdf 400001..500000 (100000)  csv-read=0.568sec ffdf-
write=0.073sec
## read.table.ffdf 500001..600000 (100000)  csv-read=0.586sec ffdf-
write=0.064sec
## read.table.ffdf 600001..700000 (100000)  csv-read=0.575sec ffdf-
write=0.063sec
## read.table.ffdf 700001..800000 (100000)  csv-read=0.567sec ffdf-
write=0.071sec
## read.table.ffdf 800001..900000 (100000)  csv-read=0.544sec ffdf-
write=0.066sec
## read.table.ffdf 900001..951111 (51111)  csv-read=0.288sec ffdf-
write=0.04sec
##  csv-read=5.357sec  ffdf-write=0.677sec  TOTAL=6.034sec
```

```
##    user  system elapsed
##   5.574   0.540   6.036
```

```
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=0.005sec  ffdf-
write=0.004sec
##  csv-read=0.005sec  ffdf-write=0.004sec  TOTAL=0.009sec
```

```
##    user  system elapsed
##   0.006   0.005   0.010
```

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

```
## 1.72 GB
```

Comparison with `read.table`

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

```
##    user  system elapsed
##   4.991   0.324   5.238
```

```
gc()
```

```
##             used (Mb) gc trigger   (Mb)  max used
## Ncells   6758389  361   10692948  571.1  10692948
## Vcells 185578317 1416  409858047 3127.0 291430251
##          (Mb)
## Ncells  571.1
## Vcells 2223.5
```

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

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

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

```
## 1.86 GB
```

9.2.1.3 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 data set 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
```

9.2.1.4 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 data sets. That is, we want to create a new data set 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 data sets 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 data sets 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"))
```

```
## 777 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"))
```

```
## 161 MB
```

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

9.2.1.5 Subsetting

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

```
mem_used()
```

```
## 2.02 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()
```

```
## 2.02 GB
```

9.2.1.6 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   6776693  362   10692948  571.1  10692948
## Vcells 205694964 1569  409858047 3127.0 291430251
##          (Mb)
## Ncells  571.1
## Vcells 2223.5
```

```
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` data set 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 data set via `ff` and then store the (often much smaller)

analytic data set 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")
```

10

Descriptive Statistics and Aggregation

10.1 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.

10.1.1 Data aggregation with chunked data files

In this tutorial we explore the world of New York's famous Yellow Caps. In in a first step, 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.[2]

[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.

[2] 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.

10.1.1.1 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",
                 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"
```

10.1.1.2 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)})
```

```
## 2022-07-29 22:11:51, calculating split sizes

## 2022-07-29 22:11:51, building up split locations

## 2022-07-29 22:11:51, working on split 1/2, extracting data in RAM of 1 split elements, totalling

## 2022-07-29 22:11:52, ... applying FUN to selected data

## 2022-07-29 22:11:52, ... appending result to the output ffdf

## 2022-07-29 22:11:52, working on split 2/2, extracting data in RAM of 3 split elements, totalling

## 2022-07-29 22:11:52, ... applying FUN to selected data

## 2022-07-29 22:11:52, ... 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)})
```

```
## 2022-07-29 22:11:52, calculating split sizes

## 2022-07-29 22:11:52, building up split locations

## 2022-07-29 22:11:52, working on split 1/2, extracting data in RAM of 1 split elements, totalling

## 2022-07-29 22:11:53, ... applying FUN to selected data

## 2022-07-29 22:11:53, ... appending result to the output ffdf

## 2022-07-29 22:11:53, working on split 2/2, extracting data in RAM of 3 split elements, totalling

## 2022-07-29 22:11:53, ... applying FUN to selected data

## 2022-07-29 22:11:54, ... 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
```

10.1.2 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.

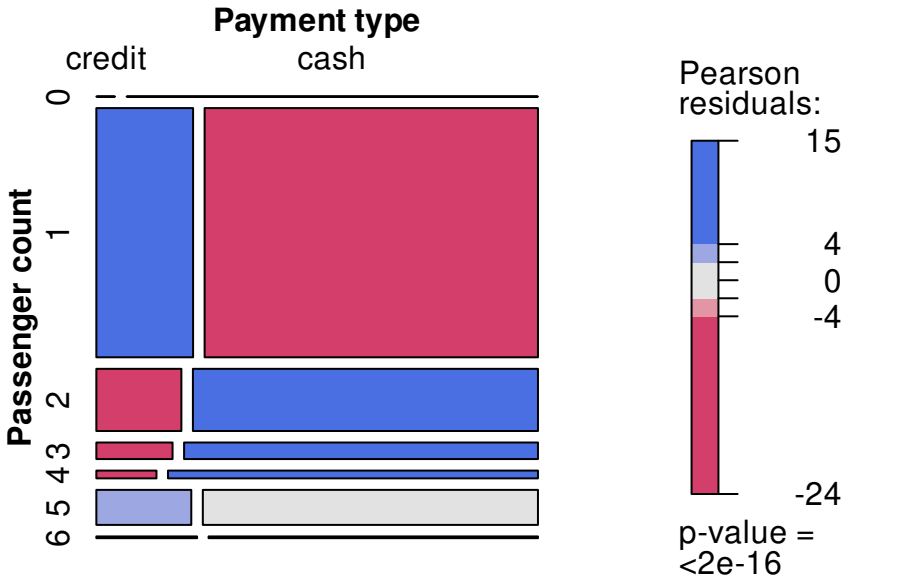
```
# install.packages(vcd)
# load package for mosaic plot
library(vcd)
```

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

```
##
## Attaching package: 'grid'
```

```
## The following object is masked from 'package:ff':
##
##     pattern
```

```
# 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.

10.2 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.

10.2.0.1 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)
```

10.2.0.2 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.

10.2.0.3 data.table-syntax for 'split-apply-combine' operations

With the []-syntax we index/subset usual data.frame objects in R. When working with data.tables, much more can be done in the step of 'sub-setting' the frame.[3]

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
```

[3] 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
```

11

(Big) Data Visualization

11.1 Data exploration with `ggplot2`

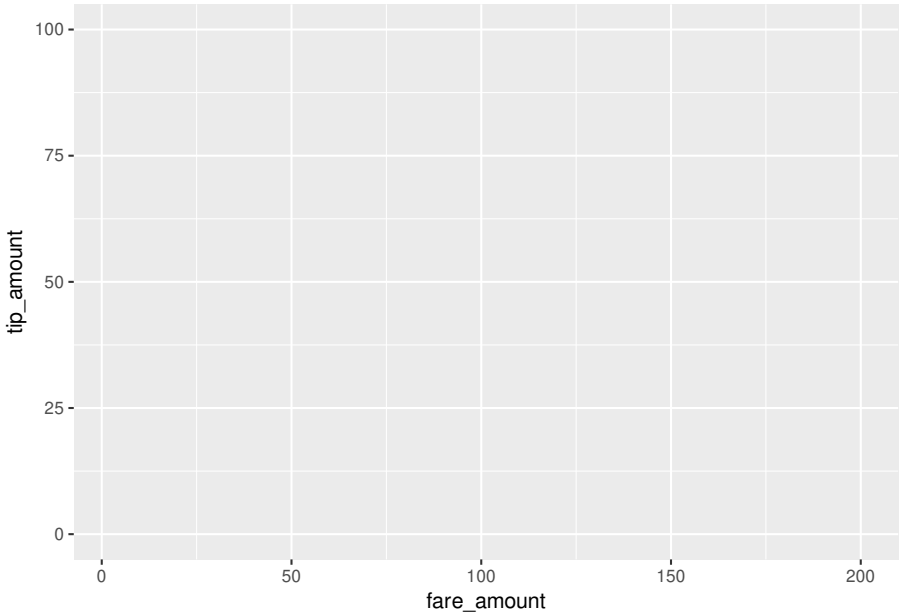
In this tutorial we will work with the TLC data used in the data aggregation session. The raw data consists of several monthly CSV-files and can be downloaded via the TLC's website[1]. Again, we work only with the first million observations.

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`.

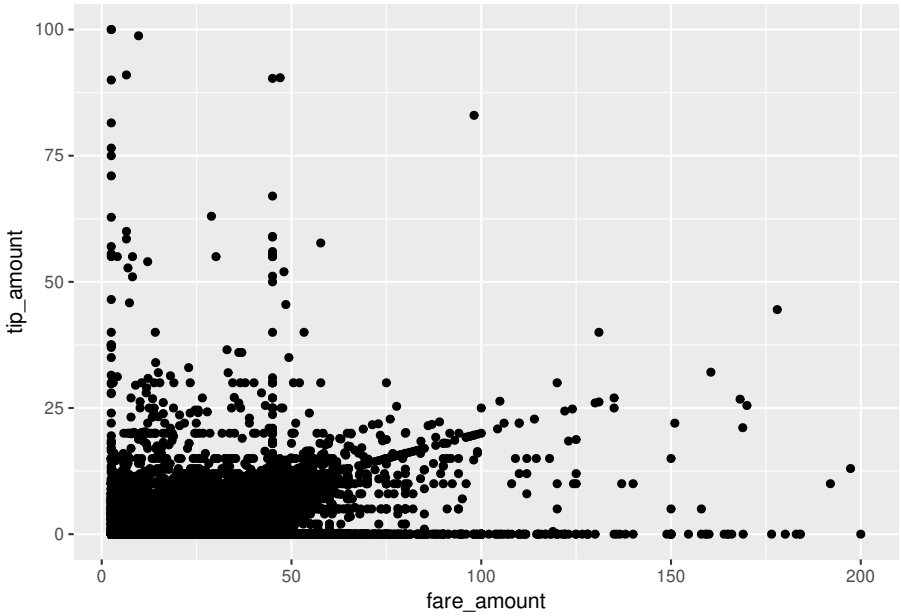
```
# load packages
library(ggplot2)

# 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



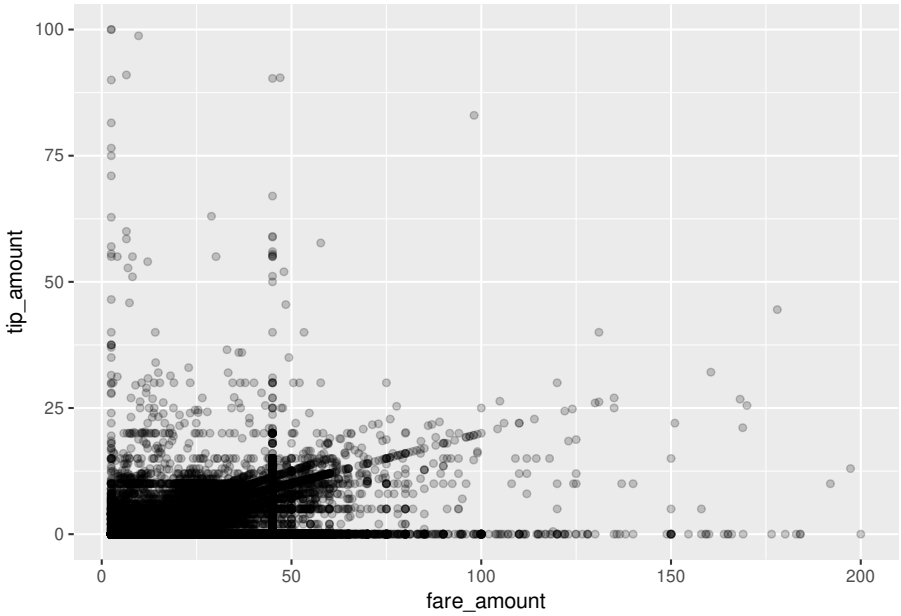
Now we visualize the co-distribution of the two variables with a simple scatter-plot.

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



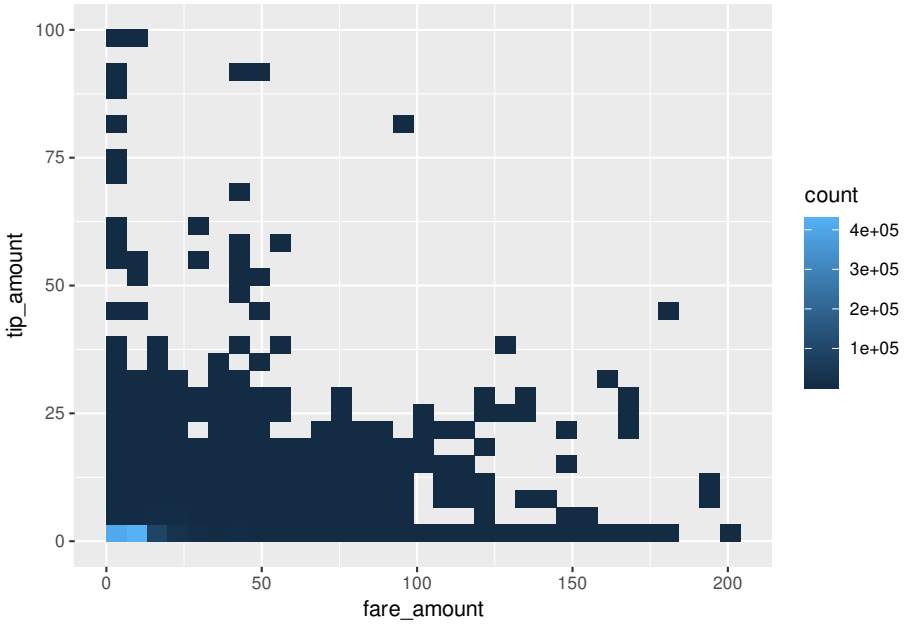
Note that this took quite a while, as R had to literally plot one million dots on the canvas. Moreover many dots fall within the same area, making it impossible to recognize how much mass there actually is. This is typical for visualization exercises with large data sets. One way to improve this, is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

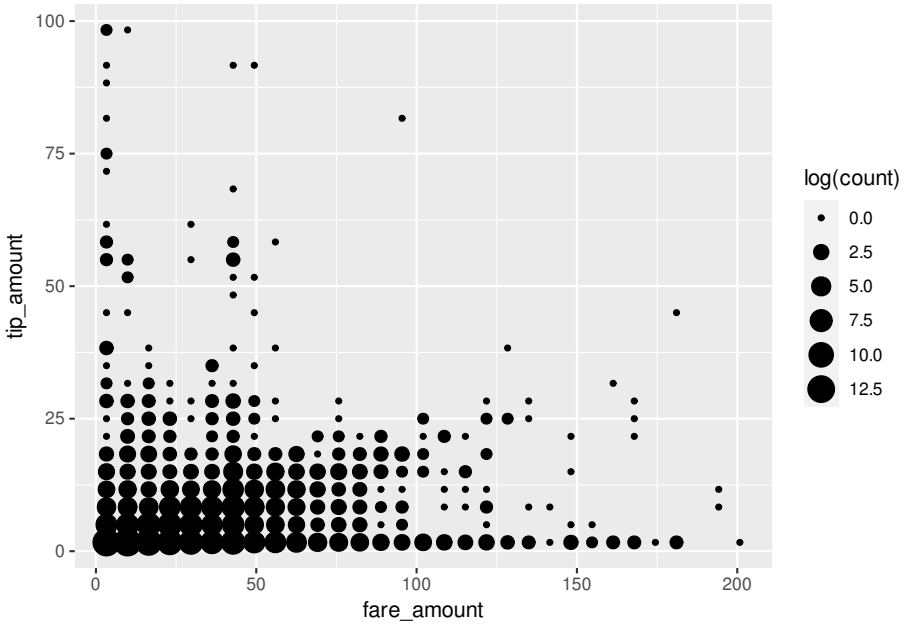
```
# 2-dimensional bins
taxiplot +
     geom_bin2d()
```



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.

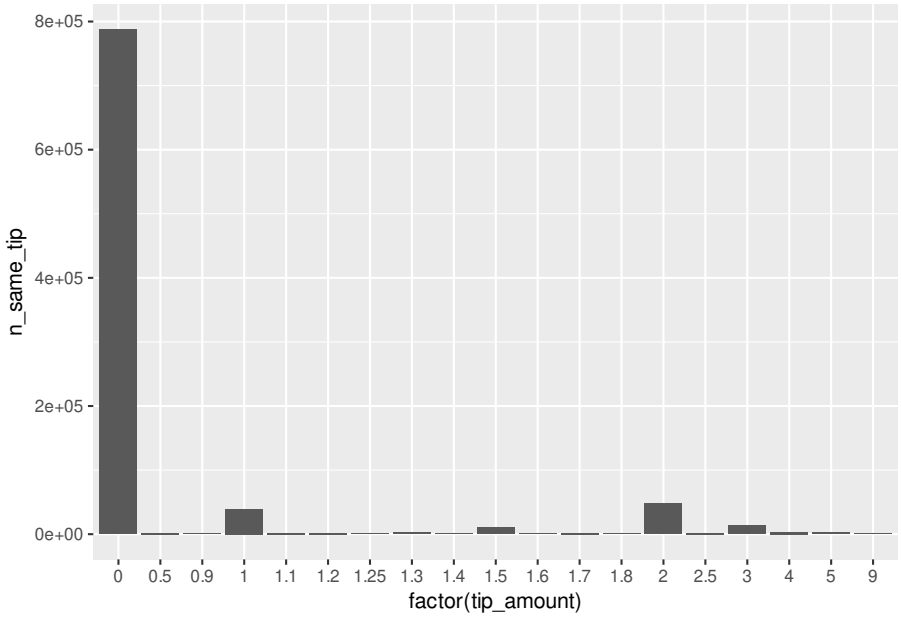
```
# 2-dimensional bins
taxiplot +
     stat_bin_2d(geom="point",
                 mapping= aes(size = log(..count..))) +
     guides(fill = FALSE)
```

```
## Warning: `guides(<scale> = FALSE)` is deprecated.
## Please use `guides(<scale> = "none")` instead.
```



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.

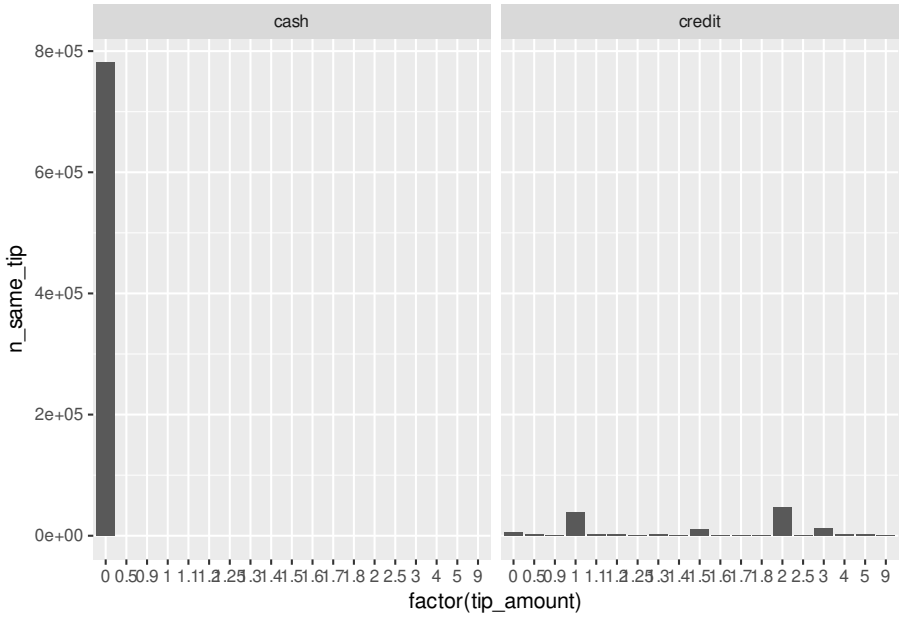
```
# compute frequency of per tip amount and payment method
taxi[, n_same_tip:= .N, by= c("tip_amount", "payment_type")]
frequencies <- unique(taxi[payment_type %in% c("credit", "cash"),
                           c("n_same_tip", "tip_amount", "payment_type")][order(n_same_tip, de


# 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")
```

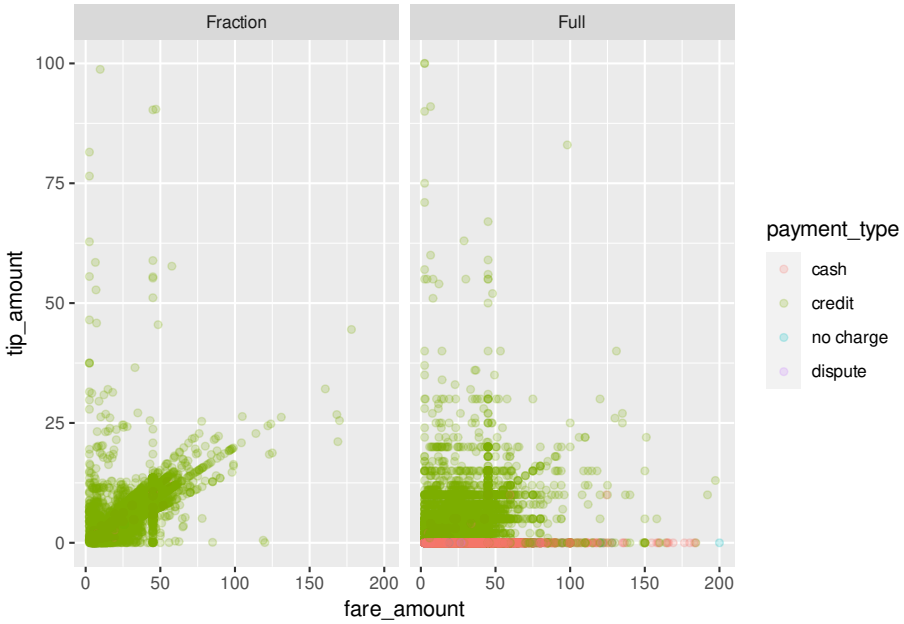
[2] Or, could there be another explanation for this pattern in the data?



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

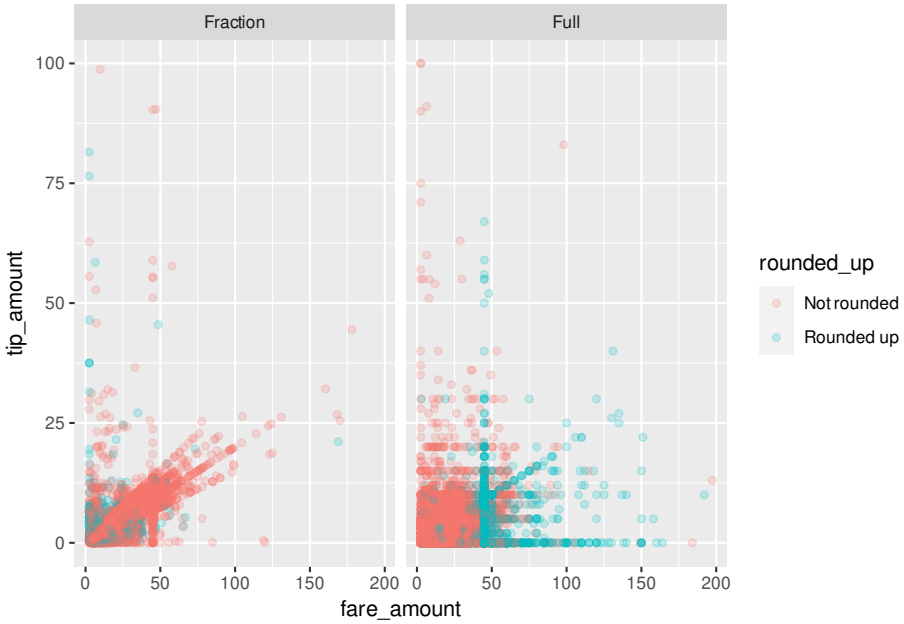
Let's try to tease this information out of the initial points plot. Trips paid in cash are often not tipped, we thus should indicate the payment method. Moreover, tips paid in full dollar amounts might indicate a habit.

```
# 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.

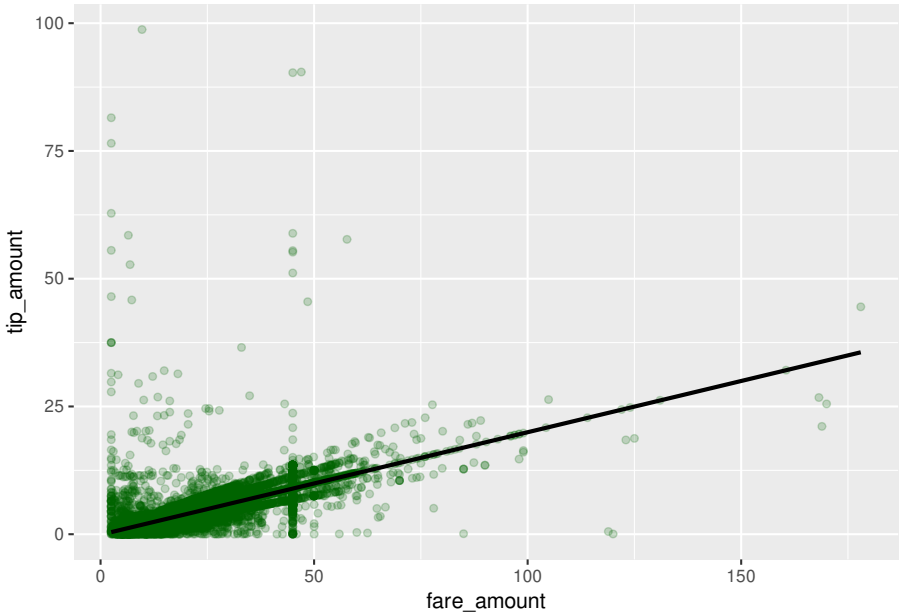
```
taxi[, rounded_up := ifelse(fare_amount + tip_amount == round(fare_amount + tip_amount, 0),
                            "Rounded up",
                            "Not rounded")]
# extended x/y plot
taxiplot +
     geom_point(data= taxi[payment_type == "credit"],
                alpha=0.2, aes(color=rounded_up))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

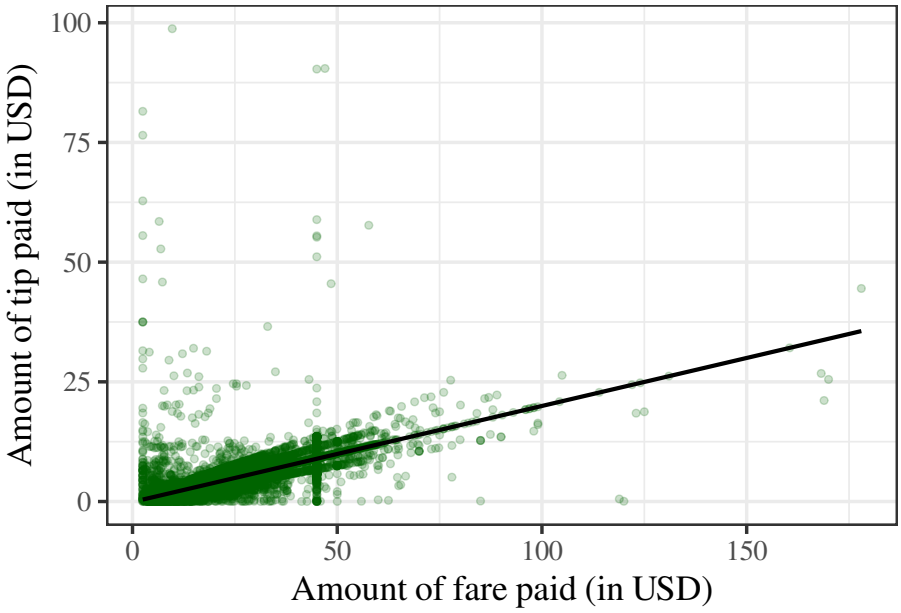
```
modelplot <- ggplot(data= taxi[payment_type == "credit" & dollar_paid == "Fraction" & 0 < tip_
                    aes(x = fare_amount, y = tip_amount))
modelplot +
     geom_point(alpha=0.2, colour="darkgreen") +
     geom_smooth(method = "lm", colour = "black") +
```

[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'
```



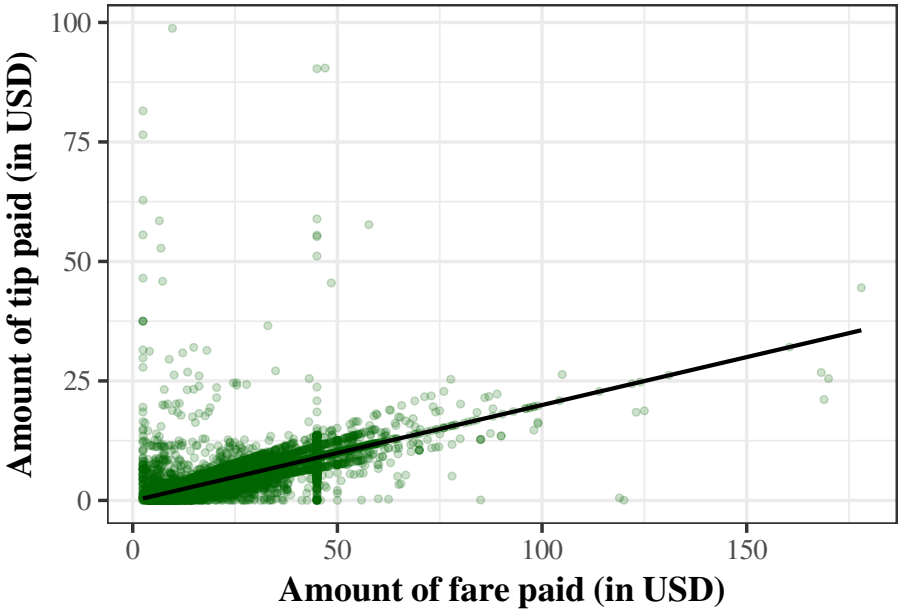
11.2 Excursus: modify and create themes

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'
```



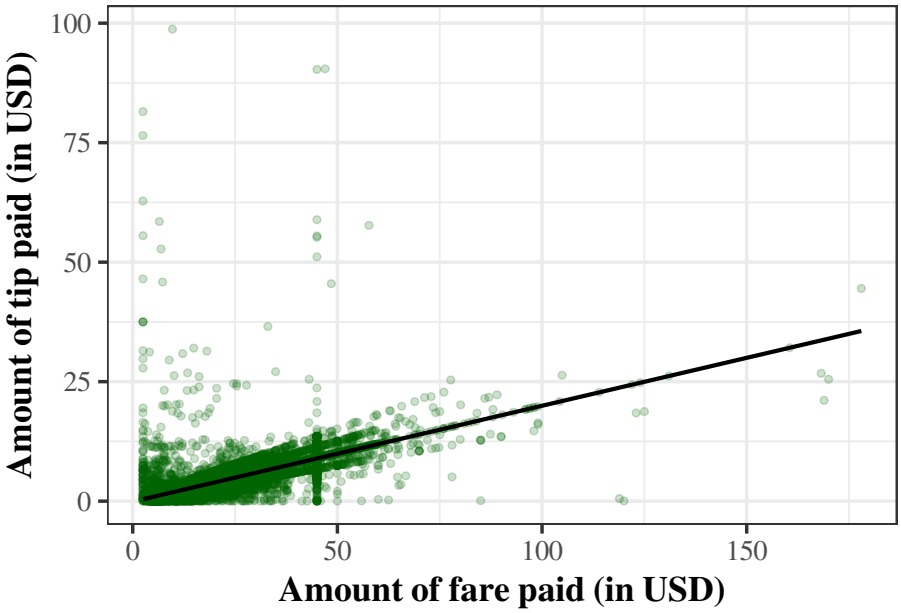
There is a large list of plot design aspects that can be modified in this way (see `?theme()` for details).

11.2.1 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
```

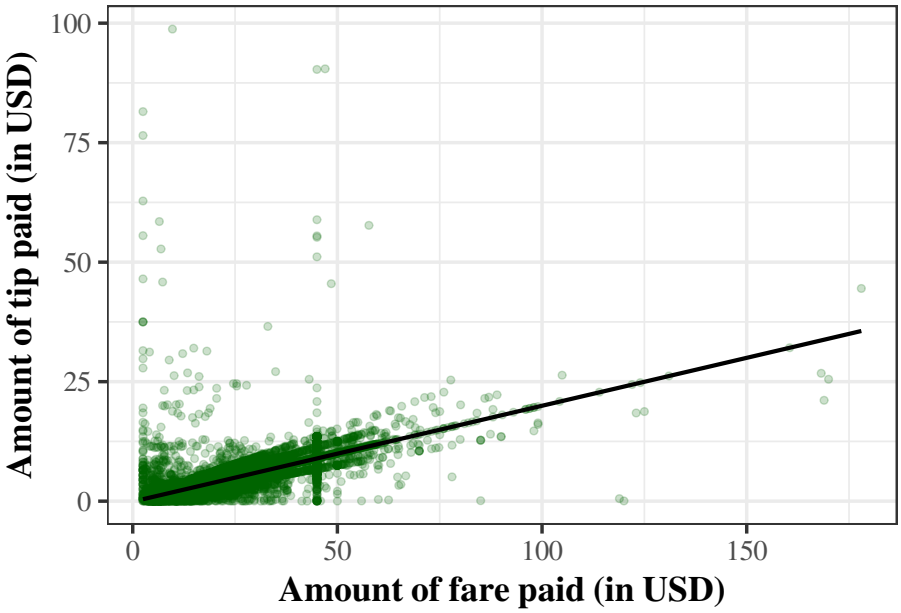
```
## `geom_smooth()` using formula 'y ~ x'
```



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

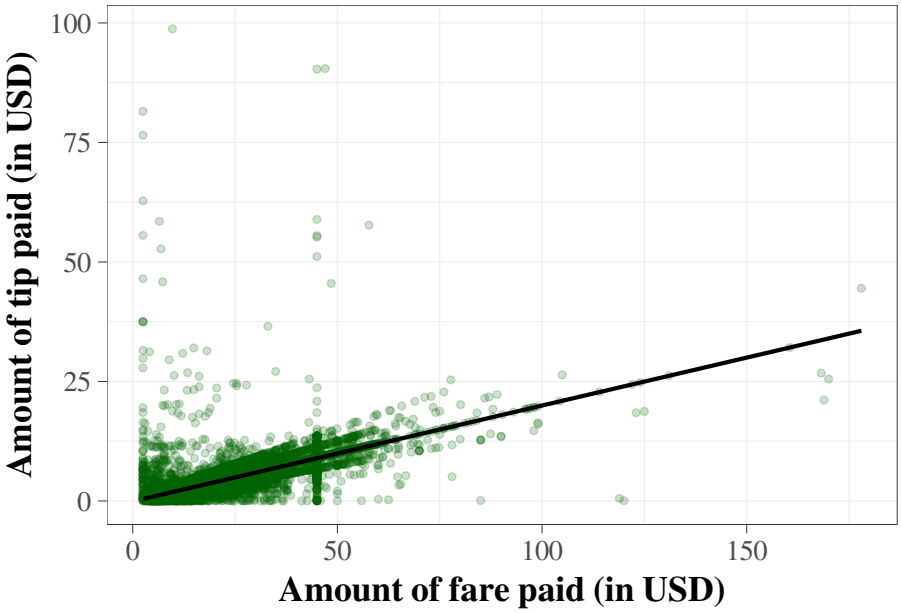
with `+ theme_my_serif` to the plot (no parentheses). In practice this is the most simple approach and it provides all the functionality you need in order to apply your own 'theme' to each of your plots.

11.2.2 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'
```



11.3 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.

11.3.1 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)
```

11.3.2 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[base::sample(1: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.

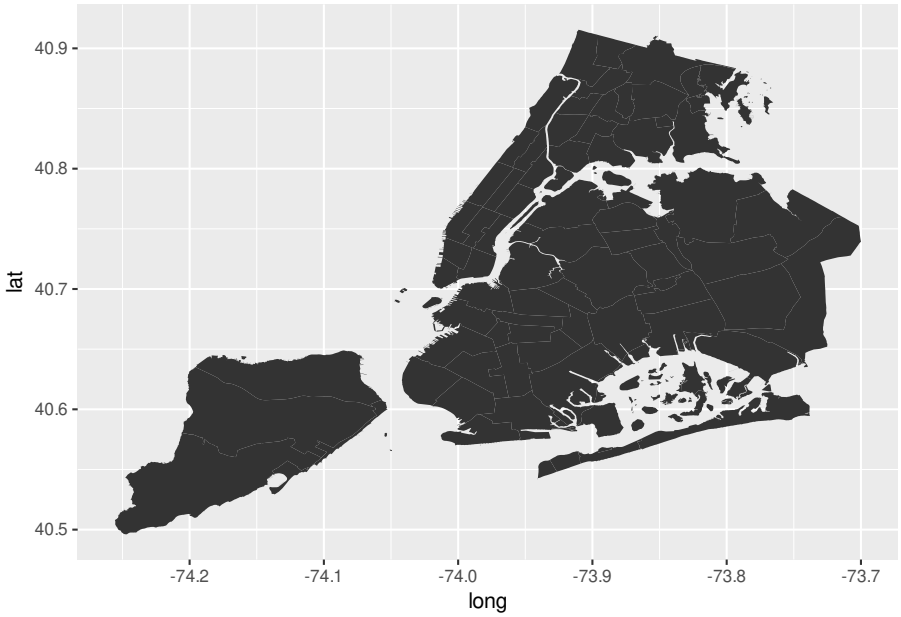
```
taxi_trips$start_time <- hour(taxi_trips$pickup_time)
```

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", '
```

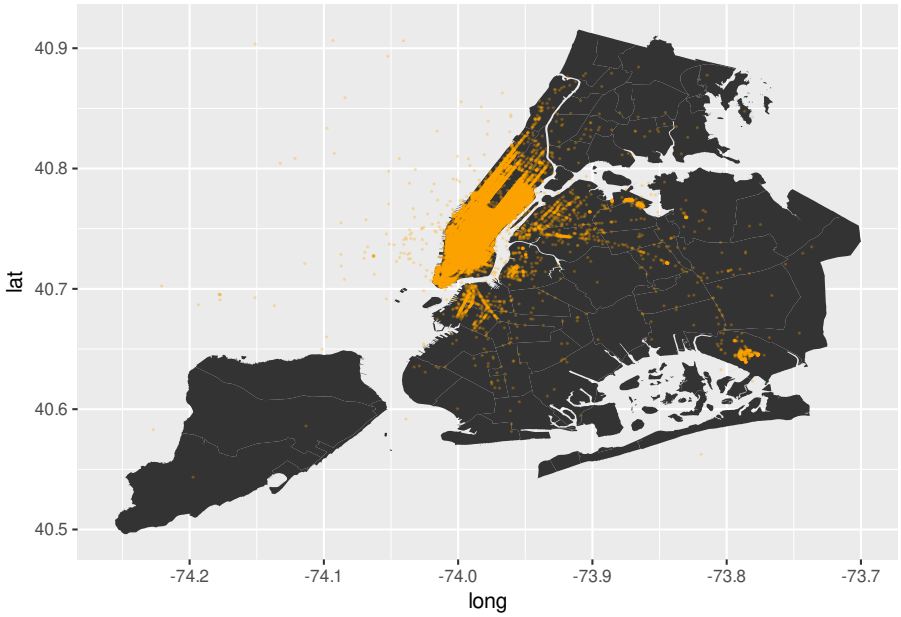
We initiate the plot by first setting up the canvas with our taxi trips data. Then, we add the map as a first layer.

```
# 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
```



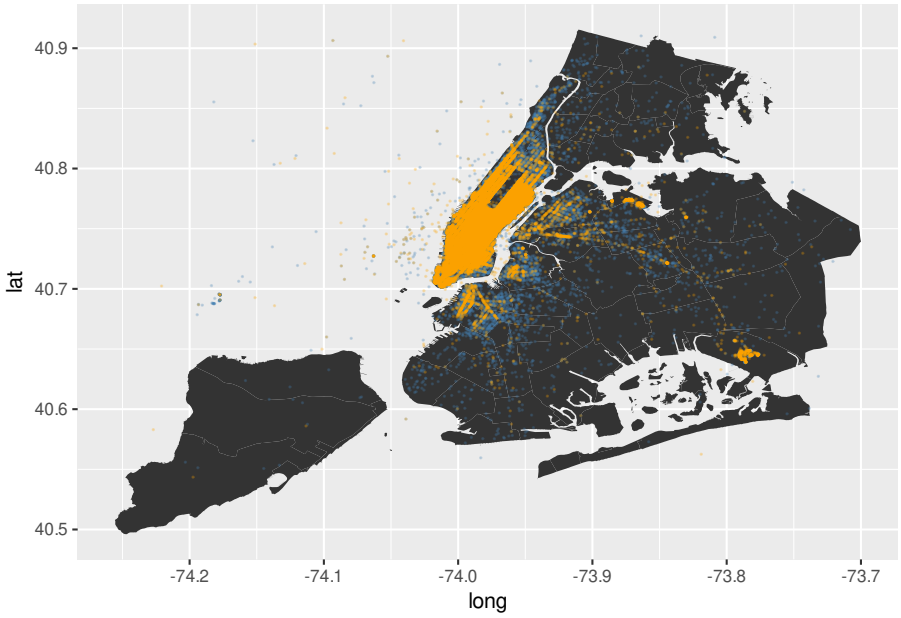
Now we can start adding the pick-up and drop-off locations of cap trips.

```
# 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.

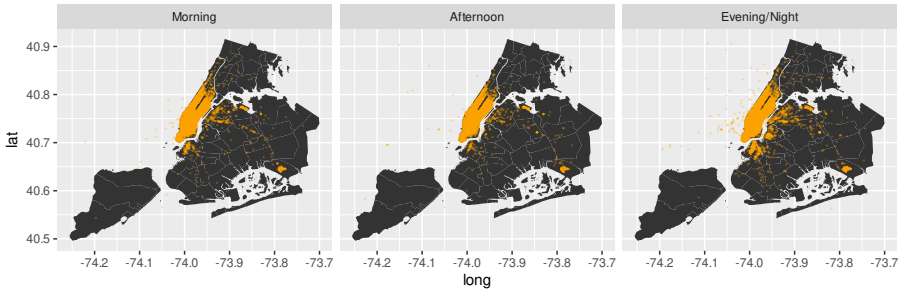
```
# add pick-up locations to plot
locations +
     geom_point(aes(x=dest_long, y=dest_lat),
                color="steelblue",
                size = 0.1,
                alpha = 0.2) +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
    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



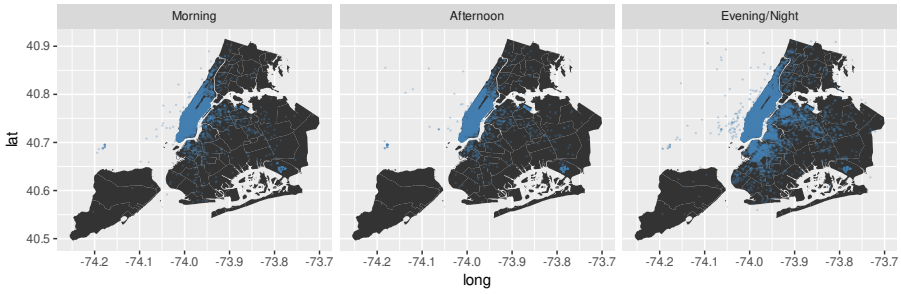
Incidentally, more trips tend to end outside of Manhattan. And the destinations seem to be broader spread across the city then the pick-up locations. Most destinations are still in Manhattan, though.

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))
```

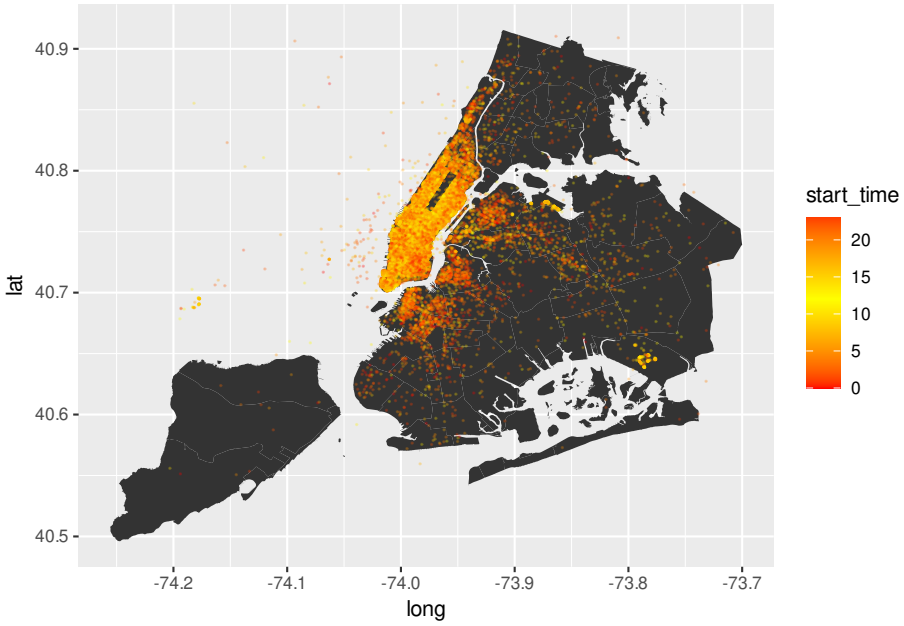



```
# drop-off locations
locations +
     geom_point(aes(x=dest_long, y=dest_lat),
                color="steelblue",
                size = 0.1,
                alpha = 0.2) +
     facet_wrap(vars(time_of_day))
```



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

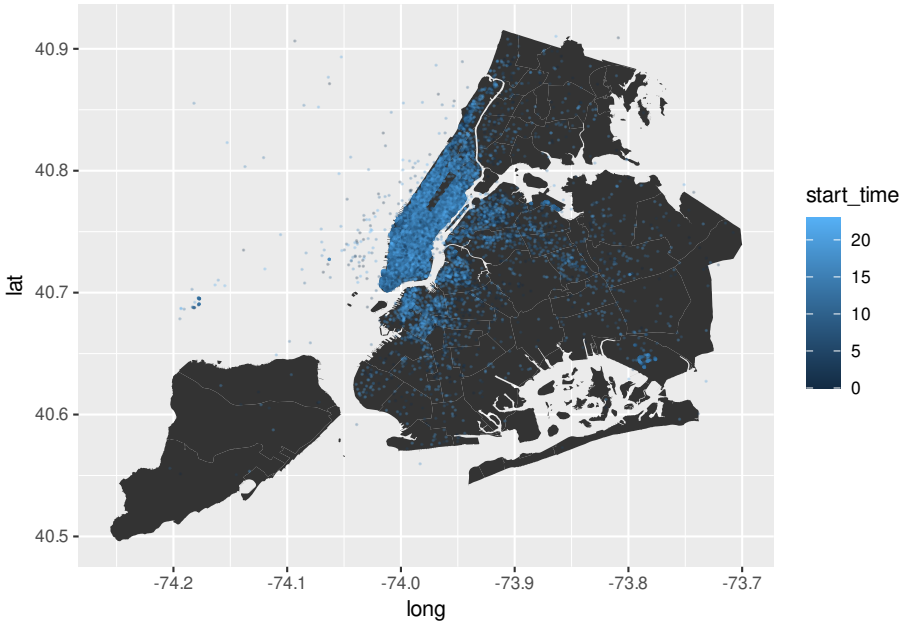
```
# drop-off locations
locations +
     geom_point(aes(x=dest_long, y=dest_lat, color = start_time ),
                size = 0.1,
                alpha = 0.2) +
     scale_colour_gradient2( low = "red", mid = "yellow", high = "red",
                             midpoint = 12)
```



11.4 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):

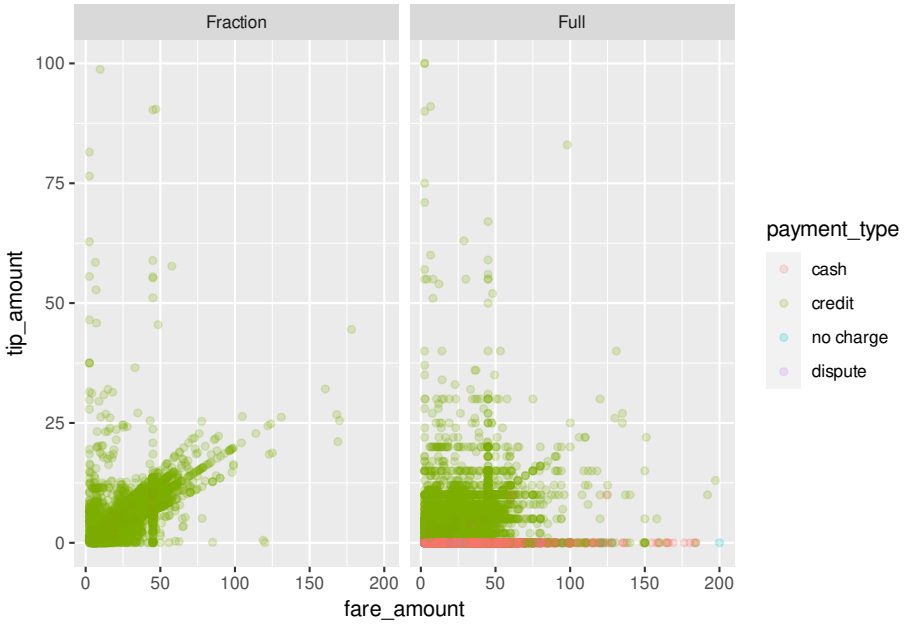
```
# drop-off locations
locations +
     geom_point(aes(x=dest_long, y=dest_lat, color = start_time ),
                size = 0.1,
                alpha = 0.2)
```



`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.

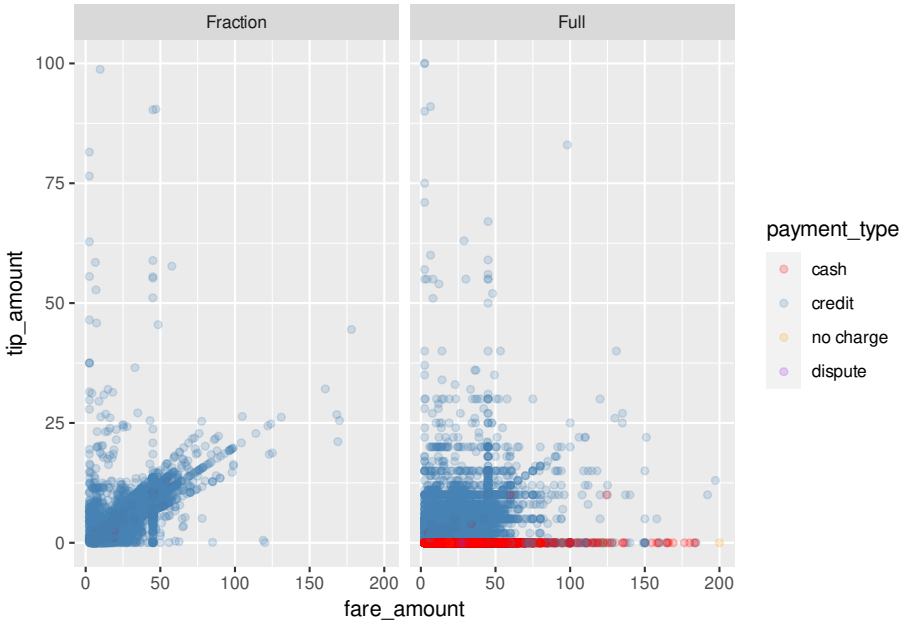
Recall the plot illustrating tipping behavior, where we highlight in which observations the client payd with credit card, cash, etc.

```
# 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
100
75
50
25
0
tip_amount
0
50
100
150
200
fare_amount
payment_type
cash
credit
no charge
dispute

12

Bottle Necks in Local Big Data Analytics

12.1 Case study: Data Import and 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 <- "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 certain 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 read-

ing 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)


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

```
## 2.31 GB
```

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

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

```
## 33 MB
```

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

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

```
## 2.34 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   8529715  455.6   14810401  791  12986236
## Vcells 232255241 1772.0  409858047 3127 409846454
##          (Mb)
## Ncells  693.6
## Vcells 3126.9
```

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.
## Input contains no \n. Taking this to be a filename to open
## [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 00:00.076 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
##   0.072s ( 95%) Reading 30 chunks (0 swept) of 0.984MB (each chunk 11225 rows) using 6 threads
##             +             0.015s    (   20%)   Parse   to   row-
major thread buffers (grown 0 times)
##    +    0.034s ( 44%) Transpose
##    +    0.023s ( 31%) Waiting
##        0.000s  (    0%)  Rereading  0  columns  due  to  out-of-
sample type exceptions
##    0.076s        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   8522088  455.2   14810401  791  12986236
## Vcells 229092840 1747.9  409858047 3127 409846454
##          (Mb)
## Ncells  693.6
## Vcells 3126.9
```

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

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

```
## 35.6 MB
```

12.2 Case Study: Loops, Memory, and Vectorization

12.2.1 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
```

12.2.2 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
##   0.095   0.000   0.095
```

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
## 2.069e-05
```

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

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

```
## elapsed
##    2069
```

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

```
## elapsed
##   34.48
```

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

```
## elapsed
##  0.5747
```

Can we improve this?

12.2.3 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 simple 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
## 1.742e-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
##   174.2
```

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

```
## elapsed
##   2.904
```

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

```
## elapsed
## 0.04839
```

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

12.2.4 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] 0.1603
```

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

```
## [1] 0.002672
```

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

```
## [1] 4.453e-05
```

12.3 Case study: Bootstrapping and Parallel Processing

In this example, we estimate a simple regression model that aims to assess racial discrimination in the context of police stops.[1] 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 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
```

[1] 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.

```
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[2] 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.

```
# 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
```

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

```
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
##    0.004507    0.005998
```

12.3.1 Parallelization with an EC2 instance

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 ---------------------------

# 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
```

12.4 Case Study: Efficient Fixed Effects Estimation

In this case study we look into a very common computational problem in applied econometrics: estimation of a fixed effects model with various fixed-effects units (i.e., many intercepts). We look at this in the context of the study on "Friends in High Places"[3] by Cohen and Malloy (2014). Cohen and Malloy show that US Senators who are alumni of the same university/college tend to help each other out in votes on industrial policies if the corresponding policy is highly relevant for the state of one senator but not relevant for the state of the other senator. The data is provided along the published article and can be accessed here: `http://doi.org/10.3886/E114873V1`. The data (and code) is provided in STATA format. We can import the main data set with the `foreign` package.

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



# load packages
library(foreign)
```

[3] https://www.aeaweb.org/articles?id=10.1257/pol.6.3.63

```
library(data.table)
library(lmtest)

# fix vars
DATA_PATH <- "data/data_for_tables.dta"

# import data
cm <- as.data.table(read.dta(DATA_PATH))
# keep only clean obs
cm <- cm[!(is.na(yes)|is.na(pctsumyessameparty)|is.na(pctsumyessameschool)|is.na(pctsumyessame
```

As part of this case study, we will replicate parts of Table 3 of the main article (p. 73). Specifically, we will estimate specifications (1) and (2). In both specifications, the dependent variable is an indicator `yes` that is equal to 1 if the corresponding senator voted Yes on the given bill and 0 otherwise. The main explanatory variables of interest are `pctsumyessameschool` (the percentage of senators from the same school as the corresponding senator who voted Yes on the given bill), `pctsumyessamestate` (the percentage of senators from the same state as the corresponding senator who voted Yes on the given bill), and `pctsumyessameparty` (the percentage of senators from the same party as the corresponding senator who voted Yes on the given bill) Specification 1 accounts for congress (time) fixed effects and senator (individual) fixed effects, and specification 2 account for congress-session-vote fixed effects and senator fixed effects.

First, let us look at a very simple example to highlight where the computational burden in the estimation of such specifications is coming from. In terms of the regression model 1, the fixed effect specification means that we introduce an indicator variable (an intercept) for $N-1$ senators and $M-1$ congresses. That is, while the simple model matrix (X) without accounting for fixed effects has dimensions 425653×4.

```
# pooled model (no FE)
model0 <-   yes ~
  pctsumyessameschool +
```

```
  pctsumyessamestate +
  pctsumyessameparty

dim(model.matrix(model0, data=cm))
```

```
## [1] 425653      4
```

In contrast, the model matrix of specification (1) is of dimensions 425653×221, and the model matrix of specification (2) even of 425653×6929.

```
model1 <-
  yes ~ pctsumyessameschool + pctsumyessamestate + pctsumyessameparty +
  factor(congress) + factor(id) -1
mm1 <- model.matrix(model1, data=cm)
dim(mm1)
```

```
## [1] 425653    168
```

Using OLS to estimate such a model thus involves the computation of a very large matrix inversion (because $\hat{\beta}_{OLS} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$). In addition, the model matrix for specification 2 is about 22GB, which might further slow down the computer due to a lack of physical memory or even crash the R session altogether.

In order to set a point of reference, we first estimate specification (1) with standard OLS.

```
# fit specification (1)
runtime <- system.time(fit1 <- lm(data = cm, formula = model1))
coeftest(fit1)[2:4,]
```

```
##                     Estimate Std. Error t value
## pctsumyessamestate   0.11861   0.001085 109.275
## pctsumyessameparty   0.92640   0.001397 662.910
## factor(congress)101 -0.01458   0.006429  -2.269
##                     Pr(>|t|)
```

```
## pctsumyessamestate    0.0000
## pctsumyessameparty    0.0000
## factor(congress)101   0.0233
```

```
# median amount of time needed for estimation
runtime[3]
```

```
## elapsed
##   11.81
```

As expected, this takes quite some time to compute. However, there is an alternative approach to estimating such models that substantially reduces the computational burden by "sweeping out the fixed effects dummies". In the simple case of only one fixed effect variable (e.g., only individual fixed effects), the trick is called "within transformation" or "demeaning" and is quite simple to implement. For each of the categories in the fixed effect variable the mean of the covariate and subtract the mean from the covariate's value.

```
# illustration of within transformation for the senator fixed effects
cm_within <-
  with(cm, data.table(yes = yes - ave(yes, id),
                      pctsumyessameschool = pctsumyessameschool - ave(pctsumyessameschool, id)
                      pctsumyessamestate = pctsumyessamestate - ave(pctsumyessamestate, id),
                      pctsumyessameparty = pctsumyessameparty - ave(pctsumyessameparty, id)
                      ))

# comparison of dummy fixed effects estimator and within estimator
dummy_time <- system.time(fit_dummy <-
              lm(yes ~ pctsumyessameschool +
                          pctsumyessamestate + pctsumyessameparty + factor(id) -1, data = cm
                        ))
within_time <- system.time(fit_within <-
                             lm(yes ~ pctsumyessameschool +
                             pctsumyessamestate + pctsumyessameparty -1, data = cm_within))
# computation time comparison
as.numeric(within_time[3])/as.numeric(dummy_time[3])
```

```
## [1] 0.003924
```

```
# comparison of estimates
coeftest(fit_dummy)[1:3,]
```

```
##                      Estimate Std. Error t value
## pctsumyessameschool  0.04424   0.001352   32.73
## pctsumyessamestate   0.11864   0.001085  109.30
## pctsumyessameparty   0.92615   0.001397  662.93
##                       Pr(>|t|)
## pctsumyessameschool 1.205e-234
## pctsumyessamestate   0.000e+00
## pctsumyessameparty   0.000e+00
```

```
coeftest(fit_within)
```

```
##
## t test of coefficients:
##
##                      Estimate Std. Error t value
## pctsumyessameschool  0.04424    0.00135    32.7
## pctsumyessamestate   0.11864    0.00109   109.3
## pctsumyessameparty   0.92615    0.00140   663.0
##                     Pr(>|t|)
## pctsumyessameschool   <2e-16 ***
## pctsumyessamestate    <2e-16 ***
## pctsumyessameparty    <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Unfortunately, we cannot simply apply the same procedure in a specification with several fixed effects variables. However, Gaure (2013b) provides a generalization of the linear within-estimator to several fixed effects variables. This method is implemented in the `lfe` package (Gaure (2013a)). With this package, we can easily estimate both fixed-effect specifications (as well as the corresponding cluster-robust standard

errors) in order to replicate the original results by Cohen and Malloy (2014).

```
library(lfe)

# model and clustered SE specifications
model1 <- yes ~ pctsumyessameschool + pctsumyessamestate + pctsumyessameparty |congress+id|0|
model2 <- yes ~ pctsumyessameschool + pctsumyessamestate + pctsumyessameparty |congress_sessio

# estimation
fit1 <- felm(model1, data=cm)
fit2 <- felm(model2, data=cm)
```

Finally we can display the regression table.

```
stargazer::stargazer(fit1,fit2,
                     type="text",
                     dep.var.labels = "Vote (yes/no)",
                     covariate.labels = c("School Connected Votes",
                                          "State Votes",
                                          "Party Votes"),
                     keep.stat = c("adj.rsq", "n"))
```

```
##
## ================================================
##                            Dependent variable:
##                        ----------------------------
##                               Vote (yes/no)
##                             (1)            (2)
## ------------------------------------------------
## School Connected Votes    0.045***      0.052***
##                           (0.016)        (0.016)
##
## State Votes               0.119***      0.122***
##                           (0.013)        (0.012)
##
## Party Votes               0.926***      0.945***
```

```
##                            (0.022)        (0.024)
##
## ------------------------------------------------
## Observations               425,653        425,653
## Adjusted R2                 0.641          0.641
## ================================================
## Note:                   *p<0.1; **p<0.05; ***p<0.01
```

13

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[1], and the high-level Keras[2] 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.

13.1 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[3]. The example follows closely this keras tutorial[4] published by RStudio. See RStudio's keras installation guide[5] for how to install keras (and tensorflow) and

[1] https://www.tensorflow.org/

[2] https://keras.io/

[3] https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html

[4] https://keras.rstudio.com/articles/tutorial_basic_regression.html#the-boston-housing-prices-dataset

[5] https://keras.rstudio.com/index.html

the corresponding R package `keras`.[6] 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 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()
```

```
## Loaded Tensorflow version 2.5.0
```

```
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 ...
```

[6] This might involve the installation of additional packages and software outside the R environment.

13.2 Data preparation

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.[7] As

[7] Typical textbook examples in machine learning deal with classification (e.g. a

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
##  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
```

logit model), while in microeconometrics the typical example is usually a linear model (continuous dependent variable).

```
##  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)
```

13.3 Model specification

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[8] 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.

[8] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

```
# get a summary of the model
model
```

13.4 Training and prediction

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)
```

13.4.1 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).[9].

[9] For a more detailed discussion of the relevant factors for well done parallelization (either on CPUs or GPUs), see Matloff (2015)

Part IV

Application: Topics in Big Data Econometrics

14

Regression Analysis and Categorization with Spark and R

14.1 Simple regression analysis

Suppose we want to conduct a correlation study of what factors are associated with longer or shorter arrival delays in air travel. Via its built-in 'MLib' library, Spark provides 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.[1]

```
# flights_r <- collect(flights) # very slow!
flights_r <- data.table::fread("data/flights.csv", nrows = 300)
```

[1]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.

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 `sparklyr`.[2] In order to use Spark via the `sparklyr` package, we need to first

[2] Most regression models commonly used in traditional applied econometrics are in some form provided in `sparklyr` or `SparkR`. See the package documentation for more details.

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")
```

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
flights_spark <- copy_to(sc, flights_r, "flights_spark")
# fit the model
fit1_spark <- ml_linear_regression(flights_spark, 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
```

Alternatively, we can use the `spark_apply()` function to run the regression analysis in R via the original R `lm()`-function.[3]

[3] Note, though, that this approach might take longer.

```
# fit the model
spark_apply(flights_spark, function(df) broom::tidy(lm(arr_delay ~ dep_delay + distance, df)),
        names = c("term", "estimate", "std.error", "statistic", "p.value")
    )
```

Finally, the `parsnip` package (together with the `tidymodels` package) provides a simple interface to run the same model (or similar specifications) on different "engines" (estimators/fitting algorithms), and several of the `parsnip` models are also supported in `sparklyr`. This significantly facilitates the transition from local testing (with a small subset of the data) to running the estimation on the entire data set on spark.

```
library(tidymodels)
library(parsnip)

# simple local linear regression example from above
# via tidymodels/parsnip
fit1 <- fit(linear_reg(engine="lm"), model1, data=flights_r)
tidy(fit1)
```

```
## # A tibble: 3 x 5
##   term         estimate std.error statistic   p.value
##   <chr>           <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -0.183      1.68      -0.109  9.13e-  1
## 2 dep_delay    0.990      0.0173    57.3    1.63e-162
## 3 distance     0.000114   0.00124    0.0920 9.27e-  1
```

```
# run the same on Spark
fit1_spark <- fit(linear_reg(engine="spark"), model1, data=flights_spark)
tidy(fit1_spark)
```

```
## # A tibble: 3 x 5
##   term         estimate std.error statistic p.value
##   <chr>           <dbl>     <dbl>     <dbl>   <dbl>
## 1 (Intercept) -0.183      1.68      -0.109    0.913
## 2 dep_delay    0.990      0.0173    57.3      0
```

```
## 3 distance     0.000114   0.00124    0.0920   0.927
```

We will further build on this interface in the next section where we look at different machine learning procedures for a classification problem.

14.2 Machine learning for classification

Building on `sparklyr`, `tidymodels`, and `parsnip`, we test a set of machine learning models on the classification problem discussed in Varian (2014): predicting Titanic survivors. The data for this exercise can be downloaded from here: http://doi.org/10.3886/E113925V1.

We import and prepare the data in R.

```
# load into R, # select variables of interest, remove missing
titanic_r <- read.csv("data/titanic3.csv")
titanic_r <- na.omit(titanic_r[, c("survived",
                           "pclass",
                           "sex",
                           "age",
                           "sibsp",
                           "parch")])
titanic_r$survived <- ifelse(titanic_r$survived==1, "yes", "no")
```

In order to assess the performance of the classifiers later on, we split the sample into training and test data sets. We do so with the help of the `rsample` package, which provides a number of high-level functions to facilitate this kind of pre-processing.

```
library(rsample)

# split into training and test set
titanic_r <- initial_split(titanic_r)
ti_training <- training(titanic_r)
ti_testing <- testing(titanic_r)
```

For the training and assessment of the classifiers, we transfer the two data sets to the spark cluster.

```
# load data to spark
ti_training_spark <- copy_to(sc, ti_training, "ti_training_spark")
ti_testing_spark <- copy_to(sc, ti_testing, "ti_testing_spark")
```

Now we can set up a 'horse race' between different ML approaches to find the best performing model. Overall, we will consider the following models/algorithms:

- Logistic regression
- Boosted trees
- Random forest

```
# models to be used
models <- list(logit=logistic_reg(engine="spark", mode = "classification"),
               btree=boost_tree(engine = "spark", mode = "classification"),
               rforest=rand_forest(engine = "spark", mode = "classification"))
# train/fit the models
fits <- lapply(models, fit, formula=survived~., data=ti_training_spark)
```

The fitted models (trained algorithms) can now be assessed with the help of the test data set. To this end, we use the high-level `accuracy` function provided in the `yardstick` package in order to compute the accuracy[4] of the fitted models. We proceed in three steps. First, we use the fitted models to predict the outcomes (we classify cases into survived/not survived) of the *test set*. Then we fetch the predictions from the Spark cluster, format the variables, and add the actual outcomes as an additional column.

```
# run predictions
predictions <- lapply(fits, predict, new_data=ti_testing_spark)
# fetch predictions from Spark, format, add actual outcomes
pred_outcomes <-
```

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

```
     lapply(1:length(predictions), function(i){
          x_r <- collect(predictions[[i]]) # fetch from spark cluster (load into local R envir
          x_r$pred_class <- as.factor(x_r$pred_class) # format for predictions
          x_r$survived <- as.factor(ti_testing$survived) # add true outcomes
          return(x_r)

})
```

Finally, we compute the accuracy of the models, stack the results, and display them (ordered from best-performing to worst-performing.)

```
acc <- lapply(pred_outcomes, accuracy, truth="survived", estimate="pred_class")
acc <- bind_rows(acc)
acc$model <- names(fits)
acc[order(acc$.estimate, decreasing = TRUE),]
```

```
## # A tibble: 3 x 4
##   .metric  .estimator .estimate model
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.802 logit
## 2 accuracy binary         0.802 rforest
## 3 accuracy binary         0.782 btree
```

In this simple example, all models perform similarly well. However, none of them really performs outstandingly. In a next step, we might want to learn about which variables are considered more or less important for the predictions. Here, the `tidy()`-function is very useful. As long as the model types are comparable (here `btree` and `rforest`), `tidy()` delivers essentially the same type of summary for different models.

```
tidy(fits[["btree"]])
```

```
## # A tibble: 5 x 2
##   feature  importance
##   <chr>         <dbl>
## 1 age          0.379
```

```
## 2 sex_male     0.234
## 3 pclass       0.199
## 4 sibsp        0.0951
## 5 parch        0.0931
```

```
tidy(fits[["rforest"]])
```

```
## # A tibble: 5 x 2
##   feature  importance
##   <chr>         <dbl>
## 1 sex_male     0.580
## 2 pclass       0.204
## 3 age          0.138
## 4 sibsp        0.0517
## 5 parch        0.0268
```

Finally, we clean up and disconnect from the Spark cluster.

```
spark_disconnect(sc)
```

14.3 Building machine learning pipelines with R and Spark

Spark provides a framework to implement machine learning pipelines called ML Pipelines[5] with the aim of facilitating the combination of various preparatory steps and ML algorithms into a pipeline/workflow. `sparklyr` provides a straightforward interface to ML Pipelines that allows implementing and testing the entire ML workflow in R and then easily deploying the final pipeline to a Spark cluster or more generally to the production environment. In the following example, we will revisit the e-commerce purchase prediction model (Google Analytics data from the Google Merchandise Shop) introduced in Chapter 1. That is, we want to prepare the Google Analytics data and then use a

[5] https://spark.apache.org/docs/latest/ml-pipeline.html

LASSO to find a set of important predictors for purchase decisions, all built into a ML pipeline.

14.3.1 Set up and data import

All of the key ingredients are provided in `sparklyr`. However, I recommend using the 'piping' syntax provided in `dplyr` to implement the ML pipeline. In this context, using this syntax is particularly helpful to make the code easy to read and understand.

```
# load packages
library(sparklyr)
library(dplyr)

# fix vars
INPUT_DATA <- "data/ga.csv"
```

Recall that the Google Analytics data set is small subset of the overall data generated by Google Analytics on a moderately sized e-commerce site. Hence, it makes perfectly sense to first implement and test the pipeline locally (on a local Spark installation), before deploying it on an actual Spark cluster in the cloud. In a first step, we thus copy the imported data to the local Spark instance.

```
# import to local R session, prepare raw data
ga <- na.omit(read.csv(INPUT_DATA))
#ga$purchase <- as.factor(ifelse(ga$purchase==1, "yes", "no"))
# connect to, and copy the data to the local cluster
sc <- spark_connect(master = "local")
ga_spark <- copy_to(sc, ga, "ga_spark")
```

14.3.2 Building the pipeline

The pipeline object is initiated via `ml_pipeline()`, in which we refer to the connection to the local Spark cluster. We then add the model specification (the formula) with `ft_r_formula()` to the pipeline. `ft_r_formula` essentially transforms the data in accordance with the common speci-

fication syntax in R (here: `purchase ~ .`). Among other things, this takes care of properly setting up the model matrix. Finally, we add the model via `ml_logistic_regression()`. We can set the penalization parameters via `elastic_net_param` (with `alpha=1`, we get the lasso).

```
# ml pipeline
ga_pipeline <-
     ml_pipeline(sc) %>%
     ft_string_indexer(input_col="city",
                       output_col="city_output",
                       handle_invalid = "skip") %>%
     ft_string_indexer(input_col="country",
                       output_col="country_output",
                       handle_invalid = "skip") %>%
     ft_string_indexer(input_col="source",
                       output_col="source_output",
                       handle_invalid = "skip") %>%
     ft_string_indexer(input_col="browser",
                       output_col="browser_output",
                       handle_invalid = "skip") %>%
     ft_r_formula(purchase ~ .) %>%
     ml_logistic_regression(elastic_net_param = list(alpha=1))
```

Finally, we create a cross-validator object to train the model with a k-fold cross-validation and fit the model.

```
# specify the hyperparameter grid
# (parameter values to be considered in optimization)
ga_params <- list(logistic_regression=list(max_iter=80))

# create the cross-validator object
set.seed(1)
cv_lasso <- ml_cross_validator(sc,
                         estimator=ga_pipeline,
                         estimator_param_maps = ga_params,
                         ml_multiclass_classification_evaluator(sc),
```

```
                          num_folds = 50,
                          parallelism = 2)

# train/fit the model
cv_lasso_fit <- ml_fit(cv_lasso, ga_spark)
```

15

Large-scale Text Analysis with sparklyr

Text analysis/natural language processing often involves rather large amounts of data and is particularly challenging for in-memory-processing. `sparklyr` provides several easy-to-use functions to run some of the computationally most demanding text data handling on a Spark cluster. In this chapter we explore these functions and the corresponding workflows to do text analysis on an AWS EMR cluster running Spark.

15.1 Getting started: import, preparation, and word frequencies

The following example briefly guides through some of the most common first steps when processing text data for NLP. In the code example, we process Friedrich Schiller's "Wilhelm Tell" (English edition; Project Gutenberg Book ID 2782), which we download from Project Gutenberg[1]. The example can easily be extended to process many more books.

The example is set up to work straightforwardly on an AWS EMR cluster. However, given the relatively small amount of data processed here, you can also run it locally. In case you want to run it on EMR, simply follow the steps Chapter 6.4 to set up the cluster and log in to RStudio on the master node. The `sparklyr` package is already installed on EMR (if you use the bootstrap-script introduced in Chapter 6.4 for the set up of the cluster), but other packages might still have to be installed.

We first load the packages and connect the RStudio session to the cluster (in case you run this locally, use `spark_connect(master="local")`).

[1] https://www.gutenberg.org/

```
# install additional packages
# install.packages("gutenbergr") # to download book texts from Project Gutenberg
# install.packages("dplyr") # for the data preparatory steps

# load packages
library(sparklyr)
library(gutenbergr)
library(dplyr)

# fix vars
TELL <- "https://www.gutenberg.org/cache/epub/6788/pg6788.txt"


# connect rstudio session to cluster
sc <- spark_connect(master = "yarn")
```

We fetch the raw text of the book and copy it to the Spark cluster. Note that you can do this sequentially for many books without exhausting the master node's RAM and then further process the data on the cluster.

```
# Data gathering and preparation
# fetch Schiller's Tell, load to cluster
tmp_file <- tempfile()
download.file(TELL, tmp_file)
raw_text <- readLines(tmp_file)
tell <- data.frame(raw_text=raw_text)
tell_spark <- copy_to(sc, tell, "tell_spark", overwrite = TRUE)
```

The text data will be processed in a Spark DataFrame columnd behind the `tbl_spakr`-object. First, we remove empty lines of text, select the column containing all the text, and then remove all non-numeric and non-alphabetical characters. The latter step is an important text cleaning step as we want to avoid special characters to be considered words or part of words later on.

```
# data cleaning
tell_spark <- filter(tell_spark, raw_text!="")
tell_spark <- select(tell_spark, raw_text)
tell_spark <- mutate(tell_spark,
                     raw_text = regexp_replace(raw_text, "[^0-9a-zA-Z]+", " "))
```

Now we can split the lines of text in column `raw_text` into individual words (sequences of characters that have been separated by white space). To this end we can call a Spark feature transformation routine called the tokenization, which essentially breaks text into individual terms. Specifically, each line of raw text in column `raw_text` will be split into words. The overall result (stored in a new column specified with `output_col`), is then a nested list in which each word is an element of the corresponding line element.

```
# split into words
tell_spark <- ft_tokenizer(tell_spark,
                           input_col = "raw_text",
                           output_col = "words")
```

Now we can call another feature transformer called "stop words remover", which excludes all the stop words (words often occurring in a text but not carrying much information) from the nested word list.

```
# remove stop-words
tell_spark <- ft_stop_words_remover(tell_spark,
                                    input_col = "words",
                                    output_col = "words_wo_stop")
```

Finally, we combine all of the words in one vector and store the result in a new Spark DataFrame called "all_tell_words" (by calling `compute()`) and add some final cleaning steps.

```
# unnest words, combine in one row
all_tell_words <- mutate(tell_spark,
```

```
                word = explode(words_wo_stop))

# final cleaning
all_tell_words <- select(all_tell_words, word)
all_tell_words <- filter(all_tell_words, 2<nchar(word))
```

Based on this cleaned set of words, we can compute the word count for the entire book.

```
# word count and store result in Spark memory
compute(count(all_tell_words, word), "wordcount_tell")
```

```
## # Source: spark<wordcount_tell> [?? x 2]
##    word           n
##    <chr>      <dbl>
##  1 language       2
##  2 martin         1
##  3 baron          8
##  4 nephew         3
##  5 hofe           3
##  6 reding        16
##  7 fisherman     39
##  8 baumgarten    32
##  9 hildegard      3
## 10 soldiers       4
## # ... with more rows
## # i Use `print(n = ...)` to see more rows
```

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`[2].
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`[3], 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>`:

[2] www.github.com/umatter/BigData
[3] 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

.2 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.

.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
```

.3 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.

.3.1 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.

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`.

```
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
```

.3.2 Matrices/Arrays

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

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
```

.3.3 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.

The historical implementation of data frames in R is not very comfortable to work with large data sets.[4] 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:

[4] 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).

```
# 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
```

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

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 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.

Appendix C

.5 Install Hadoop (on Ubuntu Linux)

```
# download binary
wget https://dlcdn.apache.org/hadoop/common/hadoop-2.10.1/hadoop-2.10.1.tar.gz
# download checksum
wget https://dlcdn.apache.org/hadoop/common/hadoop-2.10.1/hadoop-2.10.1.tar.gz.sha512

# run the verification
shasum -a 512 hadoop-2.10.1.tar.gz
# compare with value in mds file
cat hadoop-2.10.1.tar.gz.sha512

# if all is fine, unpack
tar -xzvf hadoop-2.10.1.tar.gz
# move to proper place
sudo mv hadoop-2.10.1 /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-2.10.1.tar.gz
rm hadoop-2.10.1.tar.gz.sha512
```

After running all of the steps above, run the following line in the terminal to check the installation

```
# check installation
/usr/local/hadoop/bin/hadoop
```

Bibliography

Burns, P. (2011). *The R Inferno*. Lulu Press, Inc.

Cohen, L. and Malloy, C. J. (2014). Friends in high places. *American Economic Journal: Economic Policy*, 6(3):63–91.

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.

Donoho, D. (2017). 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766.

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.

Gaure, S. (2013a). lfe: Linear Group Fixed Effects. *The R Journal*, 5(2):104–116.

Gaure, S. (2013b). Ols with multiple high dimensional category variables. *Computational Statistics & Data Analysis*, 66:8–18.

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.

Varian, H. R. (2014). Big data: New tricks for econometrics. *Journal of Economic Perspectives*, 28(2):3–28.

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.