Workflow of statistical data analysis

Oliver Kirchkamp
Workflow of empirical work may seem obvious. It is not. Small initial mistakes can lead to a lot of hard work afterwards. In this course we discuss some techniques that hopefully facilitate the organisation of your empirical work.

This handout provides a summary of the slides from the lecture. It is not supposed to replace a book.

Many examples in the text are based on the statistical software R. I urge you to try these examples on your own computer.

As an attachment of this PDF you find a file wf.zip with some raw data. You also find a file wf.Rdata with some R functions and some data already in R’s internal format.

The drawing on the previous page is Albercht Dürer’s “Der Hafen von Antwerpen” — an example for workflow in a medieval city.
### Organising work

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Scripts</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Robust scripts</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Robustness towards different computers</td>
<td>29</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Robustness towards changes in context</td>
<td>30</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Functions increase robustness</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Calculations that take a lot of time</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Nested functions</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>Reproducible randomness</td>
<td>33</td>
</tr>
<tr>
<td>3.5</td>
<td>Recap — writing scripts and using functions</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Human readable scripts</td>
<td>34</td>
</tr>
</tbody>
</table>

### Some programming techniques

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Debugging functions</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Lists of variables</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Return values of functions</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Repeating things</td>
<td>42</td>
</tr>
</tbody>
</table>

### Data manipulation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Subsetting data</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Merging data</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Reshaping data</td>
<td>55</td>
</tr>
</tbody>
</table>

### Preparing Data

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Reading data</td>
<td>56</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Reading z-Tree Output</td>
<td>56</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Reading and writing R-Files</td>
<td>58</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Reading Stata Files</td>
<td>58</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Reading CSV Files</td>
<td>67</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Filesize</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Checking Values</td>
<td>68</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Range of values</td>
<td>68</td>
</tr>
<tr>
<td>6.2.2</td>
<td>(Joint) distribution of values</td>
<td>69</td>
</tr>
<tr>
<td>6.2.3</td>
<td>(Joint) distribution of missings</td>
<td>71</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Checking signatures</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Naming variables</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Labeling (describing) variables</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>Labeling values</td>
<td>74</td>
</tr>
<tr>
<td>6.6</td>
<td>Recoding data</td>
<td>76</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Replacing values by missings</td>
<td>76</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Replacing values by other values</td>
<td>77</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Comparison of missings</td>
<td>77</td>
</tr>
<tr>
<td>6.7</td>
<td>Changing variables – creating new variables</td>
<td>77</td>
</tr>
<tr>
<td>6.8</td>
<td>Select subsets</td>
<td>78</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

Literature: Surprisingly, there is not much literature about workflow of statistical data analysis:

General Literature


Literate Programming

- Friedrich Leisch; Sweave User Manual.
• Nicola Sartori; Sweave = R \cdot \LaTeX
• Yihui Xie; knitr - Elegant, flexible, and fast dynamic report generation with R
• Max Kuhn; CRAN Task View: Reproducible Research.

Version control

• Scott Chacon, Ben Straub; Pro Git.
• Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato; Version Control with Subversion.

What is workflow:

• A sequence of operations.

• A pattern of actions that can be documented and learned.

• We spend a lot of time explaining statistical methods to students.

• We do not tell students how to apply these methods (how to integrate methods into a "workflow")

• Why?

• Is “workflow” obvious? — I do not think so.

  Is the wrong workflow not costly? — On the contrary.
  - Mistakes in the statistical method can always be cured.
  - Mistakes in the workflow can render the entire project invalid — no cure possible (e.g. loss of data, loss of understanding the data, loss of methods applied)

• Isn’t it sufficient to simply store and backup everything?

  - unfortunately not — statistical analysis tends to create a lot of data.→ storing everything means hiding everything very well from ourselves and from others.
1.2 Structure of a paper

- Describe the research question
  Which *economic model* do we use to structure this question?
  Which *statistical model* do we use for inference? (Estimation, hypothesis testing, classification...)

- Describe the economic method (experiment, field data,...)

- Describe the sample
  How many observations, means, distributions of main variables, key statistics
  Is there enough variance in the independent variables to test what we want to test?

- Statistical inference (estimate, test hypotheses, classify,...)
  possibly different variants of the model (increasing complexity)

- Discuss the model, robustness checks

1.3 Aims of statistical data analysis

- Limit work and time

- Get interesting results

- Replicability
  - for us, to understand our data and our methods after we get back to work after a short break
  - for our friends (coauthors), so that they can understand what we are doing
  - for our enemies — we should always (even years after) be able to *prove* our results *exactly*

- If statistical analysis was a straightforward procedure, then there would be no problem:
  - Store the raw data. All methods we apply are obvious and trivial.

- In the real world our methods are far from obvious:
  - We think quite a lot about details of our statistical analysis

- Assume we have another look at our paper (and our analysis) after a break of 6 month:
  - What does it mean if *sex==1* ?
  - For the variable *meanContribution*: was the mean taken with respect to all players and the same period, or with respect to the same player and all periods, or ...
  - What is the difference between *payoff* and *payoff2*...
Do the tables and figures in version 27 of the paper...
* ...refer to all periods of the experiment or only to the last 6 periods?
* ...do they include data from the two pilot experiments we ran?
* ...do they refer to the “cleaned” dataset, or to the “cleaned dataset in long form” (where we eliminated a few outliers)
* Do all tables and figures and p-values and t-tests... actually refer to the same data? (or do some include outliers, some not,...)

Assume we take only 10 not completely obvious decisions between two alternatives during our analysis (which perhaps took us 1 week),...

(coding of data, data to include, treatments to compare, lags to include, outliers to remove, interaction terms to include, types of model comparison, dealing with non-linearities, correlation structure of error terms,...)

...→ we will have to explore \(2^{10} = 1024\) variants of our analysis (= 1024 weeks) to recover what we actually did.

Often we take more than 10 not completely obvious decisions. → we should follow a workflow that facilitates replicability.

This is not obvious, since workflow is (unfortunately) not linear:

During this process we create a lot of intermediate results. How can we organise these results?

**Solutions and restrictions:**

- Store everything — not feasible
- We want to be creative, take shortcuts, we want to explore things, play with different representations of a solution...
- During this phase we can not document everything.
1.4 Creativity and chaos

Living two lives:

- creative (undocumented)
- permanent (documented)

Let our computer(s) reflect these two lives:

```bash
.../projectXYZ/
  /permanent/
    /rawData
    /cleanData
    /R
    /Paper
    /Slides
  /creative/
    /cleanData
    /R
    /Paper
    /Slides
```

You might need more directories for your work.
(In terms of version control, which we will cover later, “permanent” could be a trunk, while “creative” could be a branch)

**Rules**

1. Anything that we give to other people (collaborators, journals,...) must come entirely from permanent

2. Never delete anything from permanent

3. Never change anything in permanent

4. We must be able trace back everything in permanent clearly to our raw data.

Since we give things to other people more than once (first draft, second draft,..., first revision, ..., second revision,...), we must be able to replicate each of these instances.

**Consequences — permanent data has versions** (Below we will discuss the advantages of a version control system (git, svn). Let us assume for a moment that we have to do everything manually.)

- We will accumulate versions in our permanent life (do not delete them, do not change them)
cleaned_data_150721.Rdata
cleaned_data_150722.Rdata
cleaned_data_150722b.Rdata
:
preparingData_150721.R
preparingData_150722.R
descriptives_150722.R
econometrics_150723.R
:
paper_150724.Rnw
paper_150725.Rnw
paper_150727.Rnw
:

**What is the optimal workflow?** The optimal workflow is different for each of us

**Aims**

- Exactness (allow clear replication)
- Efficiency
- We must like it (otherwise we don’t do it)
- Whatever we do, we should do it in a systematic way
  - Follow a routine in our work (all projects should follow similar conventions)
  - Let the computer follow a routine (a mistake made in a routine will show up “routinely”, a hand coded mistake is harder to detect).
    - Use functions, try to make them as general as possible.
  - Prepare for the unexpected! We should not assume that our data will always look the way it looks at the moment.

**More on routines** Example:

- Probability to make a mistake: 0.1
- Probability to discover (and fix) a mistake: 0.8

Now you solve two related problems, A and B:

- Both problems are solved independently:
  - Probability of (undiscovered) mistake A: 0.1 · 0.2
  - Probability of (undiscovered) mistake B: 0.1 · 0.2
  - Probability of some undiscovered mistake: $1 - .98^2 \approx 0.04$
• Both problems are solved with the same routine (one function in your code):
  – Probability of some undiscovered mistake: \( 0.1 \cdot 0.2^2 = 0.004 \)

Producing your results with the help of identical (and computerised) routines makes it much easier to discover mistakes.

1.5 Making the analysis reproducible

Here are again the steps in writing a paper:

1. organise raw data
2. descriptive analysis (figures, descriptive tables...)
3. develop methods for analysis
4. get results (run program code)
5. write paper (mix results with text and explanations)
6. interact with collaborators

• All these tasks require decisions.
• All these decisions should be documented.
• When is our documentation sufficient? — If a third person, without our help, can find out what we were doing in all the above steps. If we want to have another look at our data in one year’s time we will be in the same position as an outsider today.
• We keep a log where we document the above steps for a given project on a daily basis (research log) (nobody wants to keep logs, so this must be easy)

1.6 Preserve raw data

• If our raw data comes from z-Tree experiments: We better keep all programs (the current version can always be found as @1.ztt,...in the working directory).
• If our raw data includes data from a questionnaire:
  – We need a codebook
    * variable name — question number — text of the questions
    * branching in the questionnaire
    * levels (value labels) used for factors
    * missing data, how was it coded?
    * cleaned data, how was it cleaned? (if we have no access to the raw data)
1.7 Interaction with coauthors

- Clear division of labour
  - the “experimenter” decides how the experiment is actually run
  - the “empiricist” decides what statistics and graphs are produced
  - the “writer” decides how to present the text
  - help, do not interfere
- In your communication: concentrate on the essentials:
  - exchange one file
  - make only essential changes to this file
  - clearly explain why these changes are necessary

2 Digression: R

For the purpose of the course we take R as an example for one statistical language. Even if you use other languages for your work, you will find that the concepts are similar.

2.1 Installation of R

On the [Homepage of the R Projekt](https://cran.r-project.org) you find in the menu on the left a link [Download / CRAN](https://cran.r-project.org). This link leads to a choice of “mirrors”. If you are in Jena, the [GWDG Mirror in Göttingen](https://cran.r-project.org/mirrors.html) might be fast. There you also find instructions how to install R on your OS.

**Installation of Libraries** If the command `library` complains about not being able to find the required library, then the library is most likely not installed. The command

```r
install.packages("Ecdat")
```

installs the library `Ecdat`. Some installations have a menu “Packages” that allows you to install missing libraries. Users of operating systems of Microsoft find support at the [FAQ for Packages](https://cran.r-project.org/doc/FAQ/R-FAQ.html).

2.2 Types and assignments

R knows about different types of data. We will meet some types in this chapter. To assign a number (or a value, or any object) to a variable, we use the operator `<-`

```r
x <- 4
```

R stores the result of this assignment as `double`
Now we can use \( x \) in our calculations:

\[
2 \times x \\
[1] 8
\]

\[
\text{sqrt}(x) \\
[1] 2
\]

Often our calculations will not only involve a single number (a scalar) but several which are connected as a vector. Several numbers are connected with \( \text{c} \):

\[
x <- \text{c}(21,22,23,24,25,16,17,18,19,20) \\
x \\
\]

When we need a long list of subsequent numbers, we use the operator \( : \):

\[
21:30 \\
\]

\[
y <- 21:30
\]

**Subsets** We can access single elements of a vector with \([\]\)

\[
x[1] \\
[1] 21
\]

When we want to access several elements at the same time, we simply use several indices (which are connected with \( \text{c} \)). We can use this to change the sequence of values (e.g. to sort).

\[
x[c(3,2,1)] \\
[1] 23 22 21
\]

\[
x[3:1] \\
[1] 23 22 21
\]
(to sort a long vector we would use the function `order`).

```r
order(x)
```

```r
[1]  6  7  8  9 10  1  2  3  4  5
```

```r
x[order(x)]
```

```r
[1] 16 17 18 19 20 21 22 23 24 25
```

(order determines an “ordering”, i.e. a sequence in which the elements of the vector should be used to be “ordered”. We use `x[... ]` to see the ordered result. )

Negative indices drop elements:

```r
x[-1:-3]
```

```r
[1] 24 25 16 17 18 19 20
```

### Logicals

Logicals can be either `TRUE` or `FALSE`. When we compare a vector with a number, then all the elements will be compared (this follows from the `recycling rule`, see below):

```r
x < 20
```

```r
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE
```

We can use logicals as indices, too:

```r
x [ x < 20 ]
```

```r
[1] 16 17 18 19
```

### Characters

Not only numbers, also character strings can be assigned to a variable:

```r
x <- "Mary"
```

We can also work with vectors of character strings:

```r
x <- c("John","Mary","Jane")
x[2]
```

```r
[1] "Mary"
```

```r
x[3]<-"Lucy"
x
```

```r
[1] "John" "Mary" "Lucy"
```
Factors  Often it is clumsy to store a string of characters again and again if this string appears in the dataset several times. We might, e.g., want to store whether an observation belongs to a man or a woman. This can be done in an efficient way by storing 2 for "male", and 1 for "female".

```r
x <- factor(c("male", "female", "female", "male"))
levels(x)
[1] "female" "male"

x[2]
[1] female
Levels: female male

as.numeric(x)
[1] 2 1 1 2
```

Usually the first level in a factor is the level that comes first on the alphabet. If we do not want this, we can `relevel` a factor:

```r
x <- relevel(x, "male")
x
[1] male female female male
Levels: male female

as.numeric(x)
[1] 1 2 2 1
```

Note that the meaning of the values remains unchanged. Sometimes, when we have more than only two levels, we want to order levels of a factor along a third variable. This is done by `reorder`.

```r
y <- c(12, 7, 8, 11)
reorder(x, y)
[1] male female female male
Levels: female male
```

Note that the meaning of the values remains unchanged.

2.3 Functions

R knows many built-in functions:
When we need more, we can write our own:

```r
square <- function(x) {
  x*x
}
```

The last expression in a function (here `x*x`) is the return value. Now we can use the function.

```r
square(7)
```

```
[1] 49
```

When we want to apply a function to many numbers, `sapply` helps:

```r
range <- 1:10
sapply(range, square)
```

```
[1]  1  4  9 16 25 36 49 64 81 100
```

With `sapply` we do not have to define a name for a function:

```r
sapply(range, function(x) x*x)
```

```
[1]  1  4  9 16 25 36 49 64 81 100
```

## 2.4 Random numbers

Random numbers can be generated for rather different distributions. R calculates pseudo-random numbers, i.e. R picks numbers from a very long list that appears random. Where we start in this long list is determined by `set.seed`:

```r
set.seed(123)
```

10 pseudo-random numbers from a normal distribution can be obtained with

```r
rnorm(10)
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
[7]  0.46091621 -1.26506123  -0.68685285 -0.44566197
```

We get the same list when we initialise the list with the same starting value:
set.seed(123)
rnorm(10)

This is very useful, when we want to replicate the same “random” results. 10 uniformly distributed random numbers from the interval [100; 200] can be obtained with

runif(10, min=100, max=200)

Often we use random numbers when we simulate (stochastic) processes. To replicate a process we use the command replicate. E.g.

replicate(10, mean(rnorm(100)))

takes 10 times the mean of each 100 pseudo-normally distributed random numbers.

### 2.5 Example Datasets

We just saw that the command c allows us to describe the elements of a vector. For long datasets this is not very convenient. R contains already a lot of example datasets. These datasets are, similar to statistical functions, organised in libraries. To save space and time R does not load all libraries initially. The command library allows us to load a library with a dataset at any time.

The library Ecdat provides a lot of interesting economic datasets. The library memisc gives access to some interesting functions that help us organising our data.

When we need a specific function and we do not know in which library to look for this function we can use the command RSiteSearch or the R Site Search Extension for Firefox.

The dataset BudgetFood is, e.g., contained in the library Ecdat.

data(BudgetFood, package="Ecdat")

To really see the numbers, we can use the command fix:

fix(BudgetFood)

Usually we do not want to see many numbers. Instead we want to derive (in a structured way) a few numbers (parameters, confidence intervals, p-values,...)

The command help aids us in finding out the meaning of the numbers of the different columns of a dataset.
An important command to get a summary is `summary(BudgetFood)`

How can we access specific columns from our dataset? Since R may have several datasets at the same time in its memory, there are several possibilities. One possibility is to append the name of the dataset `BudgetFood` with a `$` and then the name of the column.

```
BudgetFood$age
[1]  43  40  28  60  37  35  40  68  43  51  43  48  51  58  61  53  58  64  60  50  50  47  76  49  44  49
[26]  51  56  63  30  70  29  60  50  56  36  46  43  32  45  34
[ reached getOption("max.print") -- omitted 23932 entries ]
```

This is helpful when we work with several different datasets at the same time. The example also shows that R does not flood our screen with long lists of numbers. Instead we only see the first few numbers, and then the text "omitted ... entries".

When we want to use only one dataset, then the command `attach` is helpful.

```
attach(BudgetFood)
age
[1]  43  40  28  60  37  35  40  68  43  51  43  48  51  58  61  53  58  64  60  50  50  47  76  49  44  49
[26]  51  56  63  30  70  29  60  50  56  36  46  43  32  45  34
[ reached getOption("max.print") -- omitted 23932 entries ]
```

From now on, all variables will first be searched in the dataset `BudgetFood`. When we no longer want this, then we say

```
detach(BudgetFood)
```

A third possibility is the command `with`:

```
with(BudgetFood, age)
[1]  43  40  28  60  37  35  40  68  43  51  43  48  51  58  61  53  58  64  60  50  50  47  76  49  44  49
[26]  51  56  63  30  70  29  60  50  56  36  46  43  32  45  34
[ reached getOption("max.print") -- omitted 23932 entries ]
```

We often use `with` when we use a function and want to refer to a specific dataset in this function. E.g. `hist` shows a histogram:

```
with(BudgetFood, hist(age))
```
Most commands have several options which allow you to fine-tune the result. Have a look at the help-page for `hist` (you can do this with `help(hist)`). Perhaps you prefer the following graph:

```
with(BudgetFood, hist(age, breaks=40, xlab="Age [years]", col=gray(.7), main="Spain"))
```

### 2.6 Graphs

There is more than one way to represent numbers as graphs.
2.7 Basic Graphs

Here are three basic graphs:

```r
with(BudgetFood, {
  hist(age)
  plot(density(age))
  boxplot(age ~ sex, main="Boxplot")
})
```

Two further helpful plots are `ecdf` and `qqnorm`:

```r
x <- sample(BudgetFood$age, 100)
plot(ecdf(x), main="ecdf")
qqnorm(x)
qqline(x)
```

- Sometimes it is obvious how to prepare our data for these functions. Sometimes it is more complicated. Then other commands help and calculate an object that can be plotted (with `plot`)
  - `density`, `ecdf`, `xyplot`...
- Some commands then plot whatever we have prepared:
  - `plot`, `hist`, `boxplot`, `barplot`, `curve`, `mosaicplot`,...
- Yet other commands add something to an existing plot:
  - `points`, `text`, `lines`, `abline`, `qqline`...

### 2.7.1 Plotting functions

We can plot functions of `x` with `curve`. 
2.7.2 Empty plots

Sometimes it is helpful to start with an empty plot. Then we have to help `plot` a little bit. Usually, `plot` can guess from the data the limits and labels of the axes. With an empty plot we have to specify them explicitly.

```r
plot(NULL, xlim=c(0,10), ylim=c(-3,6), xlab="x", ylab="y", main="an empty plot")
```

2.7.3 Line type

Almost all commands that draw lines follow the following conventions:

- `lty` linetype ("dashed", "dotted", or simply a number)
2.7.4 Points

The character used to draw points is determined with `pch`.

```
range=1:20
plot(range,range/range,pch=range,frame=FALSE)
text(range,range/range+.2,range)
```

![Points](image)

- `lwd` linewidth (a number)
- `col` colour ("red", "green", gray(0.5))

2.7.5 Legends

When we use more than one line or more than one symbol in our plot we have to explain their meaning. This is done in a legend.

Usually `legend` gets as an option a vector of linetypes `lty` and symbols `pch`. They will be used to construct example lines and symbols next to the actual text of the legend. If the `lty` or `pch` is `NA`, then no line or point is drawn.

```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="empty plot")
legend("topleft",c("Text 1","more Text","even more"),lty=1:3,pch=1:3)
legend("bottomright",c("no line no symbol","line only","line and symbol","symbol only"),
       lty=c(NA,2,3,NA),pch=c(NA,NA,3,4),bg="white")
```
2.7.6 Auxiliary lines

The command `abline` allows us to add auxiliary lines to a plot.

```r
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="main title")
abline(h=2:6,lty="dotted")
abline(v=5,lty="dashed")
abline(a=-1,b=1,lwd=5,col=grey(.7))
legend("bottomright",c("h","v","a/b"),lty=c("dotted","dashed","solid"),col=c("black","black",grey(.7)),lwd=c(2,2,5))
```

`abline` knows the following important parameters:

- `h` for horizontal lines
- \( v = \) for vertical lines
- \( a=\ldots, \ b=\ldots \) for lines with intercept \( a \) and slope \( b \)

Note, that these arguments can be vectors if we want to draw several lines at the same time.

### 2.7.7 Axes

The options \( \text{log}='x' \), \( \text{log}='y' \) or \( \text{log}='xy' \) determine whether which axis is shown in a logarithmic style.

```r
data(PE, package="Ecdat")
xx<-as.data.frame(PE)
attach(xx)

plot(price, earnings) plot(price, earnings, log="x") plot(price, earnings, log="xy")
```

To gain more flexibility \( \text{axis} \) can draw a wide range of axes. Before using \( \text{axis} \) the previous axes can be removed entirely (\( \text{axes}=\text{FALSE} \)) or suppressed selectively (\( \text{xaxt}=\text{n} \) or \( \text{yaxt}=\text{n} \)).

```r
plot(price, earnings, log="xy",axes=FALSE) plot(price, earnings, log="xy",xaxt="n")
axis(1,at=c(5,10,20,40,80,160,320,640,1280))
```
If we specify a lot of axes labels, as in the example above, R does not print them all if they overlap.

### 2.8 Fancy math

R can also render more than only textual labels with `plotmath`:

```r
plot(price, earnings, xlab='\$\pi_1\$', ylab='\$\hat{\gamma}_0\$', main="the $\int_\theta^{\bar{\theta}} \sqrt{\xi} d\phi$
abline(lm(earnings~price))
legend("bottomright", c("legend","\$\xi^2\$","line $\phi$"),pch=c(NA,1,NA),lty=c(NA,NA,1))
```

![Graph showing the integral \(\int_\theta^{\bar{\theta}} \sqrt{\xi} d\phi\) and some legend entries.](image)

**2.8.1 Several diagrams**

#### Diagrams side by side

To put several diagrams on one plot side by side we can call `par(mfrow=c(...))` or `layout` or `split.screen`.

```r
par(mfrow=c(1,2))
with(BudgetFood, {
  hist(age)
  plot(density(age))
})
```
Superimposed graphs

- Anything that can create lines or points (like density or ecdf) can immediately be added to an existing plot.
- Plot-objects that would otherwise create a new figure (like plot, hist, or curve) can be added to an existing plot with the optional parameter add=TRUE.

```r
with(BudgetFood, {
  plot(density(age), lwd=2)
  lines(density(age[sex=="man"], na.rm=TRUE),
        lty=3, lwd=2, col="green")
  hist(age, freq=FALSE, add=TRUE)
  curve(dnorm(x, mean(age), sd(age)),
        add = TRUE, lty=2, col="blue")
})
```

Coplots We will discuss coplots in section ??.
2.9 Tables

**Tables of frequencies** The command `table` calculates a table of frequencies. Here we show only the first 16 columns:

```r
with(BudgetFood, table(sex, age))[1:16]
```

<table>
<thead>
<tr>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>man</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>36</td>
</tr>
<tr>
<td>22</td>
<td>37</td>
</tr>
<tr>
<td>23</td>
<td>87</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
</tr>
<tr>
<td>25</td>
<td>132</td>
</tr>
<tr>
<td>26</td>
<td>201</td>
</tr>
<tr>
<td>27</td>
<td>210</td>
</tr>
<tr>
<td>28</td>
<td>248</td>
</tr>
<tr>
<td>29</td>
<td>264</td>
</tr>
<tr>
<td>30</td>
<td>329</td>
</tr>
<tr>
<td>31</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>woman</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>

**Other statistics** The command `aggregate` groups our data by levels of one or several factors and applies a function to each group. In the following example the factor is `sex`, the function is the mean which is applied to the variable `age`.

```r
with(BudgetFood, aggregate(age ~ sex, FUN=mean))
```

<table>
<thead>
<tr>
<th>sex</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>man</td>
<td>49.08985</td>
</tr>
<tr>
<td>woman</td>
<td>59.47445</td>
</tr>
</tbody>
</table>

2.10 Regressions

Simple regressions can be estimated with `lm`. The operator `~` allows us to describe the regression equation. The dependent variable is written on the left side of `~`, the independent variables are written on the right side of `~`.

```r
lm (wfood ~ totexp, data=BudgetFood)
```

Call: lm(formula = wfood ~ totexp, data = BudgetFood)

Coefficients:
(Intercept) totexp
0.4950397225 -0.0000001348

The result is a bit terse. More details are shown with the command `summary`.

```r
summary(lm (wfood ~ totexp, data=BudgetFood))
```

Call: lm(formula = wfood ~ totexp, data = BudgetFood)

Residuals:

<table>
<thead>
<tr>
<th>Min</th>
<th>1Q</th>
<th>Median</th>
<th>3Q</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.49307</td>
<td>-0.09374</td>
<td>-0.01002</td>
<td>0.08617</td>
<td>1.06182</td>
</tr>
</tbody>
</table>
Coefficients:

|               | Estimate | Std. Error | t value | Pr(>|t|) |
|---------------|----------|------------|---------|----------|
| (Intercept)   | 0.495039722500 | 0.001561819134 | 316.96 | <2e-16 *** |
| totexp        | -0.000000134849 | 0.000000001459 | -92.41 | <2e-16 *** |

---

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.1422 on 23970 degrees of freedom
Multiple R-squared: 0.2627, Adjusted R-squared: 0.2626
F-statistic: 8540 on 1 and 23970 DF, p-value: < 2.2e-16

### 2.11 Starting and stopping R

Whenever we start R, the program attempts to find a file `.Rprofile`, first in the current working directory, then in the home directory. If the file is found, it is “sourced”, i.e. all R commands in this file are executed. This is useful when we want to run the same commands whenever we start R. The following line

```r
options(browser = "/usr/bin/iceweasel")
```

in `.Rprofile` makes sure that the help system of R always uses `iceweasel`.

Also when we quit R with the command `q()`, the application tries to make our life easier.

`q()`

R first asks us

**Save workspace image? [y/n/c]:**

Here we have the possibility to save all the data that we currently use (and that are in our workspace) in a file `.Rdata` in the current working directory. When we start R for the next time (from this directory) R automatically reads this file and we can continue our work.

### 3 Organising work

#### 3.1 Scripts

Most of the practical work in data analysis and statistics can be see as a sequence of commands to a statistical software.

How can we run these commands?

- Execute a command in the command window (or with mouse and dialog boxes)
  - clumsy
  - hard to repeat actions
  - hard to replicate what we did and why we did it (logs don’t really help).
- hard to find mistakes (structure of the mistake is easy to overlooked).

- Write a file (.R or .do) and execute single lines (or small regions) from the file while editing the file.
  - great way to creatively develop code line by line. Not reproducible since the file changes permanently.
  - one window with the file another window with mainly the R output

- Write a source file (.R or .do), open it in an editor and then always execute the entire file (while editing the file).
  - great way to creatively develop larger parts of code

- Source “public” files (.R or .do) from a “master file”

```r
source("read_data_160715.R")
source("clean_data_160715.R")
source("create_figures_160715.R")
```

This is the first step to reproducible research. When our script seems to do what it is supposed to do, we make it “public”, give it a unique name, and never change it again.

- From a master file, first source a file which defines functions. Then call these functions.

```r
source("functions_XYZ_160715.R")
read_data()
clean_data()
create_figures()
```

- Advantages of using source files (with or without functions):
  - We keep a record of our work.
  - We can work incrementally, fix mistakes and introduce small changes (if we refer to a public file, we should work on a copy of this file with a new name).
  - We can use the editor of our choice (Emacs is a nice editor)

- Advantage of using functions:
  - functions can take parameters.
  - several functions go in one file (still do not harm each other).
  - Systematic changes are easier with only one file.
  - Regardless whether we divide our work into source files or into functions: This division allows us to save time. Some of these steps take a lot of time. Once they work, we do not have to do them over and over again.
3.1.1 Robust scripts

How can we make our scripts "robust"? Remember:

- The structure of the data may change over time.
  - New variables might come with new treatments of our experiment.
  - New treatments might require that we code variables differently.
- The scripts may not only run on our computer.
- The scripts are not always sourced in the same context.
- Our random number generator may start from different seeds.

3.1.2 Robustness towards different computers

We better use relative pathnames.

Assume that on my computer the script is stored in

/home/oliver/projectXYX/R

next to it we have

/home/oliver/projectXYX/data/munich/1998/test.Rdata

From the script I might call either (absolute path)

```r
load(file="/home/oliver/projectXYX/data/munich/1998/test.Rdata")
```

or (relative path)

```r
load(file="../data/munich/1998/test.Rdata")
```

The latter assumes that there is a file

../data/munich/1998/test.Rdata

next to the script. But it does not assume that everything is in

/home/oliver/projectXYZ

Hence, the latter works even if my coauthor has stored everything as

C:/users/eva/PhD/projectXYZ/R
C:/users/eva/PhD/projectXYZ/data/munich/1998/test.Rdata

If a lot happens in ../data/munich/1998/ anyway, use the `setwd` command

```r
setwd("../data/munich/1998/")
...
load(file="test.Rdata")
```

(and remember to make the `setwd` relative, i.e. avoid the following:...
3.1.3 Robustness towards changes in context

Assume we have the following two files:

```r
# script1.R
load("someData.Rdata")
# now two variables, x and y are defined
source("script2.R")
```

The content of `script2.R` might be this:

```r
# script2.R
est <- lm ( y ~ x)
```

In this example `script2.R` assumes that variables `y` and `x` are defined. As long as `script2.R` is called in this context, everything is fine. Changing `script1.R` might have unexpected side effects since we transport variables from one script to the other. The call

```r
source("script2.R")
```

does not reveal how `y` and `x` are used by the script.

3.1.4 Functions increase robustness

```r
# script1.R
source("script2.R")
load("someData.Rdata")
myFunction(y,x)
```

```r
# script2.R
# defines myFunction
myFunction <- function(y,x) {
  est <<- lm ( y ~ x)
}
```

Now `script2.R` only defines a function. The function has arguments, hence, when we use it in `script1.R` we realise which variable goes where. Note that the function takes arguments. This is more elegant (and less risky) than to write functions like this one:

```r
# script2.R
# defines myFunction
myFunction <- function() {
  est <<- lm ( y ~ x)
}
```
and then say

```r
# script1.R
source("script2.R")
load("someData.Rdata")
x <- ...
y <- ...
myFunction()
```

It will still work, but later it will be less clear to us that the assignments before the function call are essential for the function.

```r
myFunction <- function(y,x) {
  est <<- lm ( y ~ x)
}
```

This function has a side effect. It changes a variable `est` outside the function. Often it is less confusing to define functions with return values and no side effects.

```r
myFunction <- function(y,x) {
  lm ( y ~ x)
}
```

When we call this function later as

```r
est <- myFunction(y,x)
```

it is clear where the result of the function goes.

**Recap**

- Functions which use *global variables*: risky
- Functions with *side effects*: risky
- Functions which only use arguments and return values: often better

Note: If we use *scripts* instead of *functions*:

→ Scripts must use *global variables* and can only produce *side effects*.

→ Scripts are more likely to lead to *mistakes* than functions.

→ Replace scripts by *functions* (with arguments) whenever possible.

### 3.2 Calculations that take a lot of time

If a sequence of functions takes a lot of time to run, let it generate intermediate data. Our master-R-file could look like this:
3.3 Nested functions

If our functions become long and complicated, we can divide them into small chunks.

```r
doAnalysis <- function () {
  firstStepAnalysis()
  secondStepAnalysis()
  thirdStepAnalysis()
}
```

```r
firstStepAnalysis <- function() {
  ...
}
```

```r
secondStepAnalysis <- function() {
  ...
}
```

Actually, if we need some functions only within a specific other function then we can define them `within` this function:

```r
doAnalysis <- function () {
  firstStep <- function() {
    ...
  }
  secondStep <- function() {
    ...
  }
  firstStep()
  secondStep()
  thirdStep()
}
```
• Advantage: These functions are only visible from within doAnalysis and can do no harm elsewhere (where we, perhaps, defined functions with the same name that do different things).

Nesting of functions has three advantages:
• It structures our work.
• It facilitates debugging.
• It facilitates communication with coauthors. (we can say: “...there is a problem in thirdStep in doAnalysis...”)

3.4 Reproducible randomness

```
set.seed(123)
```

Random numbers affect our results:
• Simulation
• MCMC samples
• Bootstrapping
• Approximate permutation tests
• Selection of training and confirmation samples
• ...

3.5 Recap — writing scripts and using functions

• If there is a systematic structure in our problem, then we can exploit it
• If we make mistakes, we make them systematically!

```
N <- 100
profit88 <- rnorm(N)
profit89 <- rnorm(N)
profit98 <- rnorm(N)
myData <- data.frame(profit88,profit89,profit98)
```

Compare
```
t.test(profit88,data=myData)$p.value
t.test(profit89,data=myData)$p.value
t.test(profit98,data=myData)$p.value
```
The first looks simpler. The second is more robust against
- a change in the dataset (instead of myData we now use myDataClean)
- a change in the names of the variables (profit becomes Profit_)
- adding another profit-variable (profit2016...)
- typos (use profit88 twice, instead of profit88 and profit89 once each.

### 3.6 Human readable scripts
- Weaving and knitting → we do this later
- Comments at the beginning of each file

```r
# scriptExample160715.R
# the purpose of this script is to illustrate the use of
# comments
# first version: 160715
# this version: 160715
# last change by: Oliver
# requires: test160715.Rdata, someFunctions160715.R
# provides: ...
#
set.seed(123)
```

- Comments at the beginning of each function

```r
# exampleFun transforms two vectors into an example
# side effects: ...
# returns: ...
# exampleFun <- function(x,y) {
  ...
}
```

- Comment non-obvious steps

```r
# to detect outliers we use lrt-method.
# We have tried depth.trim and depth.pond
# but they produce implausible results...
outl <- foutliers(data,method="lrt")
```
• Document your thoughts in your comments

... # 16/07/21: although I thought that age should not affect
# profits, it does here! I also checked
# my specification and it still does.
# Perhaps age is a proxy for income.
# Unfortunately we do not have data on
# income here.
...

• Formatting
  Compare

```
lm ( s1 ~ trust + ineq + sex + age + latitude )
lm ( otherinvestment ~ trust + ineq + sex + age + latitude )
```

with

```
lm ( s1 ~ trust + ineq + sex + age + latitude )
lm ( otherinvestment ~ trust + ineq + sex + age + latitude )
```

Insert linebreaks  Compare

```
lm ( otherinvestment ~ trust + ineq + sex + age + latitude, data=trustExp, subset=sex="female" )
```

with

```
lm ( otherinvestment ~ trust + ineq + sex + age + latitude, 
    data=trustExp, 
    subset=sex="female" )
```

• Variables names
  short but not too short

```
lm ( otherinvestment ~ trustworthiness + inequalityaversion + sexOfProposer + ageOfProposer + latitudeOfProposer )
lm ( otherinvestment ~ trust + ineq + sex + age + latitude )
lm ( oi ~ t + i + s + a + l1 + l2)
lm ( R100234 + R100412 + R100017 + R100178 + R100671 + R100229 )
```

We will say more about variable names in section 6.3.

• Abbreviations in scripts
  R (and other languages too) allows you to refer to parameters in functions with names:
To save space, we can abbreviate these names:

```r
qnorm(p=.01, low=FALSE)
```

To save space, we can abbreviate these names:

```r
qnorm(p=.01, low=FALSE)
```

4 Some programming techniques

4.1 Debugging functions

```r
library(Ecdat)
data(Kakadu)
head(Kakadu)
```

```
   lower upper answer recparks jobs lowrisk wildlife future aboriginal finben
1     1     2       nn      3      1       5       5      1      1      1
```

```
   mineparks moreparks gov envcon v parks t v env conservation sex age schooling
1     4     5       1     yes    yes     1      no    male    27     3
   income major
1     25     no
```

[ reached getOption("max.print") -- omitted 5 rows ]

**general strategies: debug the function with a simple example**

```r
sqMean <- function (x) {
  z <- mean(x)
  z^2
}
sqMean(Kakadu$lower)
```

```
[1] 2361.471
```

Is this correct? Take a (simpler) subsample of the data:

```r
(xx <- sample(Kakadu$lower, 3))
```

```
[1] 100  0  20
```

```r
sqMean(xx)
```

```
[1] 1600
```
Assume that we still do not trust the function. `debug` allows us to debug a function. `ls` allows us to list the variables in the current environment.

```r
debug(sqMean)

debugging in: sqMean(xx)
dump at <text>#1: {
    z <- mean(x)
    z^2
}
dump at <text>#2: z <- mean(x)
dump at <text>#3: z^2
exiting from: sqMean(xx)
[1] 1600

undump(sqMean)
```

If the function returns with an error, it helps to set

```r
options(error=recover)
```

In the following function we refer to the variable `xxx` which is not defined. The function will, hence, fail. With `options(error=recover)` we can inspect the function at the time of the failure.

```r
sqMean <- function (x) {
    z <- mean(xxx)
    z^2
}

sqMean(xx)

Error in mean(xxx) (from #2) : object 'xxx' not found
Enter a frame number, or 0 to exit
1: sqMean(xx)
2: #2: mean(xxx)

Selection: 1
Called from: top level
Browse[1]> xxx
Error during wrapup: object 'xxx' not found
Browse[1]> x
[1] 20 0 0 250 100 50 20 50 50 100
Browse[1]> Q
```
4.2 Lists of variables

To make the analysis more consistent.
Whenever things repeat, we define them in variables at the top of the paper:

```r
models <- list(a="income",
               b="income + age + sex",
               c="income + age + sex + conservation + vparks")
```

(We use here character strings to represent parts of formulas. Alternatively, we could also
store objects of class `formula`. However, manipulating these objects is not always to obvious.
To keep things simple, we will use character strings here.) Later in the paper we compare
the different models:

```r
mylm <- function (m) lm(paste("as.integer(answer) ~ ",m),data=Kakadu)
lmList<-lapply(models,mylm)
class(lmList)<-c("list","by")
mtable(lmList)
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>2.122***</td>
<td>2.765***</td>
<td>2.648***</td>
</tr>
<tr>
<td></td>
<td>(0.035)</td>
<td>(0.065)</td>
<td>(0.076)</td>
</tr>
<tr>
<td>income</td>
<td>0.003*</td>
<td>0.003*</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>(0.001)</td>
<td>(0.001)</td>
<td>(0.001)</td>
</tr>
<tr>
<td>age</td>
<td>-0.013***</td>
<td>-0.012***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.001)</td>
<td>(0.001)</td>
<td></td>
</tr>
<tr>
<td>sex: male/female</td>
<td>-0.196***</td>
<td>-0.190***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.043)</td>
<td>(0.043)</td>
<td></td>
</tr>
<tr>
<td>conservation: yes/no</td>
<td></td>
<td>0.215**</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.083)</td>
<td></td>
</tr>
<tr>
<td>vparks: yes/no</td>
<td>0.120*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.047)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-squared | 0.003 | 0.073 | 0.080 |
adj. R-squared | 0.002 | 0.071 | 0.078 |
sigma | 0.902 | 0.870 | 0.867 |
F | 4.697 | 47.635 | 31.708 |
p | 0.030 | 0.000 | 0.000 |
Log-likelihood | -2402.751 | -2336.158 | -2328.842 |
Deviance | 1484.485 | 1380.118 | 1369.109 |
AIC | 4811.503 | 4682.316 | 4671.684 |
BIC | 4828.034 | 4709.868 | 4710.257 |
N | 1827 | 1827 | 1827 |

Now we use the same explanatory variables to explain a different dependent variable:
```r
mylogit <- function(m) glm(paste("answer=='yy' ~ ",m),
                     data=Kakadu,family=binomial(link=logit))
logitList <- lapply(models,mylogit)
class(logitList)<-c("list","by")
mtable(logitList)
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>-0.121</td>
<td>1.100***</td>
<td>0.796***</td>
</tr>
<tr>
<td></td>
<td>(0.078)</td>
<td>(0.155)</td>
<td>(0.181)</td>
</tr>
<tr>
<td>income</td>
<td>0.008**</td>
<td>0.009**</td>
<td>0.008*</td>
</tr>
<tr>
<td></td>
<td>(0.003)</td>
<td>(0.003)</td>
<td>(0.003)</td>
</tr>
<tr>
<td>age</td>
<td>-0.025***</td>
<td>-0.023***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.003)</td>
<td>(0.003)</td>
<td></td>
</tr>
<tr>
<td>sex: male/female</td>
<td>-0.343***</td>
<td>-0.332**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.102)</td>
<td>(0.102)</td>
<td></td>
</tr>
<tr>
<td>conservation: yes/no</td>
<td>0.345</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.202)</td>
<td></td>
</tr>
<tr>
<td>vrparks: yes/no</td>
<td>0.334**</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.110)</td>
<td></td>
</tr>
<tr>
<td>Aldrich-Nelson R-sq.</td>
<td>0.005</td>
<td>0.051</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.057</td>
<td></td>
<td></td>
</tr>
<tr>
<td>McFadden R-sq.</td>
<td>0.003</td>
<td>0.039</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.044</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cox-Snell R-sq.</td>
<td>0.005</td>
<td>0.052</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.059</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nagelkerke R-sq.</td>
<td>0.006</td>
<td>0.069</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.078</td>
<td></td>
<td></td>
</tr>
<tr>
<td>phi</td>
<td>1.000</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Likelihood-ratio</td>
<td>8.540</td>
<td>97.606</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110.776</td>
<td></td>
<td></td>
</tr>
<tr>
<td>phi</td>
<td>1.000</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log-likelihood</td>
<td>-1261.282</td>
<td>-1216.749</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-1210.164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deviance</td>
<td>2522.564</td>
<td>2433.498</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2420.328</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AIC</td>
<td>2526.564</td>
<td>2441.498</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2432.328</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>2537.584</td>
<td>2463.540</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2465.391</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>1827</td>
<td>1827</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1827</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similarly, we might define at the beginning of the paper...

- lists of random effects
- lists of variables to group by
- palettes for plots

### 4.3 Return values of functions

Most functions do not only return a number (or a vector) but rather complex objects. In R `str()` helps us to learn more about the structure of these objects. (In Stata similar return values are provided by `return`, `ereturn`, and `sreturn`
```r
lm1 <- mylm(models[[1]])
str(lm1)

List of 12
$ coefficients : Named num [1:2] 2.12202 0.00278
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "income"
$ residuals : Named num [1:1827] -1.19 -1.15 -1.19 -1.19 -1.22 ...
  ..- attr(*, "names")= chr [1:1827] "1" "2" "3" "4" ...
$ effects : Named num [1:1827] -93.28 1.95 -1.17 -1.17 -1.21 ...
  ..- attr(*, "names")= chr [1:1827] "(Intercept)" "income" "" ""
$ rank : int 2
$ fitted.values: Named num [1:1827] 2.19 2.15 2.19 2.19 2.22 ...
  ..- attr(*, "names")= chr [1:1827] "1" "2" "3" "4" ...
$ assign : int [1:2] 0 1
$ qr :List of 5
  ..$ qr : num [1:1827, 1:2] -42.7434 0.0234 0.0234 0.0234 0.0234 ...
  。。。- attr(*, "dimnames")=List of 2
  。。。 。。。$ : chr [1:1827] "1" "2" "3" "4" ...
  。。。 。。。$ : chr [1:2] "(Intercept)" "income"
  。。。- attr(*, "assign")= int [1:2] 0 1
  。。。$ qr: num [1:2] 1.02 1.02
  。。。$ pivot: int [1:2] 1 2
  。。。$ tol : num 0.0000001
  。。。$ rank : int 2
  。。。- attr(*, "class")= chr "qr"
$ df.residual : int 1825
$ xlevels : Named list()
$ call : language lm(formula = paste("as.integer(answer) ~ ", m), data = Kakadu)
$ terms :Classes lm(formula = as.integer(answer) ~ income, data = Kakadu)
  ..- attr(*, "variables")= language list(as.integer(answer), income)
  。。。- attr(*, "factors")= int [1:2, 1] 0 1
  。。。- dimnames=List of 2
  。。。 。。。$ : chr [1:2] "as.integer(answer)" "income"
  。。。 。。。$ : chr "income"
  。。。- attr(*, "term.labels")= chr "income"
  。。。- attr(*, "order")= int 1
  。。。- attr(*, "intercept")= int 1
  。。。- attr(*, "response")= int 1
  。。。- .Environment"= <environment: 0x81b19adb8>
  。。。- attr(*, "predvars")= language list(as.integer(answer), income)
  。。。- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  。。。- attr(*, "names")= chr [1:2] "as.integer(answer)" "income"
$ model :'data.frame': 1827 obs. of 2 variables:
  ..$ as.integer(answer): int [1:1827] 1 1 1 1 1 1 1 1 1 1 ...
  。。。- attr(*, "terms")=Classes 'terms', 'formula' language as.integer(answer) ~ income
  。。。- attr(*, "variables")= language list(as.integer(answer), income)
  。。。- attr(*, "factors")= int [1:2, 1] 0 1
  。。。- dimnames=List of 2
  。。。 。。。$ : chr [1:2] "as.integer(answer)" "income"
  。。。 。。。$ : chr "income"
  。。。- attr(*, "term.labels")= chr "income"
```
There are at least two ways to extract data from these objects:

- Extractor functions

```r
coef(lm1)
```

```
(Intercept)  income
 2.122018102 0.002781938
```

```r
vcov(lm1)
```

```
     (Intercept)     income
(Intercept) 0.00121806075 -0.000035685812
 income -0.00003568581  0.000001647787
```

```r
hccm(lm1)
```

```
     (Intercept)     income
(Intercept) 0.00123366056 -0.000036812592
 income -0.00003681259  0.000001719666
```

```r
logLik(lm1)
```

```
'log Lik.' -2402.751 (df=3)
```

```r
effects(lm1)
fitted.values(lm1)
residuals(lm1)
```

(the equivalent in Stata are postestimation commands)

- Whatever is a list item can also be accessed directly:

```r
lm1$coefficients
lm1$residuals
lm1$fitted.values
lm1$residuals
```

Note: Some interesting values are not provided by the `lm`-object itself. These can often be accessed as part of the `summary`-object.
4.4 Repeating things

**Looping** The simplest way to repeat a command is a loop:

```r
for (i in 1:10) print(i)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

If the command is a sequence of expressions, we have to enclose it in braces.

```r
for (i in 1:10) {
  x <- runif(i)
  print(mean(x))
}
```

```
[1] 0.3565607
[1] 0.9663778
[1] 0.5063639
[1] 0.4378409
[1] 0.487012
[1] 0.5853594
[1] 0.3502112
[1] 0.499148
[1] 0.5078825
[1] 0.4557163
```

**Avoiding loops** In R loops should be avoided. It is more efficient (faster) to apply a function to a vector.

```r
sapply(1:10,print)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
```
Or, the more complex example:

```r
sapply(1:10, function(i) {
  x <- runif(i)
  mean(x)
})
```

```
[1] 0.6538133 0.4623162 0.8092458 0.4935831 0.6997635 0.4856793 0.6413399
     0.5610393 0.5781580 0.4712342
```

Note that `sapply` already returns a vector which is in many cases what we want anyway.

In the above examples we applied a function to a vector. Sometimes we want to apply functions to a matrix.

### Applying a function along one dimension of a matrix

In the following example we apply a function along the second dimension of the dataset Kakadu.

```r
apply(Kakadu, 2, function(x) mean(as.integer(x)))
```

```r
cbind(apply(Kakadu, 2, function(x) mean(as.integer(x))))
```

<table>
<thead>
<tr>
<th>lower</th>
<th>upper</th>
<th>answer</th>
<th>recparks</th>
<th>jobs</th>
<th>lowrisk</th>
</tr>
</thead>
<tbody>
<tr>
<td>48.594964</td>
<td>536.714286</td>
<td>NA</td>
<td>3.688560</td>
<td>2.592228</td>
<td>2.790367</td>
</tr>
<tr>
<td>wildlife</td>
<td>future</td>
<td>aboriginal</td>
<td>finben</td>
<td>mineparks</td>
<td>moreparks</td>
</tr>
<tr>
<td>4.739464</td>
<td>4.466886</td>
<td>3.569787</td>
<td>2.915709</td>
<td>3.643678</td>
<td>3.864806</td>
</tr>
<tr>
<td>1.083196</td>
<td>NA</td>
<td>NA</td>
<td>1.785441</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>age</td>
<td>schooling</td>
<td>income</td>
<td>major</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>42.968254</td>
<td>3.683634</td>
<td>21.656814</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```r
cbind(apply(Kakadu, 2, function(x) mean(as.integer(x))))
```

<table>
<thead>
<tr>
<th>[,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower</td>
</tr>
<tr>
<td>upper</td>
</tr>
<tr>
<td>answer</td>
</tr>
<tr>
<td>recparks</td>
</tr>
<tr>
<td>jobs</td>
</tr>
<tr>
<td>lowrisk</td>
</tr>
<tr>
<td>wildlife</td>
</tr>
<tr>
<td>future</td>
</tr>
<tr>
<td>aboriginal</td>
</tr>
<tr>
<td>finben</td>
</tr>
<tr>
<td>mineparks</td>
</tr>
</tbody>
</table>
44 Workflow of statistical data analysis — 4  SOME PROGRAMMING TECHNIQUES

moreparks  3.864806
gov         1.083196
envcon      NA
vparks      NA
tvenv       1.785441
conservation NA
sex         NA
age         42.968254
schooling   3.683634
income      21.656814
major       NA

Rectangular and ragged arrays  Rectangular array:

<table>
<thead>
<tr>
<th>wide</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>hor  vert  x</td>
<td></td>
</tr>
<tr>
<td>a  A  1</td>
<td></td>
</tr>
<tr>
<td>b  A  2</td>
<td></td>
</tr>
<tr>
<td>c  A  3</td>
<td></td>
</tr>
<tr>
<td>a  B  4</td>
<td></td>
</tr>
<tr>
<td>b  B  5</td>
<td></td>
</tr>
<tr>
<td>c  B  6</td>
<td></td>
</tr>
</tbody>
</table>

Ragged array:

<table>
<thead>
<tr>
<th>wide</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>hor  vert  x</td>
<td></td>
</tr>
<tr>
<td>a  A  2</td>
<td></td>
</tr>
<tr>
<td>c  A  3</td>
<td></td>
</tr>
<tr>
<td>a  B  4</td>
<td></td>
</tr>
<tr>
<td>b  B  5</td>
<td></td>
</tr>
</tbody>
</table>

Applying a function to each element of a ragged array  In R ragged arrays can be represented as datasets grouped by one or more factors. These variables describe which records belong together (e.g. to the same person, year, firm,...)

In the following example we use the dataset Fatality. This dataset contains for each state of the United States and for each year in 1982 to 1988 in mrand the traffic fatality rate (deaths per 10000).

data(Fatality)
head(Fatality)

| state year mrand beertax mlda jaild comserd vmiles unrate perinc |
|-----------|------------------|----------------|-----------|----------------|------------|-------|-------|-------|
| 1         | 1 1982 2.12836 1.539379 19.00 no no 7.233887 14.4 10544.15 |
| 2         | 1 1983 2.34848 1.788991 19.00 no no 7.836348 13.7 10732.80 |
| 3         | 1 1984 2.33643 1.714286 19.00 no no 8.262990 11.1 11108.79 |
| 4         | 1 1985 2.19348 1.652542 19.67 no no 8.726917 8.9 11332.63 |

[ reached getOption("max.print") -- omitted 2 rows ]
by(Fatality, list(Fatality$year), function(x) mean(x$mrall))

: 1982
  [1] 2.089106
------------------------------------------------------------
: 1983
  [1] 2.007846
------------------------------------------------------------
: 1984
  [1] 2.017122
------------------------------------------------------------
: 1985
  [1] 1.973671
------------------------------------------------------------
: 1986
  [1] 2.065071
------------------------------------------------------------
: 1987
  [1] 2.060696
------------------------------------------------------------
: 1988
  [1] 2.069594
------------------------------------------------------------

by(Fatality, list(Fatality$state), function(x) mean(x$mrall))

: 1
  [1] 2.412627
------------------------------------------------------------
: 4
  [1] 2.7059
------------------------------------------------------------
: 5
  [1] 2.435336
------------------------------------------------------------
: 6
  [1] 1.904977
------------------------------------------------------------
: 8
  [1] 1.866981
------------------------------------------------------------
: 9
  [1] 1.463509
------------------------------------------------------------
: 10
  [1] 2.068231
------------------------------------------------------------
: 12
  [1] 2.477799
------------------------------------------------------------
: 13
  [1] 2.401569
------------------------------------------------------------
by does not return a vector but an object of class by. If we actually need a vector we have to use `c` and `sapply`.

In the following example we let `by` actually return two values.

```r
byObj <- by(Fatality, list(Fatality$year),
            function(x) c(year=median(x$year),
                           fatality=mean(x$mrall),
                           meanbeertax=mean(x$beertax)))
sapply(byObj,c)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fatality</td>
<td>2.0891059</td>
<td>2.007846</td>
<td>2.0171225</td>
<td>1.9736708</td>
<td>2.0650710</td>
</tr>
<tr>
<td>meanbeertax</td>
<td>0.5302734</td>
<td>0.532393</td>
<td>0.5295902</td>
<td>0.5169272</td>
<td>0.5086639</td>
</tr>
</tbody>
</table>

```

```r
xx <- data.frame(t(sapply(byObj,c)))
xyplot(fatality ~ meanbeertax, type="l", data=xx) +
       layer(with(xx, panel.text(label=year, y=fatality, x=meanbeertax, adj=c(1,1))))
```
We can do more complicated things in by. In the following example we calculate a regression. To get only the coefficients from the regression (and not fitted values, residuals, etc.) we use the extractor function `coef`.

```r
byObj <- by(Fatality, list(Fatality$year), function(x) 
  lm(mrall ~ beertax + jaild, data=x))
sapply(byObj, coef)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>1.9079924</td>
<td>1.7503870</td>
<td>1.6768093</td>
<td>1.6567128</td>
<td>1.7108657</td>
<td>1.7188081</td>
</tr>
<tr>
<td>beertax</td>
<td>0.1824028</td>
<td>0.2991742</td>
<td>0.4066922</td>
<td>0.4057889</td>
<td>0.4944595</td>
<td>0.4920275</td>
</tr>
<tr>
<td>jaildyes</td>
<td>0.4500807</td>
<td>0.3625151</td>
<td>0.4283417</td>
<td>0.3430232</td>
<td>0.3286131</td>
<td>0.3369277</td>
</tr>
</tbody>
</table>

1988
(Intercept) 1.7411593
beertax 0.4509099
jaildyes 0.3842788

xx <- data.frame(t(sapply(byObj, coef)))
xyplot(beertax ~ jaildyes, type="l", data=xx) +
  layer(with(xx, panel.text(label=rownames(xx), y=beertax, x=jaildyes, adj=c(1,1))))
```
by is very complex. It offers the entire subset of the *dataframe*, as defined by the index variable, to the function.

Sometimes we want simply to apply a function of only a *vector* along a ragged array.

```r
with(Fatality, aggregate(mrall ~ year, FUN=mean))
```

<table>
<thead>
<tr>
<th>year</th>
<th>mrall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.089106</td>
</tr>
<tr>
<td>2</td>
<td>2.007846</td>
</tr>
<tr>
<td>3</td>
<td>2.017122</td>
</tr>
<tr>
<td>4</td>
<td>1.973671</td>
</tr>
<tr>
<td>5</td>
<td>2.065071</td>
</tr>
<tr>
<td>6</td>
<td>2.060696</td>
</tr>
<tr>
<td>7</td>
<td>2.069594</td>
</tr>
</tbody>
</table>

Again, the function (which was `mean` in the previous example) can be defined by us:

```r
with(Fatality, aggregate(mrall ~ year, FUN=function(x) sd(x)/mean(x)))
```

<table>
<thead>
<tr>
<th>year</th>
<th>mrall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3196449</td>
</tr>
<tr>
<td>2</td>
<td>0.3017002</td>
</tr>
<tr>
<td>3</td>
<td>0.2721300</td>
</tr>
<tr>
<td>4</td>
<td>0.2726437</td>
</tr>
<tr>
<td>5</td>
<td>0.2709500</td>
</tr>
<tr>
<td>6</td>
<td>0.2738153</td>
</tr>
<tr>
<td>7</td>
<td>0.2518286</td>
</tr>
</tbody>
</table>
5 Data manipulation

5.1 Subsetting data

There are several ways to access only a part of a dataset:

- Many functions take an option ...,subset=...

\[
\text{lm}(\text{mrall} - \text{beertax} + \text{jaild}, \text{data}=\text{Fatality}, \text{subset} = \text{year} == 1982)
\]

Call:
\[
\text{lm(formula = mrall} - \text{beertax} + \text{jaild, data = Fatality, subset = year == 1982)}
\]

Coefficients:
(Intercept)    beertax    jaildyes
  1.908000  0.182400  0.450083

- The \text{subset()} function

\[
\text{subset(\text{Fatality, year} == 1982 )}
\]

\[
\begin{array}{cccccccccccc}
\text{state} & \text{year} & \text{mrall} & \text{beertax} & \text{mlda} & \text{jaild} & \text{comserd} & \text{vmiles} & \text{unrate} & \text{perinc} \\
1 & 1 & 1982 & 2.12836 & 1.53937948 & 19.0 & no & no & 7.233887 & 14.4 & 10544.152 \\
8 & 4 & 1982 & 2.49914 & 0.21479714 & 19.0 & yes & yes & 6.810157 & 9.9 & 12309.069 \\
15 & 5 & 1982 & 2.38405 & 0.65035802 & 21.0 & no & no & 7.208500 & 9.8 & 10267.303 \\
22 & 6 & 1982 & 1.86194 & 0.10739857 & 21.0 & no & no & 6.858677 & 9.9 & 15797.136 \\
\end{array}
\]

\[
\text{with(\text{subset(\text{Fatality, year} == 1982 ), \text{lm(\text{mrall} - \text{beertax} + \text{jaild})})}
\]

Call:
\[
\text{lm(formula = mrall} - \text{beertax} + \text{jaild)}
\]

Coefficients:
(Intercept)    beertax    jaildyes
  1.908000  0.182400  0.450083

- The first index of the dataset

\[
\text{Fatality[ Fatality$year==1982 , ]}
\]

\[
\begin{array}{cccccccccccc}
\text{state} & \text{year} & \text{mrall} & \text{beertax} & \text{mlda} & \text{jaild} & \text{comserd} & \text{vmiles} & \text{unrate} & \text{perinc} \\
1 & 1 & 1982 & 2.12836 & 1.53937948 & 19.0 & no & no & 7.233887 & 14.4 & 10544.152 \\
8 & 4 & 1982 & 2.49914 & 0.21479714 & 19.0 & yes & yes & 6.810157 & 9.9 & 12309.069 \\
15 & 5 & 1982 & 2.38405 & 0.65035802 & 21.0 & no & no & 7.208500 & 9.8 & 10267.303 \\
22 & 6 & 1982 & 1.86194 & 0.10739857 & 21.0 & no & no & 6.858677 & 9.9 & 15797.136 \\
\end{array}
\]
5.2 Merging data

- Appending two datasets

```r
library(plyr)
rbind.fill(x,y)
```

(In Stata this is done by `append`)

- Matching two datasets (inner join)

```r
merge(x,y)
```

(In Stata this is done by `merge`)

- Joining two datasets (left join)

```r
merge(x,y,all.x=TRUE)
```

(In Stata this is done by `joinby`)

<table>
<thead>
<tr>
<th>Dataset A</th>
<th>Dataset B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Grade</td>
</tr>
<tr>
<td>Eva</td>
<td>2.0</td>
</tr>
<tr>
<td>Mary</td>
<td>1.0</td>
</tr>
<tr>
<td>Mike</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inner join: <code>merge(A,B)</code></th>
<th>Left join: <code>merge(A,B,all.x=TRUE)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Grade</td>
</tr>
<tr>
<td>Eva</td>
<td>2.0</td>
</tr>
<tr>
<td>Eva</td>
<td>2.0</td>
</tr>
<tr>
<td>Mike</td>
<td>3.0</td>
</tr>
<tr>
<td>Mike</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Appending  In the following example we first split the data from an experiment into two parts. Merge helps us to append them to each other.

```r
load("data/160716_060x.Rdata")
experiment1 <- subset(trustGS$subjects,Date="160716_0601")
extperiment2 <- subset(trustGS$subjects,Date="160716_0602")
dim(experiment1)
[1] 0 14
dim(experiment2)
[1] 0 14

library(plyr)
dim(rbind.fill(experiment1,experiment2))
[1] 0 14
```

Joining  A frequent application for a join are tables in z-Tree that have something to do with each other. E.g. the globals and the subjects tables both provide information about each period. Common variables in these tables are Date, Treatment, and Period.

By merging globals with subjects, merge looks up for each record in the subjects table the matching record in the globals table and adds the variables which are not already present in subjects.

```r
head(trustGS$global)
   Date Treatment Period NumPeriods RepeatTreatment
1 130716_0601    1       1         6              0
2 130716_0601    1       2         6              0
3 130716_0601    1       3         6              0
4 130716_0601    1       4         6              0
5 130716_0601    1       5         6              0
6 130716_0601    1       6         6              0

head(trustGS$subject)
    Date Treatment Period Subject Pos Group Offer Receive Return
1 130716_0601    1       1       1 2 1 0.000 1.5300 0.585990
2 130716_0601    1       1       2 2 4 0.000 1.6740 1.131624
     GetBack  country  siblings sex age
1 0.000000 6 1 1 27
2 0.000000 15 3 1 19
[ reached getOption("max.print") -- omitted 4 rows ]
```

In the following example we simply get two more variables in the dataset (NumPeriods and RepeatTreatment). With more variables in globals we would, of course, also get more variables in the merged dataset.
Joining aggregates  A common application for a join is a comparison of our individual data with aggregated data. Let us come back to the Fatalities example. We want to compare the traffic fatality rate `mrall` for each state with the average values for each year.

```r
head(Fatality)

state | year | mrall | beertax | mlda | jaild | comserd | vmiles | unrate | perinc
--- | --- | --- | --- | --- | --- | --- | --- | --- | ---
1 | 1982 | 2.12836 | 1.539379 | 19.00 | no | no | 7.233887 | 14.4 | 10544.15
2 | 1983 | 2.34848 | 1.788991 | 19.00 | no | no | 7.836348 | 13.7 | 10732.80
3 | 1984 | 2.33643 | 1.714286 | 19.00 | no | no | 8.262990 | 11.1 | 11108.79
4 | 1985 | 2.19348 | 1.652542 | 19.67 | no | no | 8.726917 | 8.9 | 11332.63
[ reached getOption("max.print") -- omitted 2 rows ]

Aggregate(c(avgMrall=mean(mrall)) ~ year, data=Fatality)

<table>
<thead>
<tr>
<th>year</th>
<th>avgMrall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1982</td>
<td>2.089106</td>
</tr>
<tr>
<td>1983</td>
<td>2.007846</td>
</tr>
<tr>
<td>1984</td>
<td>2.017122</td>
</tr>
<tr>
<td>1985</td>
<td>1.973671</td>
</tr>
<tr>
<td>1986</td>
<td>2.065071</td>
</tr>
<tr>
<td>1987</td>
<td>2.060696</td>
</tr>
<tr>
<td>1988</td>
<td>2.069594</td>
</tr>
</tbody>
</table>

merge(Fatality, Aggregate(c(avgMrall=mean(mrall)) ~ year, data=Fatality))

<table>
<thead>
<tr>
<th>year</th>
<th>state</th>
<th>mrall</th>
<th>beertax</th>
<th>mlda</th>
<th>jaild</th>
<th>comserd</th>
<th>vmiles</th>
<th>unrate</th>
<th>perinc</th>
<th>avgMrall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1982</td>
<td>1</td>
<td>2.12836</td>
<td>1.539379</td>
<td>19.00</td>
<td>no</td>
<td>no</td>
<td>7.233887</td>
<td>14.4</td>
<td>10544.15</td>
<td>2.089106</td>
</tr>
<tr>
<td>1982</td>
<td>30</td>
<td>3.15528</td>
<td>0.34644747</td>
<td>19.00</td>
<td>yes</td>
<td>no</td>
<td>8.284474</td>
<td>8.6</td>
<td>12033.413</td>
<td>2.089106</td>
</tr>
<tr>
<td>1982</td>
<td>10</td>
<td>2.03333</td>
<td>0.17303102</td>
<td>20.00</td>
<td>no</td>
<td>no</td>
<td>7.651654</td>
<td>8.5</td>
<td>14263.724</td>
<td>2.089106</td>
</tr>
<tr>
<td>reached getOption(&quot;max.print&quot;) -- omitted 333 rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

merge has joined the two datasets, the large Fatality one, and the small aggregated one, on the variable year.
5.3 Reshaping data

Sometimes we have different observations of the same (or similar) variable in the same row (e.g. profit.1 and profit.2), sometimes we have them stacked in one column (e.g. as profit). We call the first format wide, the second long.

For the long case we need a variable that distinguishes the different instances of this variable (profit.1 and profit.2) from each other. In R such a variable is called timevar (Stata call them j).

We also need one or more variables that tells us, which observations actually belonged to one row in the wide format. In R we call these variables idvar (Stata call these variables i).

Let us look at a part of our trust dataset

```r
trustLong <- trustGS$subjects[,c("Date","Period","Subject","Pos",
"Group","Offer")]
trustLong[1:4,]
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Period</th>
<th>Subject</th>
<th>Pos</th>
<th>Group</th>
<th>Offer</th>
</tr>
</thead>
<tbody>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0.000</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0.000</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>0.495</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0.000</td>
</tr>
</tbody>
</table>

```r
trustWide <- reshape(trustLong,v.names=c("Offer","Subject"),
idvar=c("Date","Period","Group"),timevar="Pos",
direction="wide")
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Period</th>
<th>Group</th>
<th>Offer.2</th>
<th>Subject.2</th>
<th>Offer.1</th>
<th>Subject.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.5100000</td>
<td>13</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0.5580000</td>
<td>5</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>0.4950000</td>
<td>3</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0.8422333</td>
<td>8</td>
</tr>
</tbody>
</table>

```r
reshape(trustWide,direction="long")[1:4,]
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Period</th>
<th>Group</th>
<th>Pos</th>
<th>Offer</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>130716_0601</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

↑ Reshaping back returns more or less the orignal data. The ordering has changed and rows have got names now.

```r
library(reshape2)
recast( trustLong, Date + Period + Group ~ Pos, measure.var=c("Offer"))
```
Reshaping with **reshape2** recast does not give us Subject, though.

6 Preparing Data

- read data
- check structure (names, dimension, labels)
- check values
- create new data:
  - recode variables
  - rename variables
  - label variables
  - eliminate outliers
  - reshape data

6.1 Reading data

6.1.1 Reading z-Tree Output

The function

\[
\text{zTreeTables(...vector of filenames...[,vector of tables])}
\]

reads zTree .xls files and returns a list of tables. Here we use `list.files` to find all files that match the typical z-Tree pattern. If we ever get more experiments our command will find them and use them.

```r
library(foreign)
library(readstata13)
library("zTree")
```
```r
setwd("data/rawdata/Trust")

files <- list.files(pattern = "[0-9]{6}_[0-9]{4}.xls$", recursive=TRUE)

files

[1] "130716_0601.xls"  "130716_0602.xls"  "130716_0603.xls"  "130716_0604.xls"

trustGS <- zTreeTables(files)

reading 130716_0601.xls ...
Skipping:
Doing: globals
Doing: subjects
*** 130716_0602.xls is file 2 / 4 ***
reading 130716_0602.xls ...
Skipping:
Doing: globals
Doing: subjects
*** 130716_0603.xls is file 3 / 4 ***
reading 130716_0603.xls ...
Skipping:
Doing: globals
Doing: subjects
*** 130716_0604.xls is file 4 / 4 ***
reading 130716_0604.xls ...
Skipping:
Doing: globals
Doing: subjects

save in R-format:

save(trustGS,zTreeTables, file="160716_060x.Rdata")

save in Stata-format:

xx <- with(trustGS, merge(globals, subjects))
write.dta(xx, file="160716_060x.dta")

save in Stata-13 format:

save.dta13(xx, file="160716_060x.dta13")

save as csv:

write.csv(xx, file="160716_060x.csv")

fn <- list.files(pattern = "160716_060x\.[^.]*/")
xtable(cbind(name=fn, size=file.size(fn)))
```
As long as we need only a single table, we can access, e.g. the subjects table with `$subjects`.

If we need, e.g. the `globals` table together with the `subjects` table, we can merge:

```
with(trustGS, merge(globals, subjects))
```

### 6.1.2 Reading and writing R-Files

If we want to save one or more R objects in a file, we use `save`

```r
save(trustGS, zTreeTables, file = "data/160716_060x.Rdata")
```

To retrieve them, we use `load`

```r
load("data/160716_060x.Rdata")
```

Advantages:

- Rdata is very compact, files are small.
- All attributes are saved together with the data.
- We can save functions together with data.

### 6.1.3 Reading Stata Files

<table>
<thead>
<tr>
<th>package</th>
<th>command</th>
<th>limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreign</td>
<td><code>read.dta</code></td>
<td>Stata version 5-12</td>
</tr>
<tr>
<td></td>
<td><code>write.dta</code></td>
<td>Stata version 5-12</td>
</tr>
<tr>
<td>memisc</td>
<td><code>Stata.file</code></td>
<td>Stata version 5-12</td>
</tr>
<tr>
<td>readstata13</td>
<td><code>read.dta13</code></td>
<td>Stata version 13+</td>
</tr>
<tr>
<td>haven</td>
<td><code>read_dta</code></td>
<td>Stata version 5+</td>
</tr>
</tbody>
</table>

```
library(foreign)
sta <- read.dta("data/160716_060x.dta")
```

```
sta2 <- Stata.file("data/160716_060x.dta")
```

`foreign` (`read.dta`) stores internal Stata information as attributes of the data frame.
`memisc` (`Stata.file`) stores internal Stata information as attributes of the variable.
The data frame created by `Stata.file` looks different:

```
str(sta2)
```

Formal class 'Stata.importer' [package "memisc"] with 5 slots
..@ .Data :List of 16
  ..$ : Nmnl. item chr(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
  ..$ : Itvl. item num(0)
 ..@ .data.spec:List of 8
  ..$ names : chr [1:16] "Date" "Treatment" "Period" "NumPeriods" ...
  ..$ types : raw [1:16] 0b ff ff ff ...
Also the attributes are different:

```r
attributes(sta)

$datalabel
[1] "Written by R."

$time.stamp
[1] ""

$names
[1] "Date" "Treatment" "Period" "NumPeriods"
[5] "RepeatTreatment" "Subject" "Pos" "Group"
[9] "Offer" "Receive" "Return" "GetBack"
[13] "country" "siblings" "sex" "age"

$formats
[1] "%11s" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g"
[10] "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g" "%9.0g"

$types
[1] 138 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100

$val.labels
[1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""

$var.labels
[1] "Date" "Treatment" "Period" "NumPeriods"
[5] "RepeatTreatment" "Subject" "Pos" "Group"
[9] "Offer" "Receive" "Return" "GetBack"
[13] "country" "siblings" "sex" "age"

$row.names
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14" "15"
[16] "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29" "30"
[31] "31" "32" "33" "34" "35" "36" "37" "38" "39" "40"
[ reached getOption("max.print") -- omitted 392 entries ]

$version
```
class 
[1] "data.frame"

Stata.file stores variable labels as attributes of the variables:

attributes(sta2)

$ptr <pointer: 0x811c97180>
attr(,"file.name")
[1] "data/160716_060x.dta"

$document character(0)

$names
[1] "Date"  "Treatment"  "Period"  "NumPeriods"
[5] "RepeatTreatment" "Subject"  "Pos"    "Group"
[9] "Offer"   "Receive"   "Return" "GetBack"
[13] "country" "siblings"  "sex"    "age"

$data.spec
$data.spec$names
[1] "Date"  "Treatment"  "Period"  "NumPeriods"
[5] "RepeatTreatment" "Subject"  "Pos"    "Group"
[9] "Offer"   "Receive"   "Return" "GetBack"
[13] "country" "siblings"  "sex"    "age"

$data.spec$types
[1] 0b ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

$data.spec$nobs
[1] 432

$data.spec$nvar
[1] 16

$data.spec$varlabs

<table>
<thead>
<tr>
<th>Date</th>
<th>Treatment</th>
<th>Period</th>
<th>NumPeriods</th>
</tr>
</thead>
<tbody>
<tr>
<td>RepeatTreatment</td>
<td>Subject</td>
<td>Pos</td>
<td>Group</td>
</tr>
<tr>
<td>Offer</td>
<td>Receive</td>
<td>Return</td>
<td>GetBack</td>
</tr>
<tr>
<td>country</td>
<td>siblings</td>
<td>sex</td>
<td>age</td>
</tr>
</tbody>
</table>

$data.spec$value.labels
named character(0)
Within the `memisc` world you can obtain more information with `codebook`.

```r
codebook(sta2)
```

```text
Date 'Date'

Storage mode: character  
Measurement: nominal

  Min: 130716_0601  
  Max: 130716_0604

Treatment 'Treatment'

Storage mode: double  
Measurement: interval

  Mean: 1.000  
  Variance: 0.000  
  Skewness: NaN  
  Kurtosis: NaN  
  Min: 1.000  
  Max: 1.000

Period 'Period'

Storage mode: double  
Measurement: interval
Mean: 3.500
Variance: 2.917
Skewness: 0.000
Kurtosis: -1.269
Min: 1.000
Max: 6.000

NumPeriods 'NumPeriods'

Storage mode: double
Measurement: interval
Mean: 6.000
Variance: 0.000
Skewness: NaN
Kurtosis: NaN
Min: 6.000
Max: 6.000

RepeatTreatment 'RepeatTreatment'

Storage mode: double
Measurement: interval
Mean: 0.000
Variance: 0.000
Skewness: NaN
Kurtosis: NaN
Min: 0.000
Max: 0.000

Subject 'Subject'

Storage mode: double
Measurement: interval
Mean: 9.500
Variance: 26.917
Skewness: 0.000
Kurtosis: -1.207
Min: 1.000
Max: 18.000

Pos 'Pos'

Storage mode: double
Measurement: interval

Mean: 1.500
Variance: 0.250
Skewness: 0.000
Kurtosis: -2.000
Min: 1.000
Max: 2.000

Group 'Group'

Storage mode: double
Measurement: interval

Mean: 5.000
Variance: 6.667
Skewness: 0.000
Kurtosis: -1.230
Min: 1.000
Max: 9.000

Offer 'Offer'

Storage mode: double
Measurement: interval

Mean: 0.327
Variance: 0.137
Skewness: 0.481
Kurtosis: -1.424
Min: 0.000
Max: 1.000
Receive 'Receive'

Storage mode: double
Measurement: interval

Mean: 0.981
Variance: 1.230
Skewness: 0.481
Kurtosis: -1.424
Min: 0.000
Max: 3.000

Return 'Return'

Storage mode: double
Measurement: interval

Mean: 0.409
Variance: 0.381
Skewness: 1.502
Kurtosis: 1.437
Min: 0.000
Max: 2.763

GetBack 'GetBack'

Storage mode: double
Measurement: interval

Mean: 0.409
Variance: 0.381
Skewness: 1.502
Kurtosis: 1.437
Min: 0.000
Max: 2.763

country 'country'
siblings 'siblings'

sex 'sex'

age 'age'
The `memisc` approach preserves more information. Often this is more intuitive. Some packages are, however, confused by these attributes.

**Stata 13** Every now and then stata changes their file format:

```r
library(readstata13)
sta13<-read.dta13("data/160716_060x.dta13")
```

### 6.1.4 Reading CSV Files

CSV-Files (Comma-Separated-Value) Files are in no way always comma separated. The term is rather used to denote any table with a constant separator. Some of the parameters that always change are:

- Separators: , ; TAB
- Quoting of strings: " ’ —
- Headers: with / without

As a result, the `read.table` has many parameters.

```r
csv <- read.csv("data/160716_060x.csv",sep="\t")
str(csv)
```

The advantage of CSV as a medium to exchange data is: CSV can be read by any software. The disadvantage is: No extra information (variable labels, levels of factors, ...) can be stored.

### 6.1.5 Filesize

For our example we obtain the following sizes:

<table>
<thead>
<tr>
<th>Format</th>
<th>Size / Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlsx</td>
<td>96048</td>
</tr>
<tr>
<td>xls</td>
<td>30856</td>
</tr>
<tr>
<td>dta</td>
<td>59300</td>
</tr>
<tr>
<td>dta13</td>
<td>60354</td>
</tr>
<tr>
<td>csv</td>
<td>28178</td>
</tr>
<tr>
<td>Rdata</td>
<td>8126</td>
</tr>
</tbody>
</table>
6.2 Checking Values

load("data/160716_060x_C.Rdata")

6.2.1 Range of values

codebook(data.set(trustC))

...
6.2.2 (Joint) distribution of values

Basic plots

```r
with(trustC, hist(GetBack/Offer))
boxplot(GetBack/Offer ~ sub("_", "", Date), data=trustC, main="Boxplot")
with(trustC, plot(ecdf(GetBack/Offer)))
abline(v=1)
```

Joint distributions  First pool all data:

```r
plot(GetBack ~ Offer, data=trustC)
abline(a=0, b=3)
```
If something is suspicious (which does not seem to be the case here) plot the data for subgroups:

\[
\text{coplot}(\text{GetBack \sim Offer | Period + Date, data=trustC, show.given=FALSE})
\]

The Kakadu data contains variables `lower` and `upper`.

\[
\text{data(Kakadu)}
\]
\[
\text{nrow(Kakadu)}
\]

\[\begin{array}{c}
1827
\end{array}\]

- **lower**: lowerbound of willingness to pay, 0 if observation is left censored
- **upper**: upper bound of willingness to pay, 999 if observation is right censored

When our data falls into a small number of categories a simple scatterplot is not too informative. The right graph shows a scatterplot with some *jitter* added.
With such a large number of observations, and so few categories, a table might be more informative.

```
with(Kakadu, table(lower, upper))
```

<table>
<thead>
<tr>
<th>lower</th>
<th>2</th>
<th>5</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>129</td>
<td>147</td>
<td>156</td>
<td>176</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>63</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>69</td>
<td>0</td>
<td>0</td>
<td>321</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>76</td>
<td>0</td>
<td>281</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>61</td>
<td>187</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>152</td>
</tr>
</tbody>
</table>

### 6.2.3 (Joint) distribution of missings

- Do we expect any missings at all?
- Are missings where they should be?
  - e.g. number of siblings=0, age of oldest sibling=NA ✗
  - e.g. number of siblings=NA, age of oldest sibling=25 ✗

In our dataset we do not have the age of the oldest sibling, but let us just pretend:
The discussion of value labels in section 6.5 contains more details on missings.

### 6.2.4 Checking signatures

How can we make sure that we are working on the “correct dataset”?

Assume you and your coauthors work with what you think is the same dataset, but you get different results.

Solution: compare checksums.

```r
library(tools)
md5sum("data/160716_060x.Rdata")
```

```
data/160716_060x.Rdata
"454298e932314114b32086c371f2a282"
```

It might be worthwhile to include in the draft version of your paper the checksum of your datasets.

### 6.3 Naming variables

We already mentioned variable names in section 81.

- short but not too short

```r
lm ( otherinvestment ~ trust + ineq + sex + age + latitude + longitude)
```

```r
lm ( R100234 ~ R100412 + R100017 + R100178 + R100671 + R100229 + R100228 )
```

```r
lm ( otherinvestment ~ trustworthiness + inequalityaversion + sexOfProposer + ageOfProposer +
```

```r
lm ( oi - t + i + s + a + l1 + l2)
```

- changing existing variables creates confusion, better create new ones
• Keep related variables alphabetically together.
  ... ProfitA ProfitB ProfitC ...
  and not
  ... AProfit BProfit CProfit ...
• How do we order variable names anyway?

\[
\text{trustC[,sort(names(trustC))]}\]

6.4 Labeling (describing) variables

• Variable names should be short...
• but after a while we forget the exact meaning of a variable
  – What was the difference between Receive and GetBack?
  – Did we code male=1 and female=2 or the opposite?
• Labels provide additional information.

Either...
• use a small number of source files, and keep the information somewhere in the file
...or...
• use many source files and few data files, and keep the information with the data.

```r
load("data/160716_060x.Rdata")
trust <- within(with(trustGS,merge(globals,subjects)), {
  description(Pos)<- "(1=trustor, 2=trustee)"
  description(Offer)<- "trustor's offer"
  description(Receive)<- "amount received by trustee"
  description(Return)<- "amount trustee sends back to trustor"
  description(GetBack)<- "amount trustor receives back from trustee"
  description(country)<- "country of origin"
  description(sex)<- "participant's sex (1=male, 2=female)"
  description(siblings)<- "number of siblings"
  description(age)<- "true age"
})
codebook(data.set(trust))
attr(trust,"annotation")<="Note: 160716_0601 was a pilot,..."
annotation(trust)["note"]="Note: This is not a real dataset..."
```

• labels can be long, but they should be meaningful even if they are truncated.
  The following is not a label but a wording:
Better:

```
description(uncondSend) <- "how much would you send to the other player if no binding contract was possible"
description(condSend) <- "how much would you send to the other player if you had the possibility of a binding contract"
```

---

General attributes

| Function | Description | Required?
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>description()</code></td>
<td>short description of the variable</td>
<td>always</td>
</tr>
<tr>
<td><code>wording()</code></td>
<td>wording of a question</td>
<td>if necessary</td>
</tr>
<tr>
<td><code>annotation()</code> [&quot;...&quot;]</td>
<td>e.g. specific property of dataset</td>
<td>if necessary</td>
</tr>
<tr>
<td></td>
<td>how a variable was created</td>
<td>if necessary</td>
</tr>
</tbody>
</table>

### 6.5 Labeling values

Let us again list some interesting datatypes:

- **numbers**: 1, 2, 3
- **characters**: “male”, “female”, ...
- **factors**: “male”=1, “female”=2,...
  - factors are integers + levels, often treated as characters.
  - factors have only one type of missing (this is not a restriction, since the type of missingness could be stored in another variable)

The `memisc`-package provides another type: **item**

- **item**: “male”=1, “female”=2,...
  - items are numbers + levels, often treated as numbers.
  - items can have several types of missings. Useful for questionnaire (or from z-Tree).
trustC$sex 'participant's sex (1=male, 2=female)'

Storage mode: double
Measurement: nominal
Missing values: 98, 99

<table>
<thead>
<tr>
<th>Values and labels</th>
<th>N</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 'male'</td>
<td>174</td>
<td>44.6 40.3</td>
</tr>
<tr>
<td>2 'female'</td>
<td>216</td>
<td>55.4 50.0</td>
</tr>
<tr>
<td>98 M 'refused'</td>
<td>18</td>
<td>4.2</td>
</tr>
<tr>
<td>99 M 'missing'</td>
<td>24</td>
<td>5.6</td>
</tr>
</tbody>
</table>

\[
\text{table(as.factor(trustC$sex), useNA="always")}
\]

<table>
<thead>
<tr>
<th></th>
<th>male</th>
<th>female</th>
<th>&lt;NA&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>174</td>
<td>216</td>
<td>42</td>
</tr>
</tbody>
</table>

\[
\text{table(as.numeric(trustC$sex), useNA="always")}
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>&lt;NA&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>174</td>
<td>216</td>
<td>42</td>
</tr>
</tbody>
</table>

\[
\text{table(as.character(trustC$sex), useNA="always")}
\]

<table>
<thead>
<tr>
<th></th>
<th>female</th>
<th>male</th>
<th>missing</th>
<th>refused</th>
<th>&lt;NA&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>216</td>
<td>174</td>
<td>24</td>
<td>18</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

useNA="always" allows us to count missings. mean(is.na()) allows us to calculate the fraction of missings. The result depends on the representation.

\[
\text{mean(is.na(trustC$sex))}
\]

[1] 0

\[
\text{mean(is.na(as.factor(trustC$sex)))}
\]

[1] 0.09722222

\[
\text{mean(is.na(as.numeric(trustC$sex)))}
\]

[1] 0.09722222

\[
\text{mean(is.na(as.character(trustC$sex)))}
\]

[1] 0
How do we add labels to values? (requires memisc)

```r
trust <- within(trust,
    labels(sex)<-c("male"=1,"female"=2,"refused"=98,"missing"=99)
    labels(siblings)<-c("refused"=98,"missing"=99)
    labels(age)<-c("refused"=98,"missing"=99)
    labels(country)<-c("a"=1,"b"=2,"c"=3,"d"=4,"e"=5,"f"=6,"g"=7,"h"=8,"i"=9,"j"=10,"k"=11,
    missing.values(sex)<-c(98,99)
    missing.values(siblings)<-c(98,99)
    missing.values(age)<-c(98,99)
    missing.values(country)<-c(98,99)
})
```

6.6 Recoding data

6.6.1 Replacing meaningless values by missings

In our trust game not all players have made all decisions. z-Tree coded these “decisions” as zero. This can be misleading. Better code them as missing.

```r
trustC <- within(trust, {
    Offer [Pos==2 & Offer==0] <- NA
    GetBack[Pos==2 & GetBack==0]<-NA
    Receive[Pos==1 & Receive==0]<-NA
    Return [Pos==1 & Return==0] <- NA
})
```

```r
c( ecdfplot(~Offer, data=trust), ecdfplot(~Offer, data=trustC) )
```

Introducing missings makes a difference. The left graph shows the plot where missings were coded (wrongly) as zeroes, the right graph shows the plot with missings coded as missings.
### 6.6.2 Replacing values by other values

Sometimes we want to simplify our data. E.g. the siblings variable might be too detailed.

```r
trustC <- within(trustC, altSiblings <- recode(siblings,
  "single child" = 0 <= 0,
  "siblings" = 1 <= range(1,50),
  "refused" = 98 <= 98,
  "missing" = 99 <= 99))
```

### 6.6.3 Comparison of missings

We can not compare NAs. The following will fail in R:

```r
if (NA == NA) print("ok")
```

Error in if (NA == NA) print("ok"): missing value where TRUE/FALSE needed

```r
if (7 < NA) print("ok")
```

Error in if (7 < NA) print("ok"): missing value where TRUE/FALSE needed

(Note that the equivalent in Stata, . == . and 7 < ., do not fail but returns TRUE. )

The following works:

```r
x <- NA
if (is.na(x)) print("x is na")
```

[1] "x is na"

### 6.7 Changing variabes – creating new variables

- give them new names (overwriting “forgets” previous information)
- give them labels
- keep the old variables
6.8 Select subsets

(See the remarks on subsetting in section 5.1)

- delete records you will never ever use (in the cleaned data, not in the raw data)

```r
trust <- subset(trust, Pos != 2)
```

- generate indicator variables for records you will use in a specific context

```r
trust <- within(trust, youngSingle <- age < 25 & siblings == 0)
with(subset(trust, youngSingle), ...
```

7 Weaving and tangling

- Describe the research question
  Which model do we use to structure this question?
  Which hypotheses do we want to test?

- Describe the method

- Describe the sample
  How many observations, means, distributions of main variables, key statistics
  Is there enough variance in the independent variables to test what you want to test?

- Test hypotheses based on the model.
  possibly different variants of the model (increasing complexity)

- Discuss model, robustness checks

7.1 How can we link paper and results?

Lots of notes in the paper, e.g. the following:
In your LaTeX-file:

```latex
\% the following table was created by tableAvgProfits() from projectXYZ_160621.R
\% \begin{table}
\%
\%
\%
\% ...
```

Better: Weave (Sweave, knitr)
7.2 A history of literate programming

Donald Knuth: The CWEB System of Structured Documentation (1993)

What is "literate programming":

- meaningful and readable high-quality documentation
- details are usually not included in #comments
- supposed to be read
- facilitates feedback and reuse of code
- reduces the amount of text one must read to understand the code

Literate programming for empiricists:

- tangle (Stangle, knit(..., tangle=TRUE)): foo.Rnw → foo.R
- weave (Sweave, knit): foo.Rnw → foo.tex
  (may contain parts of foo.R)

What does Rnw mean:

- R for the R project
- nw for noweb (web for no particular language, or Norman Ramsey’s Web)
Nonliterate versus literate work

Nonliterate:

```
raw data → statistical methods → paper
         | workflow           |
```

Remember: it is easy to confuse the different version of the analysis and their relation to the versions of the paper.

Literate:

```
raw data → statistical methods → workflow → paper
```

With literatate programming in the analysis we avoid one relevant source of errors: Confusion about which parts of our work do belong together and which do not.

Advantages of literate programming

- *Connection* of methods to paper (no more: ‘which version of the methods were used for which figure, which table’)

- The paper is *dynamic*
  - More raw data arrives: the new version of the paper writes itself
  - You organise and clean the data differently: the new version of the paper writes itself
  - You change a detail of the method which has implications for the rest of the paper: the new version of the paper writes itself

Don’t write:
We ran 12 sessions with 120 participants.

Instead:
```r
numSession <- length(unique(sessionID))
numPart <- length(unique(partID))
...  
We ran \$Sexpr{numSession} sessions with \$Sexpr{numPart} participants.
```
7.3 An example

Here is a brief Rnw-document:

\documentclass{article}
\begin{document}

\begin{verbatim}
\documentclass{article}
\begin{document}

Here is a brief Rnw-document:

\documentclass{article}
\begin{document}

\begin{verbatim}
<<someCalculations,results='asis',echo=FALSE>>=
library(Ecdat)
library(xtable)
library(lattice)
data(Caschool)
attach(Caschool)
est <- lm(testscr ~ avginc)
xtable(anova(est))
\end{verbatim}

\begin{verbatim}
<<aFigure,echo=FALSE,fig.width=4,fig.height=3>>=
plot(xyplot(testscr ~ avginc,xlab="average income",ylab="testscore",
    type=c("p","r","smooth")))
\end{verbatim}

the correlation between average income and testscore is \Sexpr{round(cor(testscr,avginc),4)}
more text ...
\end{document}
\end{verbatim}

To compile this Rnw-file, we can do the following:

library(knitr)
knit("<filename.Rnw>")
system("pdflatex <filename.tex>")

...or use a front end like RStudio.
The result, after knitting:

text that explains what you are doing and why it is interesting ...

<table>
<thead>
<tr>
<th></th>
<th>DF</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avginc</td>
<td>1</td>
<td>77204.39</td>
<td>77204.39</td>
<td>430.83</td>
<td>0.0000</td>
</tr>
<tr>
<td>Residuals</td>
<td>418</td>
<td>74905.20</td>
<td>179.20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the correlation between average income and testscores is 0.7124.

more text ...

7.4 Text chunks

What we saw:

• The usual \texttt{\LaTeX}-text

• “chunks” like this

\begin{verbatim}
<<>>=
lm(testscr ~ avginc)
@
\end{verbatim}

or “chunks” with parameters:

\begin{verbatim}
<<fig.height=2.5>>=
plot(est,which=1)
@
\end{verbatim}

more generally

\begin{verbatim}
<<...parameters...>>=
...R-commands...
@
\end{verbatim}

What are these parameters:
not necessary, but identifies the chunk. Also helps recycling chunks, e.g. a figure.

- \<<...,eval=FALSE,...>>=
  this chunk will not be evaluated

- \<<...,echo=FALSE,...>>=
  the code of this chunk will not be shown

- \<<...,fig.width=3,fig.height=3,...>>=
  All figures produced in this chunk will have this width and height.

- \<<...,results='asis',...>>=
  The chunk produces \LaTeX-output which should be inserted here

Furthermore you can include small parts of output in the text:
\Sexpr{...}

**Elements of a knitr-document**

\documentclass{article}
\begin{document}
\begin{verbatim}
<<>>=
  opts_chunk["set"](dev='tikz', external=FALSE, fig.width=4.5,  
    fig.height=3, echo=TRUE, warning=TRUE,  
    error=TRUE, message=TRUE,  
    cache=TRUE, autodep=TRUE,  
    size="footnotesize")
  
  \usepackage{tikz}
\end{verbatim}
\end{document}

- `dev='tikz',external=FALSE` sets the format for plots

  Since tikz is at the moment not part of the standard R packages, you have to install with
  `install.packages("tikzDevice", repos="http://R-Forge.R-project.org")`
  This works well on Unix based systems. On a Microsoft Windows system you may need
  the Windows toolset for R which is not part of the standard distribution.

- `fig.width=4.5,fig.height=3` the the size for plots
• echo=TRUE, warning=TRUE, error=TRUE, message=TRUE what kind of output is shown
• cache=TRUE, autodep=TRUE do calculate chunks only when they have changed
• size="footnotesize" size of the output

All these values can be overridden for specific knitr chunks.

Words of caution  There is still something that might break:
In case something in R changes in the future, better put somewhere in your document:

This document has been generated on \today, with \Sexpr{version$version.string}, on \Sexpr{version$platform}.

This document has been generated on February 25, 2018, with R version 3.4.3 (2017-11-30), on amd64-portbld-freebsd11.1.

7.5 Advantages
• Accuracy (no more mistakes from copying and pasting)
• Reproducability (even years later, it is always clear how results were generated)
• Dynamic document (changes are immediately reflected everywhere, this speeds up the writing process)

7.6 Practical issues

What if some calculations take too much time  Usually you will not be able (or willing) to always do the entire journey from your raw data to the paper in one single step.

<<fastOrSlow>>=
FAST=FALSE
@

<<eval=!FAST>>=
read.csv('rawData.csv')
expensiveData<-thisTakesALongTime()
save(expensiveData,file='expensive.Rdata')
@

<<>>=
load('expensive.Rdata')
...@

Alternatively: caching intermediate results  knitr can also cache intermediate results:

```r
<<expensiveStep,cache=TRUE>>=
intermediateResults <- ....
```

The above chunk is executed only once (unless it changes), results are stored on disk and can be used lateron.

### 7.7 When R produces tables

#### 7.7.1 Tables

You can save a lot of work if you harness R to create and format your tables for you. A versatile function is `xtable`:

```r
(x <- rbind(c(1,2,3),c(4,5,6)))
```

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

```r
<<results='asis'>>>
library(xtable)
xtable(x)
```

```

\begin{tabular}{ccc}
  & 1 & 2 & 3 \\
\hline
1 & 1.00 & 2.00 & 3.00 \\
2 & 4.00 & 5.00 & 6.00 \\
\end{tabular}
```

#### 7.7.2 Estimation results

Estimation results in tabular form from `mtable` are typeset by `toLatex`:

```r
library(Ecdat)
data(Caschool)
est1 <- lm(testscr ~ str,data=Caschool)
est2 <- lm(testscr ~ str + elpct,data=Caschool)
est3 <- lm(testscr ~ str + elpct + avginc,data=Caschool)
toLatex(mtable(est1,est2,est3))
```
<table>
<thead>
<tr>
<th></th>
<th>est1</th>
<th>est2</th>
<th>est3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>698.933***</td>
<td>686.032***</td>
<td>640.315***</td>
</tr>
<tr>
<td></td>
<td>(9.467)</td>
<td>(7.411)</td>
<td>(5.775)</td>
</tr>
<tr>
<td>str</td>
<td>−2.280***</td>
<td>−1.101**</td>
<td>−0.069**</td>
</tr>
<tr>
<td></td>
<td>(0.480)</td>
<td>(0.380)</td>
<td>(0.277)</td>
</tr>
<tr>
<td>elpct</td>
<td>−0.650***</td>
<td>−0.488****</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.039)</td>
<td>(0.029)</td>
<td></td>
</tr>
<tr>
<td>avginc</td>
<td>1.495***</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.075)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-squared</td>
<td>0.051</td>
<td>0.426</td>
<td>0.707</td>
</tr>
<tr>
<td>adj. R-squared</td>
<td>0.049</td>
<td>0.424</td>
<td>0.705</td>
</tr>
<tr>
<td>sigma</td>
<td>18.581</td>
<td>14.464</td>
<td>10.347</td>
</tr>
<tr>
<td>F</td>
<td>22.575</td>
<td>155.014</td>
<td>334.889</td>
</tr>
<tr>
<td>p</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Log-likelihood</td>
<td>−1822.250</td>
<td>−1716.561</td>
<td>−1575.374</td>
</tr>
<tr>
<td>Deviance</td>
<td>144315.484</td>
<td>87245.293</td>
<td>44540.732</td>
</tr>
<tr>
<td>AIC</td>
<td>3650.499</td>
<td>3441.123</td>
<td>3160.748</td>
</tr>
<tr>
<td>BIC</td>
<td>3662.620</td>
<td>3457.284</td>
<td>3180.950</td>
</tr>
<tr>
<td>N</td>
<td>420</td>
<td>420</td>
<td>420</td>
</tr>
</tbody>
</table>

**Nicer names for variables and equations**

```r
toLatex(relabel(mtable("simple"=est1,"intermediate"=est2,
"final"=est3),c(str="student/teacher",
elpct="English learners",avginc="average income",
"(Intercept)"="Constant")))
```
<table>
<thead>
<tr>
<th></th>
<th>simple</th>
<th>intermediate</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>698.933***</td>
<td>686.032***</td>
<td>640.315***</td>
</tr>
<tr>
<td></td>
<td>(9.467)</td>
<td>(7.411)</td>
<td>(5.775)</td>
</tr>
<tr>
<td>str</td>
<td>−2.280***</td>
<td>−1.101**</td>
<td>−0.069</td>
</tr>
<tr>
<td></td>
<td>(0.480)</td>
<td>(0.380)</td>
<td>(0.277)</td>
</tr>
<tr>
<td>elpct</td>
<td>−0.650***</td>
<td>−0.488***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.039)</td>
<td>(0.029)</td>
<td></td>
</tr>
<tr>
<td>avginc</td>
<td></td>
<td></td>
<td>1.495***</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.075)</td>
</tr>
<tr>
<td>R-squared</td>
<td>0.051</td>
<td>0.426</td>
<td>0.707</td>
</tr>
<tr>
<td>adj. R-squared</td>
<td>0.049</td>
<td>0.424</td>
<td>0.705</td>
</tr>
<tr>
<td>sigma</td>
<td>18.581</td>
<td>14.464</td>
<td>10.347</td>
</tr>
<tr>
<td>F</td>
<td>22.575</td>
<td>155.014</td>
<td>334.889</td>
</tr>
<tr>
<td>p</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Log-likelihood</td>
<td>−1822.250</td>
<td>−1716.561</td>
<td>−1575.374</td>
</tr>
<tr>
<td>Deviance</td>
<td>144315.484</td>
<td>87245.293</td>
<td>44540.732</td>
</tr>
<tr>
<td>AIC</td>
<td>3650.499</td>
<td>3441.123</td>
<td>3160.748</td>
</tr>
<tr>
<td>BIC</td>
<td>3662.620</td>
<td>3457.284</td>
<td>3180.950</td>
</tr>
<tr>
<td>N</td>
<td>420</td>
<td>420</td>
<td>420</td>
</tr>
</tbody>
</table>

**Requirements**  
The default of `toLatex` assumes the `dcolumn` package, i.e. in the preamble of the document we have to say something like:

```latex
\usepackage{dcolumn}
\let\toprule\hline
\let\midrule\hline
\let\bottomrule\hline
```

### 7.7.3 Mixed effects

If we use `lmer` to estimate models with mixed effects, `toLatex` needs a `summary.mer` method. The following is one example:

```r
bootSize <- 1000
getSummary.mer <- function(mer) {
  msd <- sqrt(diag(vcov(mer)))
  coefs <- fixef(mer)
  mzm <- mcmcsamp(mer, bootSize)
  mf <- mzm$fixef
  mz <- 2*pnorm(-abs(mzt <- (mzcoef <- apply(mf, 1, mean))/
                     (mzsd <- apply(mf, 1, sd))))
  mzc <- cbind(coefs) + cbind(mzsd) %*% qnorm(c(.025, .975))
  coef <- cbind(coefs, mzsd, mzt, mzp, mzc)
  colnames(coef) <- c("est", "se", "stat", "p", "lwr", "upr")
  smer <- summary(mer)
  AIC <- smer@AICtab$AIC
  return(coef)
}
```
BIC <- smer@AICtab$BIC
logLik <- smer@AICtab$logLik
deviance <- smer@AICtab$deviance
REMSdev <- smer@AICtab$REMSdev
N <- length(mer@resid)

# below we assume two random effects: one for the independent
# observations and one for the participants
# this is frequently the case for experiments but need not
# always be the case for other mer-s
ngrps <- min(smer@ngrps)
mgrps <- max(smer@ngrps)
sumstat <- c(deviance = deviance, AIC = AIC, BIC = BIC,
             logLik = logLik, N = N, ngrps = ngrps, mgrps = mgrps)
list(coef = coef, sumstat = sumstat, call = mer@call)
}

setSummaryTemplate(mer = c("Log-likelihood" = "($logLik:f#)",
                           Deviance = "($deviance:f#)",
                           AIC = "($AIC:f#)",
                           BIC = "($BIC:f#)",
                           N = "($N:d)",
                           "indep.obs." = "($ngrps:d)",
                           "participants" = "($mgrps:d)"))

setCoefTemplate(pci = c(est = "($est:#)($p:*)",
                         ci = "[$(lwr:#);$(upr:#)]"))

We should note that our definition of indep.obs. and participants as the smallest and largest number of groups, respectively, is often reasonable if we have, indeed, two random effects, one for independent observations, the other for participants. This is frequently the case for experiments but need not always be the case for other mixed effects models.

We should also note that there are several ways to bootstrap p-values. In the example we use mcmcsamp and we assume that the distribution of coefficients follows a normal distribution.

7.8 The magic of make

In the same directory where I have my Rnw file, I also have a file that is called Makefile. Let us assume that the current version of my Rnw file is called myProject_160601.Rnw. Then here is my Makefile:

```bash
PROJECT = myProject_160601

pdf: $(PROJECT).pdf
%.pdf: %.tex
   pdflatex $<

%.tex: %.Rnw
   echo "library(knitr);knit("$<");" | R --vanilla
```
Let us go through the individual lines of this Makefile.

```makefile
PROJECT = myProject_160601
```

Here we define a variable. This is useful, since this most of the time the only line of the Makefile I ever have to change (instead of changing every occurrence of the filename)

```makefile
default: pdf
```

The part pdf before the colon is a target. Since it is the first target in the file it is also the default target. I.e. make will try to make it whenever I just say

```
make
```

Make will do the same when I call it explicitly

```
make pdf
```

The part after the colon tells make on which file(s) the target actually depends (the prerequisites). Here it is only one but there could be several. If all prerequisites exist, and if they are up-to-date (newer than all files they depends on), make will apply the rule. Otherwise, make will try to create the prerequisites (the pdf file in this case, with the help of other rules) and then apply this rule.

```makefile
%.tex: %.Rnw
    echo "library(knitr);knit("$<");" | R --vanilla
```

This is a rule that make can use to create tex files. So above we requested the pdf file myProject_160601.pdf, and now make knows that we require a file myProject_160601.tex. If this already exists and is up-to-date (i.e. newer than all files it depends on), make will apply this rule. Otherwise, make will first try to create the prerequisite (the single tex file in this case would be created with the help of other rules) and then apply this rule.

To create our pdf it is now sufficient to say (from the command line, not from R)

```makefile
make
```

and make will do everything that is needed.

Note: In this context a simple shell script would work almost as well. However, make is very helpful when your pdf file depends on more than one tex or Rnw file.
A Makefile for a larger project  When I wrote this handout I split it into several Rnw files. This saves time. When I make changes to one part, only this part has to be compiled again. The files were all in the same directory. The directory also contained a “master”-tex file that would assemble the tex-files for each Rnw-file.

The following example shows how we assemble the output of several files to make one document:

```makefile
PROJECT = myProject_160601
RPARTS   = $(wildcard $(PROJECT)_[1-9].Rnw)
TEXPARTS = $(RPARTS:.Rnw=.tex)
pdf: $(PROJECT).pdf

# our project depends on several files:
$(PROJECT).pdf: $(TEXPARTS) $(PROJECT).tex
    pdflatex $(PROJECT)

# only the tex files who belong to Rnw files
# should be knitted:
$(TEXPARTS) : %.tex : %.Rnw ;
    echo "library(knitr);knit("$<");" | R --vanilla
```

8 Version control

8.1 Problem I – concurrent edits

What happens if two authors, Anna and Bob, simultaneously want to work on the same file. Chances are that one is deleting the changes of the other. (This problem is similar to one author working on two different machines)

- Anna’s work is lost — very inefficient (50% of the contribution is lost)

8.2 A “simple” solution: locking

Serialising the workflow might help. Anna could put a “lock” on a file while she wants to edit this file. Only when she is finished, the “unlocks” the file and Bob can continue.
Bob can only work with Anna’s permission — very inefficient (50% of the time Anna and Bob are forced to wait)

8.3 Problem II – nonlinear work

Even when Anna works on a problem on her own she can be in conflict with herself. Imagine the following: Anna successfully completed the steps A, B, and C on a paper and has now something readable that she could send around. Perhaps she actually has sent it around. Now she continues to work on some technical details D and E, but so far her work in incomplete – D and E are not ready for the public. Suddenly the need arises to go back to the last public version (C) and to add some work there (e.g. Anna decides to submit the paper to a conference, but wants to rewrite the introduction and the conclusion. It will take too much time to first finish the work on D and E, so she has to go back to C. Rewriting the introduction and conclusion are steps F and G. Once the paper (G) has been submitted, Anna wants to return to the technical bits D and E and merge them with F and G.

8.4 Version control

(revision control, source control) Traditional:

- Editions of a book
- Revisions of a specification

Software:

- Concurrent Versions System (CVS)
- Subversion (SVN)
- Git
- Mercurial
In this course we will use Git.

- Free
- Distributed repository
- Supports many platforms, formats

8.5 Solution to problem II: nonlinear work

Before we create our first git-repository, we have to provide some basic information about ourselves:

```bash
$ git config --global user.name "Your Name Comes Here"
$ git config --global user.email you@yourdomain.example.com
```

Now we can create our first repository:

```bash
$ git init
```

We can check the current "status" as follows:

```bash
$ git status
```

now we create a file test.Rnw

```bash
$ git add test.Rnw
```

git status

# On branch master
# Initial commit
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
# new file: test.Rnw

| git commit -a -m "first version of test.Rnw"

# On branch master
nothing to commit, working directory clean

| git log --oneline
| git log --oneline

3ea6194 first version of test.Rnw

Note that git denotes versions with identifiers like "3ea6194" (and not A, B, C).
After some changes to test.Rnw...

| git status

# On branch master
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
# modified: test.Rnw
# no changes added to commit (use "git add" and/or "git commit -a")

| git commit -a -m "introduction and first results"

# On branch master
nothing to commit, working directory clean

| git log --oneline

74fd521 introduction and first results
3ea6194 first version of test.Rnw

More changes and...

| git commit -a -m "draft conclusion"
more changes and...

| git commit -a -m "improved regression results (do not fully work)"
more changes and...

| git commit -a -m "added funny model (does not fully work yet)"
Assume we want to go back to 1d05e8f but not forget what we did between 1d05e8f and f965066.

Remember current state:

```bash
git branch funny
```

Now that we have given the current branch a name we can revert to the old state:

```bash
git reset 1d05e8f
```

Unstaged changes after reset:
```
M test.Rnw
```

```bash
git checkout test.Rnw
```

Nothing to commit, working directory clean.

```bash
git status
```

Do more work...

```bash
git commit -a -m "rewrote introduction"
```

Do even more work...

```bash
git commit -a -m "rewrote conclusion, added literature"
```
eventually we want to join the two branches:

```bash
    git merge funny
```

now two things can happen: Either this...

```
Merge made by recursive.
    test.Rnw | 1 +
    1 files changed, 1 insertions(+), 0 deletions(-)
```

or that...

```
Auto-merging test.Rnw
CONFLICT (content): Merge conflict in test.Rnw
Automatic merge failed; fix conflicts and then commit the result
```

We can fix this with `git mergetool`:

```
    git mergetool
```

Now we can make detailed merge decisions in an editor.

```
    git commit -m "merged funny"
```

---

### 8.6 Solution to problem I: concurrent edits

Version control allows all authors to work on the file(s) simultaneously.

In this example we start with an empty repository. In a first step both Anna and Bob
“checkout” the repository, i.e. they create a local copy of the repository on their computer.

Anna creates a file, adds it to version control and commits it to the repository. Bob then
updates his copy and, thus, obtains Anna’s changes.

- First step: create a “bare” repository on a “server”
This repository can now be accessed from “clients”, either on the same machine...

...or on a different machine via ssh (where user has access rights):

<table>
<thead>
<tr>
<th>Anna</th>
<th>Repository</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>git clone ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>creates a file test.Rnw:</td>
</tr>
<tr>
<td>A=...</td>
</tr>
<tr>
<td>B=...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>git add test.Rnw</th>
</tr>
</thead>
<tbody>
<tr>
<td>git commit</td>
</tr>
<tr>
<td>uploads the file:</td>
</tr>
<tr>
<td>git push</td>
</tr>
</tbody>
</table>

| A=... |
| B=... |
| A=... |
| B=... |

8.7 Edits without conflicts:

To make this more interesting we now assume that both work on the file. Anna works on the upper part (A), Bob works on the lower part (B). Both update and commit their changes. Since they both edit different parts of the file, the version control system can silently merge
their changes.

Anna | Repository | Bob
--- | --- | ---
A=1 | A=... | A=...
B=... | B=... | B=2

Both commit their work to their own local repos:

- `git commit -a -m "..."`

Anna pulls, but there is no conflict

- `git pull`

Anna pushes her changes

- `git push`

Bob pulls, and finds a merge conflict

- `git pull`
- `git mergetool`

Bob commits his merge

- `git commit -a -m "..."`

Bob pushes his merge

- `git push`

Anna pulls to get the current version

- `git pull`

### 8.8 Going back in time

Version control is not only helpful to avoid conflicts between several people, it also helps when we change our mind and want to have a look into the past. `git log` provides a list of the different revision of a file:

- `git log --oneline`

f965066 added funny model (does not fully work yet)
9100277 improved regression results (do not fully work)
git blame allows you to inspect modifications in specific files. If we want to find out who introduced or removed “something specific” (and when), we would say...

\texttt{git blame -L '/something specific/' test.Rnw}

There is a range of GUIs that allow you to browse the commit tree. Try, e.g., gitk

8.9 git and subversion

- git-Server: requires ssh access to the server machine

- subversion-Server: provided by the URZ at the FSU Jena

\texttt{git} can use subversion as a remote repository:

\begin{align*}
\text{\texttt{git clone}} & & \text{\texttt{git svn clone}} \\
\text{\texttt{git pull}} & & \text{\texttt{git svn rebase}} \\
\text{\texttt{git push}} & & \text{\texttt{git svn dcommit}}
\end{align*}

- Conceptual differences:

  - subversion has only one repository (on the server), git has one or more local repositories plus one or more on different servers.
  
  - inconsistent uploads to a server:

    subversion will not complain if after a push/commit the state on the server is different from the state on any of the clients. git will not allow this (git forces you to pull first, merge, commit, and push then)

8.10 Steps to set up a subversion repository at the URZ at the FSU Jena

If you need to set up a subversion repository here at the FSU, tell me about it and tell me the \texttt{\langle urz-login\rangle}s of the people who plan to use it. Technically, setting up a new repository means the following:

- ssh to subversion.rz.uni-jena.de

- \texttt{svnadmin create /data/svn/\langle repository\rangle}

- \texttt{chmod -R g+w /data/svn/\langle repository\rangle}
• set access rights for all involved \( \langle urz-login \rangle \)s in /svn/access-ewf

• then, at the local machine in a directory that actually contains only the files you want to add: 
  \texttt{svn --username \langle urz-login \rangle import \url{https://subversion.rz.uni-jena.de/svn/ewf/\langle repository \rangle} -m "Initial import"}

  (this “imports” data into the repository)

• then, at all client machines, 
  \texttt{svn --username \langle urz-login \rangle checkout \url{https://subversion.rz.uni-jena.de/svn/ewf/\langle repository \rangle}}

8.11 Setting up a subversion repository on your own computer

• On your own computer: 
  \texttt{svnadmin create \langle path \rangle/\langle repository \rangle}

  (\langle path \rangle is a complete path, e.g. /home/user/Documents/ or /C:MyDocuments/)

• then, in a directory that actually contains only the files you want to add:
  \texttt{svn import . file://\langle path \rangle/\langle repository \rangle -m "Initial import"}

• then, wherever you actually want to work on your own computer:
  \texttt{svn checkout file://\langle path \rangle/\langle repository \rangle}

• if you have ssh access to your computer you can also say from other machines:
  \texttt{svn checkout svn+ssh://\langle yourComputer \rangle/\langle path \rangle/\langle repository \rangle}

8.12 Usual workflow with git

While setting up a repository looks a bit complicated, using it is quite simple:

• \texttt{git pull} check whether the others did something

• editing
  - \texttt{git add} add a file to version control
  - \texttt{git mv} move a file under version control
  - \texttt{git rm} delete a file under version control

• \texttt{git commit} commit own work to local repository

• \texttt{git pull} check whether the others did something

• \texttt{git mergetool} merge their changes

• \texttt{git commit} commit merge

• \texttt{git push} upload everything to the server
8.13 Exercise

Create (in \langle path \rangle) four directories A, B, C.

From A create a repository: \texttt{svnadmin create ../R}

In A create a file \texttt{test.txt} with some text: A=… B=…

Initial import. In A say:

\begin{verbatim}
svn import . file://\langle path \rangle/R -m "My first initial import"
\end{verbatim}

In B:

\begin{verbatim}
svn checkout file://\langle path \rangle/R
\end{verbatim}

In C:

\begin{verbatim}
svn checkout file://\langle path \rangle/R
\end{verbatim}

Simultaneous changes to \texttt{test.txt}

\begin{verbatim}
A=1 B=...
A=... B=2
\end{verbatim}

Commit changes

\begin{verbatim}
svn commit
svn commit
\end{verbatim}

Update

\begin{verbatim}
svn update
svn update
\end{verbatim}

9 Exercises

Exercise 1
Have a look at the dataset \texttt{Workinghours} from the library \texttt{Ecdat}. Compare the distribution of “other household income” for whites and non-whites. Do the same for the different types of occupation of the husband.

Exercise 2
Read the data from a hypothetical experiment from \texttt{rawdata/Coordination}. Does the Effort change over time?

Exercise 3-a
Read the data from a hypothetical z-Tree experiment from \texttt{rawdata/Trust}. Do you find any relation between the number of siblings and trust?

Exercise 3-b
For the same dataset: Attach a label (description) to \texttt{siblings}. Attach value labels to this variable.

Exercise 3-c
Make the above a function.
Also write a function that compares the offers of all participants with \( n \) siblings with the other offers. This function should (at least) return a p-value of a two-sample Wilcoxon test (\texttt{wilcox.test}). The number \( n \) should be a parameter of the function.

**Exercise 4**

Read the data from a hypothetical z-Tree experiment from \texttt{rawdata/PublicGood}. The three variables \texttt{Contrib1}, \texttt{Contrib2}, and \texttt{Contrib3} are contributions of the participants to the other three players in their group (in groups of four).

1. Check that, indeed, in each period, players are equally distributed into four groups.
2. Produce for each period a boxplot with the contribution (i.e. 16 boxplots in one graph).
3. Add a regression line to the graph.
4. Produce for each contribution partner a boxplot with the contribution (i.e. 3 boxplots in one graph).
5. Produce an \texttt{Sweave} file that generates the two graphs. In this file also write when you estimate the average contribution reaches zero.