Statistical Computing

Master Data Science

Winter Semester 2017/18 **Prof. Tim Downie**

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Workshop 1
Basic statistics and exploratory data analysis I

1 Introduction

In this workshop we will learn how to do some basic data exploration in R, e.g. how to calculate summary statistics. We will also learn something about how to read and save data in R. This subject continues next week in *Basic statistics and exploratory data analysis II* with data manipulation and more analysis methods

Some of the topics today might well seem old fashioned for a Master's student in Data Science. It is important to cover the small stuff first; it will be easier to move on to the more detailed topics, helps to give gives a historical context and makes sure that all students are at the same level.

Preliminaries

Start RStudio using the Icon on the desktop

$\boxed{\text{Strg+Shift+N}}$ opens a new R script.

Type the comment

```
#Statistical Computing: Workshop 3
#Basic Statistics and exploratory data I
```

as the title.

Save the file in your `H:\\StatComp` folder with the name `Workshop3.R` using *File > Save as*.

Set your *working directory* to be `H:\\StatComp`. The code to do this is

```
> setwd(H://StatComp)
```

By now it is possible that you have accidentally saved your workspace when leaving RStudio. If your Environment Window is displaying any objects then clear the workspace using *Session > clear workspace*.

Go back to the course Moodle page. Download the data files: `Medication.csv`, `Prestige.csv` and `BloodType.csv`. To do this right click and choose *Ziel speichern unter ...* and save the file in your `H:\\StatComp` directory.

Work through the examples as you did last week. In many of the code examples the output is not included, you should always read and understand the output. Please add comments (in your own words) to your code so that it will make sense to you when you return several weeks (or years) later. When there is a question in the text e.g. *"How many students are included in the data set?"* , answer the question in the script file as a comment directly below the command which produces the answer.

2 Reading and storing data

So far in this course, we've made extensive use of the 'built-in' datasets that R provides, using the data command to access them. In practise of course, you need to be able to enter your own data into R. There was a brief introduction to this in the *Further Reading sheet*, where we saw three commands for getting your data into R: `read.table`, `scan` etc. More on these below, but first we need to understand something about electronic file formats.

2.1 Storing data electronically

You'll be familiar with the notion of a *file* as a collection of information held on a computer. When you do some work in a software package (Microsoft Word for example) and subsequently save it, the package writes the required information to a file. Many packages write information using their own unique coding system. We will call such a coding system a *file format*. For example, if you write a letter in a package such as Word and then save it, it writes a file containing all sorts of information — the fonts you used, the justification of each of your paragraphs, the style of page numbering and so on. The advantage of this is that, the next time you open the file, it should look exactly the same as it did when you saved it. The disadvantage is that the document is not *portable* i.e. only Word, along with a limited number of other packages that can interpret Word files, can read it.

Portability is an important consideration when it comes to the preparation of electronic data files for a statistical analysis. It is good to know that if you prepare a data file for someone to use, they will be able to read it even if they don't have the same software packages or operating system as you do. This mode of thinking is conspicuously absent from a lot of popular modern software ... however, there is an internationally agreed standard file format for the transmission of electronic data between packages and systems. This is called ASCII (American Standard Code for Information Interchange), and is a format we'll often use for data files in this course. If you've been wedded to Windows all your life, you've probably encountered ASCII files although you may not know it Microsoft products refer to them as 'text files'. So: how do you create an ASCII file? Well, one way is to type your data into a package such as Word, and then save the file in the appropriate format. However, a better way is to use a package that is specifically designed for the creation and editing of ASCII files (this way, you know that it's not going to do anything clever, unexpected and wrong without telling you!). Such a package is called a text editor.

Exercise: Let's start by creating a small data file. The (fictitious) data below represent measurements on 4 patients in a clinical trial; two of the patients have been treated using an experimental product and the others with a placebo:

```
Sex Treatment Response
F   0         15.3
F   1         14.2
M   0         16.1
M   1         14.9
```

Use a Text Editor to create manually an ASCII file containing these data, the exact program will depend on your operating system.

Windows The default Text Editor is called *Notepad* in English or *Editor* in German. This program is not great; for example it can only deal with one encoding of the "newline" character and one often finds upon opening a text file, that all the text is on one long line. I download the freeware *notepad++* (Website: https://notepad-plus-plus.org/) on my personal computers, which is a much more flexible program for manipulating ASCII text. If you are using your own Windows laptop you might want to install this software.

Mac If you are using a Mac use the file TextEdit. When you save the file convert the file to ASCII text via *Format > make plain text* prior to saving it. If you are using

Linux You probably already use a text editor. Popular Linux editors are *Vim*, *emacs* and *gedit*.

Enter the data exactly as shown above (including the first row with the column headings). Save the file in your `StatComp` directory (or wherever it is that you keep your files for this course). I suggest you call the file `Wkshp3test.dat`. By applying the suffix `.dat` to the file name, you'll recognise this as an ASCII data file later on. It's good to be methodical about naming files.

Each row has three data values which for the specification of a data file is called a field. So the first line after the "header" has three fields `F`, `0` and `15.3`.

2.2 R commands for reading data

Data can be read directly from an ASCII file into R using one of the following commands:

`read.table` This is suitable when, as in the exercise above, your data fields are separated by spaces. The columns don't have have to be aligned exactly, so long as there are the same number of entries on each row of the data file. Its effect is to read the data into a data frame; optionally, the columns of this data frame can be named using information in the first row of the file (called the "header"). So for example, the command

```
> test.data <- read.table("Wkshp3test.dat",header=TRUE)
```

will read the data from the file `Wkshp3test.dat` that we created above, and store it in a data frame called `test.data`. The argument `header=TRUE` tells R that the first row of the file contains column names rather than data. Unless instructed otherwise, `read.table` tries to make intelligent decisions about the nature of the data that it reads.

The basic rule is: columns that contain only numbers are interpreted as numeric variables, and columns that contain characters are interpreted as factors. This interpretation affects the way in which the data are treated in statistical analyses. By and large, it is sensible, although you can get unexpected results if your factors are coded numerically. For example, in `Wkshp3test.dat` the Treatment variable, which should strictly be regarded as a factor, is coded as 0 or 1 and hence will be analysed as though it's numeric. The default interpretation can be changed using additional arguments to the command see the help system for details.

`read.csv` A specific file format for datasets is the *CSV* Format. CSV files are still ASCII text files but have specific properties. CSV means *comma separated values*. So each field on a line is separated by a comma. Text fields especially text including a space or a comma need to be quoted with a `"` quote.

Excel is good for data entry and record keeping, such as adding up marks when correcting exam papers. Excel is not good as a statistical tool. In particular, it is terrible for data science on large data sets.[1] However it is quite common to enter date in Excel and then import it into R. The commands `read.csv` and `read.csv2` make this process very easy. If you have data in an Excel worksheet and want to import it into R, then in Excel *export* the worksheet using *.csv* format.

`read.csv2` In Germany and many other European countries the comma is used as the decimal mark. This means that decimal numbers would be read in as two values if standard comma separation is expected. The work around is to use a semi-colon `;` as the field separator and a comma `,` as the decimal mark. Technically the file is no longer "comma separated"; I have occasionally seen the expression "character separated variables" to avoid the inconsistency. If your data is in this format use

```
> somedata <- read.csv2("somedata.dat")
```

Both `read.csv` and `read.csv2` are meant as short cut functions. Both functions end up calling `read.table` but are set up with the appropriate default arguments so that the user does not need to type them. E.g. both the following commands do the same thing.

```
> medication<-read.csv(file="Medication.csv")
> medication<-read.table(file="Medication.csv",header=TRUE,sep=",")
```

`scan` This is more flexible than `read.table`, and correspondingly more difficult to use. It also allows you to input data directly from the keyboard (although this particular feature isn't that useful!). As an alternative to the `read.table` command above, we could have used:

```
> medication <- as.data.frame(scan("Medication.csv",
+ what=list(Medikament="",Zeit=1),skip=1))
```

You may prefer `read.table`!

One situation where `scan` is useful, is for importing the data as it is, and performing all the data processing in R. An example is scraping or stripping websites in HTML format.

[1] I heard recently of a problem at a large German insurance company, where one particular data analysis was implemented using Excel. The data analyst had not realised that the maximum data size had been exceeded. Rather than the data being limited to just n rows and m columns the data was overwritten with no warning message. Luckily a colleague spotted the problem before the project was completed.

Use `scan` to read in each line of text as a character object and the *parse* html code to extract the useful data. We will cover this later in the course.

`read.fwf` This is for reading fixed-width ASCII files i.e. files in which the information is stored in columns that are always the same width. You might think that `read.table` can do this. However, consider what happens if somebody sends you a file of daily rainfall values that looks like this:

```
19831129G119.39
19831129G2 3.33
19831130G1 7.83
198312 1G1 3.56
198312 1G2 0.39
198312 2G2 9.30
```

These data represent daily rainfalls (in mm) from 2 sites. Each record is in the format `YYYYMMDDSS##.##` (year, month, day, site code and rainfall amount).[2] `read.table` wouldn't work here because the columns aren't separated by spaces (or by anything else). However, we can use that fact that 'Year' occupies positions 1-4 of each line, 'Month' occupies positions 5-6, and so on, to extract the data unambiguously. The function `read.fwf` is designed for use in such situations.

`load` If the data file has been saved in R format (see next section) then it is easy to load the data. Just type `load(file="filepath/filename.Rda")`. The usual file name suffix is either `.Rda` or `.Rdata`. The name of the imported R object has the same name as the when it was saved. You can see the new object name in the Environment Window.

As usual, these commands have many options. If you want to know what they are, look in the help system!

Exercise (continued): What happens if you omit `header=TRUE` from the first `read.table` command?

2.3 Databases

You might well be thinking: *We are leaning in other courses about structured Databases. Is it possible to load data from such a Database into R?*

The answer is yes. However the more complicated the data source is the more specific we have to be when reading in the data. It is possible for R to send e.g. SQL commands to a database, which process the data at source and the send the data to R. You should learn the basics first; you will learn about the R/Database interface in the course machine learning.

[2]The rationale for providing datasets in such a format is that each character in a file (including spaces) requires some space on disk. For large datasets, it can be extremely wasteful of disk space to store superfluous spaces between data values.

2.4 Saving R Data

You have imported data from a text file and maybe you have restructured it somehow. Now you want to save the data for another session (a good idea). There are two main options available to output in ASCII format or in R format. R format is much easier. You should use ASCII format if we hand the data onto another person if you are not sure that they are an R user (remember portability!).

Save in R format

The Command to use is
```
> save(test.data,file="filepath/filename.Rda")
```
You can save multiple objects in one file
```
> save(test.data,medication,file="filepath/filename.Rda")
```
If you want to save your whole Environment you could use the following command **but please don't try it out here!**
```
> save.image(file="filepath/filename.Rdata")
```
I don't recommend this as you usually have quite a few small unnecessary objects that just get in the way.

An advantage of saving in R format is that all the fiddly things such as row names, variable names, factor levels and class, are all preserved and do not need to be specified when reloading the data. R object saved in this way are read in using `load`.

Writing data in ASCII format

The commands `write.table`, `write.csv` and `write.csv2` are analogous to the `read.table`, `read.csv`, and `read.csv2` functions. E.g.
```
> write.csv2(medication,file="test.csv",row.names=FALSE)
```
If you need to write the data in ASCII format I recommend using `write.csv2` if you normally use a computer with German settings and `write.csv` if you normally use a computer with English settings.

3 Basic summary statistics

When analysing data you will usually start by carrying out some simple procedures to get a feel for the data. Typically, we will produce some appropriate plots and calculate some useful summary statistics. You learned a variety of graphical techniques in Workshop 2; now we will examine commands for calculating summary statistics in R.

In this section we will be using the Canadian `Prestige` data set and contains the following variables:

`occupation` 102 occupation groups as defined by the Canadian Census Bureau in 1971.

`education` Average education of occupational group, years, in 1971.

`income` Average income of group, dollars, in 1971.

`women` Percentage of group who are women.

`prestige` "Pineo-Porter prestige score" for occupation, from a social survey conducted in the mid-1960s.

`census` Canadian Census occupational code.

`type` Type of occupation. A factor with levels: `bc`, Blue Collar; `prof`, Professional, Managerial, and Technical; `wc`, White Collar.

The Data is in (German) CSV format. Read in the data set using

```
> Prestige<-read.csv2(file="prestige.csv")
```

3.1 Summarising numeric variables

First of all we should find out how many rows and columns our data set has

```
> dim(Prestige)
```

The basic commands to summarise the numeric variables in the data set are straightforward, providing you know the expressions in English!

`mean()`	arithmetic mean (arithmetisches Mittel)
`var()`	variance[3]
`sd()`	standard deviation (Standardabweichung)
`median()`	median

```
> mean(Prestige$income)
> median(Prestige$income)
> sd(Prestige$income)
> var(Prestige$income)
> sqrt(var(Prestige$income))
```

The units of the mean, median and standard deviation are the same as the measured variable e.g. Canadian dollars. The units of the variance are squared units e.g. dollars^2. In terms of understanding the numbers, the standard deviation is much more helpful than the variance.

These functions all have arguments that control the treatment of missing values. These arguments are there for convenience and readability they're not really necessary. For example, `mean(x, na.rm=TRUE)` is just a convenient way of expressing `mean(x[!is.na(x)])`.

Exercise: Try this yourself. Define a vector `x` with the following definition

```
> x<-c(1:5,NA,10:15)
```

investigate the latter command by inspecting the following

```
> is.na(x)
> !is.na(x)
> x[!is.na(x)]
```

What does the operator `!` do to a logical vector? Verify that both the following commands give the same result.

```
> mean(x, na.rm=TRUE)
> mean(x[!is.na(x)])
```

We may also want to compute the quantiles of a set of numeric data. The command is `quantile`. For example, to obtain the *lower* and *upper quartiles* of income:

```
> quantile(Prestige$income,probs=c(0.25,0.75))
```

If you read about another type of *summary statistic*, there is probably a function in R or in a package to do calculate it.

Exercise:

(a) What is the mean and standard deviation of the prestige Scores?

(b) What is the median of the census codes? Does this statistic even make sense?

3.2 Summarising categorical variables

The `type` and `code` variables in our data frame `Prestige` represent categorical variables (i.e. those that take values in a discrete set of non-ordered categories). In this particular case, `code` is represented by numbers, even though the levels do not correspond to amounts. The variable can be analysed as if it is numeric even if this is not sensible (see that last exercise).

The way to summarise categorical variables is in a frequency table, i.e. count the number of times each value occurs:

```
> table(Prestige$type)
```

Add up the counts for `bc`, `prof` and `wc`. How many rows does the data frame have? Why are these two numbers not the same?

There are some `NA`s in `type`. If we want to include these in the table use:

```
> table(Prestige$type,useNA="always")
```

If you want the proportions (also known as relative frequencies) then we can use the function `prop.table`. Note that `prop.table` expects an object of class `table` as input, which means that most of the time we use the slightly odd expression `prop.table(table(...))`, e.g..

```
> prop.table(table(Prestige$type,useNA="always"))
```

3.3 The `summary` command

One of the most useful commands in R is `summary`. Try it:

```
> summary(Prestige$income)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    611    4106    5930    6798    8187   25879
```

Since *income* is a numeric variable, you get a standard 6-number summary of the data (minimum, lower quartile, median, mean, upper quartile and maximum). Such a summary would be unsuitable for a categorical variable, so:

```
> summary(Prestige$type)
  bc prof   wc NA's
  44   31   23    4
```

Notice that the `summary` command automatically reports missing values.

The `summary` command can be applied to almost any R object. It is an example of a generic function this means that it does different things depending on the type of object. At present we're only interested in simple summaries of data objects. Another important use of the command is to summarise the results of fitting a regression model. More on this later in the course.

Exercise: What does summary do if you give it a data frame as an argument? Try it and find out.

3.4 Summarising two variables at once

We can summarise each variable individually using the functions above but often we also want to investigate the joint behaviour of two variables: e. g. the sentence *"taller people have in general bigger waist measurements"* explains the joint behaviour between the two variables: peoples *heights* and *waist measurements*.

Again, we will require different methods depending on whether the variables are numeric or factor.

Summarising two categorical variables

For one categorical variable we produced a frequency table using the function `table`. You can *cross tabulate* two categorical variable also using `table`. I'm sure you have seen many examples of a cross tabulation in news articles.

The Prestige data only contains one categorical variable. So for this part we will use a dataset called `BloodType.csv`, which contains two variables: `ABO` and `Rhesus`. The data is taken from a student lecture group. Read in the data using:

```
> BloodType<-read.csv2(file="BloodType.csv")
```

How many students are included in the data set?

The cross tabulation can be obtained by:

```
> table(BloodType$ABO, BloodType$Rhesus)
```

The first variable is presented in rows and the second in columns. (Remember: rows then columns!)

Summarising two numeric variables

The **covariance** measures the joint variability in two numeric variables. Suppose there are two variables X and Y:

+ In the case that large values of X generally go together with large values of Y and conversely small values of X go together with small values of Y then the covariance of X and Y is large positive.

- If large values of X generally go together with *small* values of Y and conversely small values of X go together with *large* values of Y then the covariance of X and Y is large negative.

0 If there is no clear association between large and small values of X and Y then the covariance of X and Y will be small (positive or negative).

Example: Returning to the Canadian Prestige data: The covariance between `income` and `women` measures, to what extent occupations with a higher proportion of women have a higher average income

```
> cov(Prestige$income,Prestige$women)
```

Well it is negative, but the size of the number is difficult to interpret...

The mathematical properties of covariance are used a lot by theoretical statisticians. It is not easy however to interpret a covariance because its scale depends on the variance of the two variables. The **correlation coefficient** of X and Y is the covariance of X and Y divided by the standard deviation of X and the standard deviation of Y. The result is a number between -1 and 1. The bullet points above given in terms of correlation coefficients are:

+ large X implies large Y and vice versa[4] then the *correlation coefficient* of X and Y is near to 1 (roughly greater than 0.7) .

- large X implies small Y and vice versa then the *correlation coefficient* of X and Y is near to -1 (roughly less than -0.7).

0 If there is no clear association then the covariance of X and Y will be small (roughly between -0.2 and +0.2).

The function to do this in R is called `cor`

[4]umgekehrt

```
> cor(Prestige$income,Prestige$women)
> cov(Prestige$income,Prestige$women)/(sd(Prestige$income)*sd(Prestige$women))
```

Both methods agree. The coefficient is -0.44 so there is a moderate negative correlation between the proportion of women in an occupation and the Income! Let's plot the two variables

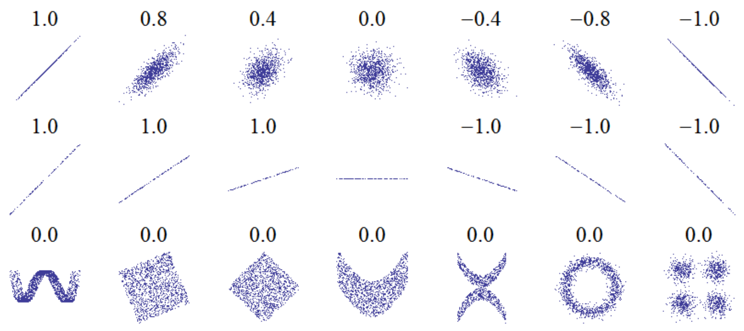
```
> plot(Prestige$women,Prestige$income)
```

There are four points with very high average incomes and a low proportion of women. The classical correlation coefficient is known to be sensitive to outliers. A similar statistic which is not nearly so sensitive to outliers is the *Spearman rank correlation coefficient*

```
> cor(Prestige$income,Prestige$women,method="spearman")
```

The correlation with the "robust" method is even worse. I find this result depressing but not surprising, however the data are now nearly 50 years old: hopefully things have got a little better since then!

Warning Correlation is one of the most misused statistics. Firstly correlation only indicates a linear relationship between the two variables. It is entirely possible for two variables to have a strong but non-linear relationship but have a correlation coefficient of 0.



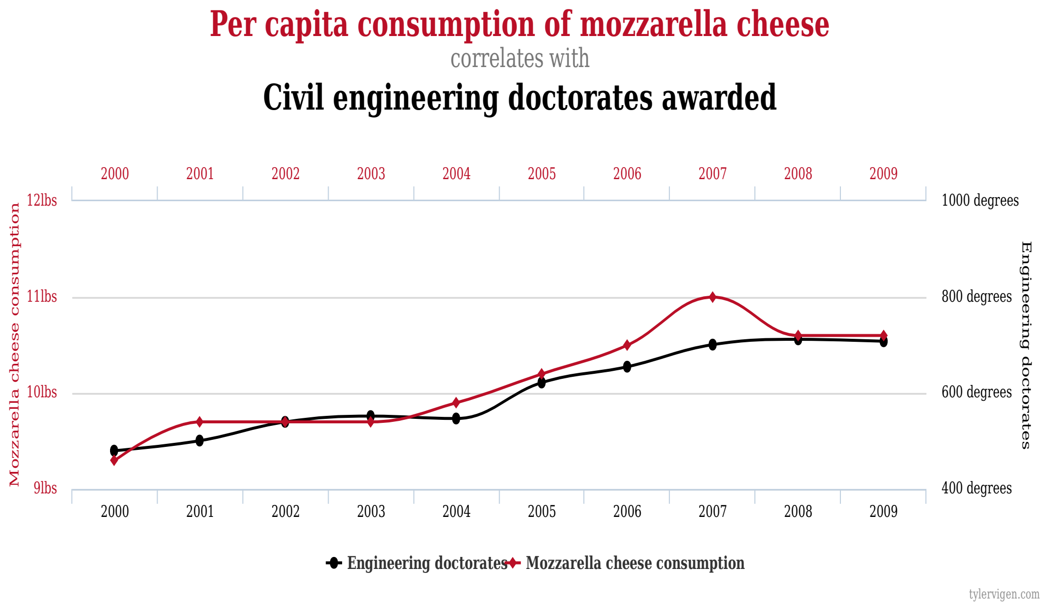
Source: `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`

A second abuse comes from ignoring the important phrase:

Correlation is not causation

Just because a large (positive or negative) correlation is found it does not mean that one variable causes the other one. A large collection of "spurious correlations" has been collected by Tyler Vigen a post-graduate student at the Harvard Law School. Website:`http://tylervigen.com/spurious-correlations`

Just one example is



Summarising One numeric and one categorical variable

The approach is to obtain the same summary statistics as in Section 3.1 for the numeric variable but split by each level in the categorical variable. The traditional way to do this is using the function `tapply`.

```
> tapply(Prestige$prestige,Prestige$type,mean)
```

So the professional occupations are notably more prestigious.

`tapply` belongs to the group of `apply` functions in R. All of these functions arrange the data in some way and *apply* a function to the arranged data. In this case the data is the `prestige` variable and is split into the groups occupation `type`. The third argument is a function name in this case `mean`. `mean` is applied to `prestige` split by `type`. If you want to pass an argument to the function `mean` then you specify this after the function name. E.g.

```
> tapply(Prestige$prestige,Prestige$type,mean,trim=0.1)
```

applies the “trimmed mean” to each group. This means that the largest and smallest 10% of the values are discarded (trimmed). This is sometimes used to avoid problems with outliers when calculating the mean. The function `tapply` is very powerful and you will meet it regularly in this course.

4 Exercises and tidying up

There is plenty more to this subject which will be continued in the double-block next week. For example we have not even started to look at reorganising the structure of a data frame. To round off this week, try the following Exercises.

4.1 Exercise

Reproduce the a selection of the summary statistics on the `medication` dataset. This is a small data set consisting of the time in minutes for blood to clot after patients had been given one of two types of blood coagulation drug types (A or B)

(a) First of all read in the data again just to be sure:

```
> medication<-read.csv(file="Medication.csv")
```

(b) Now translate the variable names

```
> names(medication)<-c("Drug_Type","Time")
```

(c) Answer the following questions by using what you have learnt in the workshop.

  (i) How many patients had each type of drug?

  (ii) What is the mean coagulation time?

  (iii) Produce a summary of the coagulation times.

  (iv) What is the mean coagulation time in type A and in type B?

  (v) (from last week) Produce a box plot of `Time` depending on `Drug_Type`

  Next week we will look at whether this is actually a *significant* difference between the two groups.

4.2 Tidying up

Make sure your script file has sensible comments. At least one comment for each main section. Try to tidy up your code a bit. If you entered a command just trying something out, but which has little to do with that section then delete the non-essential commands.

► Save the script file (source file) again: $\boxed{\text{Strg + S}}$ or *File > Save as*.

► Leave RStudio by typing the command:

```
> q()
```

When R asks you *Save workspace image ...?*, click on **Don't save**!

► Feierabend!