Tools for Science

Automate work and manage complexity

Rik Huijzer

Contents

1 Preface

Artificial intelligence (AI) is lauded by many as the fourth industrial revolution. It promises to make better decisions and can automate more work. One of the largest countries in the world is spending massive resources on autonomous driving, urban cognition, medical imaging, voice intelligence, visual computing, marketing intelligence, video perception, intelligent supply chains, image perception, smart education, smart homes and many more (Wu et al., 2020). In this sense, we live in exciting times. However, there is a common held belief that these automated analyses, machine learning, neural networks and artificial intelligence can only be achieved by large countries, or companies like DeepMind, OpenAI, Microsoft, Netflix and Amazon. This is false. Basically, what we call artificial intelligence is just machine learning (Jordan, 2019) which is basically statistics. Here, it is statistics being applied by software engineers. The benefit of software engineerings is that they are used to managing complexity with tools. So, for software engineers it doesn't matter they work on the newest maching learning systems or more traditional systems. In both cases, the work is quite similar.

There have been failures with AI software, such as Microsoft's chatbot learning offensive language or Apple's Face ID being broken by hackers (Greenberg, 2017). Still, it is mostly success stories about beating the world champion in Go matches (DeepMind, 2020), predicting the stock market, selling more products with better recommendations. The success comes from the scaling capabilities. Once the software is correct, it can be scaled to millions of users or base analysis results on petabytes of data.

However, software engineering isn't part of the typical scientist's skillset. Non-engineering students are taught to do statistical analyses in convenient statistical software with graphical user interfaces, such as Excel, SPSS, JAGS or JASP. The problem with these tools is that they cannot solve any computation problem. In their daily life, they will run into problems which are out of the scope of these convenient tools. This while the **real** tools, general-purpose programming lanuages, have never been easier to use. Everything you need to automate analyses and work with AI is publicly and freely available.

This explains why I decided to write this book. With a background in computer science, I have transitioned to a PhD in Psychology. Here, I see that days are fully spent behind a computer while being hindered by the convenient tools. I hear the same stories from colleagues in the medical, biological, chemical and mathematical sciences. I've even heard professors talk about some of their tasks being laborious while knowing that many great tools can easily automate these tasks. Unfortunately, I know that using these tools cannot be learned in a day; learning only one of them is not enough. Combining the tools in this book is where the real power lies. I've often convinced people of some tool by amazing them with the speed. However, on their way home, they would realise that the tool doesn't work in their workflow. That's why I wrote this book: to show the whole picture instead of just one tool. Also, with this book I aim to present the essentials. These essentials will remain applicable for many years to come and allow you to easily switch to other tools.

1.1 Audience

This book is aimed at scientists. Actually, most chapters will work for most office jobs, but the book is written with scientists in mind. This book shows you the tools to get rid of your tedious tasks and how to manage complexity effectively with the proper software. Software can do this for you because if you read this on a electronic device, then you probably don't know how the transistors, registors, instruction sets, assembly code or application code works. However, this complexity is all hidden away for you and you can assume all parts to do the work for you.

1.2 Reading this book

The chapters will build on top of each other in the sense that later chapters are easier when the tools of easier chapters are used. Also, I've tried to order the chapters by decreasing importance. For example, later chapters include more code which is managed best via version control. Some people will say that this book is too extreme; some of the presented tools could be replaced by a more beginner friendly version. Unfortunately, those friendlier versions will not teach you the basics. It will be easy to switch from the tools in this book to friendlier versions, but not the reverse. Also, friendlier tools are friendlier by assuming things for you. This is nice if you want to stick to their conventions, but if you want something outside of what they provide, you're out of luck. Then, the only way to get it is to go back to the more basic tools or to ask the developer to implement your feature. Also, these basic tools are used by more other tools and people, so more effort is put into the development. That, in turn, usually results in the tools being more reliable, better documented and more performant. As another argument, these basics will allow you to combine the tools more easily, giving you more power. Other people will call this book opinionated, and push for other tools. That is why I've spent a great amount of arguing for the merits of each tool. If you already use another similar too and are happy with it, then please skip the chapter in this book.

1.3 Notation

To keep the text short and clear, this book uses mathematical notation. For example, instead of “the mean for the random samples equals three,” we write

$$\text{mean}(X) = 3,$$

where X are the samples or: let X denote the random samples, then

$$\text{mean}(X) = 3.$$

In this book, I tried to keep the notation consistent, as well as as close to the code as possible. Therefore, a single value will be denoted by a lowercase symbol, such as x and a vector or matrix will be denoted by an uppercase symbol, such as X. By default, a vector is assumed to be a column vector like, for example,

$$X = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Or, more generally, a vector of length n is denoted by

$$X_n = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where $x_1, ..., x_n$ denote single values from the vector X. Similarily, a matrix with n rows and m columns is denoted by $X_{n \times m}$. Note that a vector is just a one dimensional matrix; hence,

$$X_n = X_{n \times 1}.$$

These sizes in the subscript are omitted when possible. Transposition for a vector or matrix is denoted by T. So, for example,

$$X_n^T = [x_1, x_2, ..., x_n].$$

For functions, I try to be explicit about the variables, that is, try to have as little state as possible in the functions. So, for example, instead of writing a linear model as

$$y = w_0 + w_1 x,$$

or

$$y(X) = w_0 + w_1 x_1,$$

I favor

$$\text{lm}(X, W) = w_0 + w_1 x_1,$$

where lm could be any name. On top of this, I will respectively use Greek and Latin letters for measures of the population and samples since this is a common convention in statistics (Smith, 2018).

1.4 Acknowledgements

Many people have contributed directly and indirectly to this book. My PhD supervisors have allowed me to do my work with any tools I prefer. This allowed me to experiment with different tools while keeping in mind that things have to get done.

2 Where it all starts

For most scientists, the biggest part of the work isn't in the lab or in the field. It is, basically, moving text and data around. By this, I mean that reports have to be written and updated, data has to be cleaned, and analyses have to be run, updated and re-run.

To do so effectively, Section 2.1 argues that you should primarily focus on interacting with the computer via text, and this is demonstrated in Section 2.2.

2.1 Interacting via text

To get things done on a computer effectively, the focus should be on interacting with it via text. There are multiple reasons for this. Firstly, the number of things you can express with text scales exponentially with the number of words. For instance: given 26 letters in the alphabet, you can express 26 different things with one letter, $26 \cdot 26$ different things with two letters, to 26^n different things with n letters; also known as ordered sampling with replacement. So, suppose all the programs in the world are required to be 7 letters, like `bigtree`, and all the programs take 3 letters of input. Then, you can choose from $26 \cdot 7 = 8.0 \cdot 10^9$ programs, and each program can take $26^3 \approx 17.500$ different inputs. Imagine having $8.0 \cdot 10^9$ icons on your desktop and, after opening a program, being presented with 17.500 buttons. Another benefit of text is that you can exchange it easily via online communications. A tutorial for a text-based tool includes sentences such as:

> After that, run
>
> ```
> move A B
> ```

whereas, graphical user interface users have to rely on tutorials such as:

> After that, click on the "tools" button in the menu. Then, click on "move" and drag `A` to `B` in the window.

which is confusing and verbose. Another argument is that text is "the most powerful, useful, effective communication technology" because it (Hoare, 2014)

- is the most stable, that is, can be read in many years from now,
- can express many (abstract) things which can't be expressed in images such as "Human rights are moral principles" and

- is the most efficient communication technology.

Finally, tools are written in code, which is basically text. Therefore, the default interface between tools is text, and a GUI is often just a layer on top. In the best case, software built on top of of other software implements all features, but usually the upper layers only contain a subset. From experience, I know that it is extremely frustrating to find these boundaries. Usually, you only find these after investing a lot of time in a tool, and it means that you have migrate your existing project to another tool. This is a bit like building your house from construction plans; only to realise halfway the project that the plan doesn't include a bathroom and that you have to switch over to other construction plans.

2.2 Command line

Before operating systems, such as Microsoft Windows, Apple MacOS and Linux, included fancy graphical user interfaces based around icons, the mouse and draggable windows, all the work was done from the terminal. For example, an MS-DOS computer booting into something looking like

```
MS-DOS version 4.00
Copyright 1981,82,83,84,85 Microsoft Corp.

C>echo Hello
Hello
```

From this terminal, you could do anything you want. You could start programs, move files, edit spreadsheets and send documents to printers. However, although this is quite powerful, it is also unwieldy because the most basic terminals only allow you to do one tasks at the same time. Also, browsing the web is difficult, because it is full of pictures and moving elements which can't be represented nicely via text. That is why modern operating systems, by default, are built around the graphical user interface and still allow you to run something which looks like a terminal in a window. This is also known as a *terminal emulator*. The terminal emulator allows you to execute commands. Specifically, it allows you to type a command which the system will then run. Next, it will show you the output on the next line(s). For example, to make a directory on Windows called `analysis` and rename it to `old-analysis`, you can do

```
Microsoft Windows [Version 10.0.18363.1316]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\John> mkdir tmp

C:\Users\John> cd tmp

C:\Users\John\tmp> mkdir analyis

C:\Users\John\tmp> dir
```

```
[...]
04/01/2021  09:36 PM     <DIR>     analysis
[...]

C:\Users\John\tmp> rename analysis old-analysis

C:\Users\John\tmp> dir
[...]
04/01/2021  09:36 PM     <DIR>     old-analysis
[...]
```

This behaviour with commands and its output on new lines is why a terminal emulator is also known as a *command line*. From the command line, you can start applications via text. In the rest of this chapter, the book will divert for a bit between operating systems, but this is only temporarily.

As a simple example, lets open the site of Stanford University in your favorite browser. In this example, I assume that Firefox is your favourite browser, but you can also use

- Chrome via `chrome`,
- Safari via `safari` *or*
- Edge via `msedge`.

Windows:

Start the Command Prompt and type

```
start firefox https://stanford.edu
```

Apple:

...

Linux:

Start the terminal and type

```
firefox https://stanford.edu
```

2.3 Commands

If you want to know more about commands, this section will list some common ones. Otherwise, you can skip this section and come back to it when you need it.

3 Never losing your files again

To show why the problem of not losing files is important, lets consider Kernel-land. In Kernel-land, the people all carry an extensive manual around which they follow precisely. For instance, when people in this land want to drink water, they read the according section in the manual and lift the cup to their mouth only if the manual instructs them to do so. Also, multiple people carry different manuals, but this is typically no issue since people can interact happily via conventions, such as shaking hands when meeting. The main joy in life is attained by obtaining a newer version of the manual; allowing the Kernel-landers to do new things and move around the place more quickly.

Now, let's say you and your team are responsible for maintaining and extending this manual. Maintenance is necessary because the world changes, so the manual has to be updated accordingly, and extending the manual is necessary because the people in Kernel-land are getting bored. At the same time, you try to minimise risks when updating the manual since earlier changes were disastrous. Once, you have mistakenly changed the line "the bananas can be found on the second *aisle* to the left" into "the bananas can be found on the second *isle* to the left." It look quite a lot of work to figure out this bug since the only practical difference was in the time it took to obtain bananas.

The main complexity of this work is introduced by the size of the manual. When all working in Word documents, merging the results would be laborious. By merging, I mean that some team member takes the newest version of the manual, works on it for a few hours to make changes and then call this the new version. This works until people start to work on the document at the same time. Even if the manual is updated in, say, Section 3 by person A and updated in Section 7 by person B, then these Word documents are difficult to merge. Without taking care, the person who saves his version after the other person *wins*. In other words, the changes of one person will then be overwritten and lost.

Another problem is similar to the problem with the bananas described above. It can be that one small detail in the manual causes hard to predict problems. For example, it might happen that Kernellanders report that the manual tells them to a weird thing like being instructed to brush their teeth in the nearest house. It can then take a lot of work to determine that this only happens on rainy afternoons at Sunday in a town called Sussex, because Sussex is the only place which has 14 streetlights which are all at a 34° angle with the center axis of the road.

These problems are solved by a tool created by Linus Torvalds. Linus is the maintainer of the Linux kernel which, in summary, is the manual for all the Linux computers and servers in the world. Around 2005, Linus had thousands of people working on the manual (Loeliger & McCullough, 2012). He was then reviewing the changes and merged them in the main manual version if it all looked good. A typical merge would take the computer 2 hours (Torvalds, 2005), which Linus wasn't willing to accept. He wanted 3 seconds (Torvalds, 2005) which he managed to obtain by creating *Git*.

Nowadays, Git is the tool that all software engineers agree upon and where countless other tools are build upon. This is an impressive feat because there is a joke that says: "arguing with engineers is like rolling in the mud with pigs; after a while you realise that they like it." If you talk to software engineers about Git, they will agree that it is **the best tool** for the job. Even if you are no software engineer, this is the tool that has been missing from your life. It will ensure that you never lose a file and that you don't have to struggle with files named `1`, `1b`, `1b-final` and `1b-final-final` again.

3.1 All your history

3.2 Getting started

To avoid losing files, we want to store our files at multiple computers. Therefore, we can use Git in combination with GitHub at https://github.com. Basically, GitHub is a website built around Git. If you don't like GitHub, you can also choose to use GitLab (https://gitlab.com), Bitbucket (https://bitbucket.org) or Gitea (https://gitea.io). Because these services are built around the same core, you can easily move your files from one service to the other if you want. First, we have to create a project to store your files or a, so called, *repository*. The definition of repository is "a place where things may be put for safekeeping" (Pickett, 2018). To get started with GitHub, create an account and a new repository via the web-interface (https://github.com/new). You can choose a nice name and ignore the checkboxes. In this example, we call the repository `novel-paper` and say that it is created by an example user called `researcher`. For the rest of this example, replace `novel-paper` and `researcher` with respectively your repository name and username. Next, install Git via the instructions at https://git-scm.com/downloads.

Now, the Git robot can be talked to via the terminal. (The terminal is described in Chapter 2.) For Git, we need to learn a few words to handle 99% of all the use-cases.

First, we have to make a copy of the new repository `novel-paper` on our own computer so that we can make changes. Go to a folder where you want to store the repository and tell the robot to clone the repository:

```
$ git clone https://github.com/example/novel-paper
Cloning into 'novel-paper'...
warning: You appear to have cloned an empty repository.
```

or, if you did set one or more of the checkboxes:

```
$ git clone https://github.com/example/novel-paper
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Your output should look similar to one of these two outputs. Here, Git told us that some files were received (3 in this case) and everything is done. In both cases, you will now find a folder called `novel-paper` and we can start to add a file. Lets create a ridiciously important file called `ridiciously-important-file.txt` and place some text in it

`ridiciously-important-file.txt`:

```
The solution to my research can be found in Devlin & John (2020).
```

(And, no, that is not a typo. John can, in fact, be a surname like in the name Elton John, see Atkinson & John (1991).)

After doing this, we can see one of the awesome features of the Git robot. It knows exactly how everything looked before your changes and can show what has changed. To get a sort of summary of the changes, use

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ridiciously-important-file.txt

nothing added to commit but untracked files present (use "git add" to track)
```

This shows that we have added `ridiciously-important-file.txt`. To store this file online, lets *add* all changed files in preparation to pushing the changes to the server.

```
$ git add .
```

This is not definitive yet, we can still check whether everything looks good:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   ridiciously-important-file.txt
```

As stated in the introduction of this chapter, Git is made to track complex changes.

One way to do this is to describe in words what you’re changing. When the previous `git status` looked good, we can make these changes permanent, or in other words, *commit* ourselves to these changes with

```
$ git commit -m 'Add important file'
[master (root-commit) b4f9e1d] Add important file
 1 file changed, 1 insertion(+)
 create mode 100644 ridiciously-important-file.txt
```

and *push* them to the server

```
$ git push
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 303 bytes | 303.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/researcher/novel-paper
 * [new branch]      master -> master
```

You can now see your file online by going to

```
https://github.com/<username>/<repository name>
```

3.3 Collaborating

4 Quick text editing

As an office worker, most of your day is spent on manipulating symbols. You write papers, reports, notes, presentations, abstracts and maybe use writing to structure your thoughts. Many of these symbol manipulations occur often. For instance, think about the number of times that you have copied or deleted a full sentence, one word, or one or more paragraphs. Over time, the time and effort spent on these tasks count up. Yegge (2012) even tells a story about how Mailman was an internal application built on top of *Emacs*, an advanced text editor:

> People still love it. To this very day, I still have to listen to long stories from our non-technical folks about how much they miss Mailman. [...] Last Christmas I was at an Amazon party, some party I have no idea how I got invited to, filled with business people, all of them much prettier and more charming than me and the folks I work with here in the Furnace, the Boiler Room of Amazon. Four young women found out I was in Customer Service, cornered me, and talked for fifteen minutes about how much they missed Mailman and Emacs, and how Arizona (the JSP replacement we'd spent years developing) still just wasn't doing it for them.

This is mostly because these editors help you in getting things done more quickly. In the rest of this chapter, I will provide a quick-start for Vi which is roughly the same as Vim (www.vim.org) and Neovim (https://neovim.io). Vim is an improved version of Vi and Neovim is created by a group of developers who weren't happy with Vim and decided to work on their own copied version. Both editors can be used mostly interchangibly and that is why I mean Vi, Vim or Neovim when I mention Vim. I personally prefer Neovim, but Vim works great too. That this book focusses on Vim doesn't mean that I don't like it's direct competitor, Emacs. I think both Vim and Emacs can save users tremendeous amounts of time. For me, Vim seemed more suitable, because it's less oriented around only a few keys making it less prone to repetitive strain injury. Also, Emacs wasn't as quick. On the other hand, Emacs has many great productivity enhancing tools such as Org-mode (https://orgmode.org). So, both have their pro's and con's. If you're unable to do touch typing, then consider reading Chris Wellons' blog about Vim and touch typing (https://nullprogram.com/blog/2017/04/01/).

Vim enhances productivity by introducing different modes. Different modes allow the behaviour of a key, say `d`, to change depending on the mode. In this modal editor, for example, `d` types d in insert mode; like a normal text editor. However, when pressing `Esc`, the editor switches to normal mode and now `d` will appear to do nothing. Only if you press `d` twice, the line on which you are will be deleted. These editors leverage the higher number of possible combinations available with only a few key presses. Specifically, assuming 35 keys and that we can press a few keys, say 3 keys, we have $35^3 = 42.875$ (ordered sampling with replacement) possible instructions that we can give the editor.

Like most tools in this book, the biggest problem for widespread adaptation of the tool is the learning curve. The trick here is to just do a project in Vim and learn while doing. For the first few days, this will hurt your brain. After a few days, you'll start to be quicker than you used to be and that speed improvement will only improve over time. You will also be able to switch between normal and advanced text editors without even thinking about it. Another trick is to just use it and pay attention to repetitive/boring key presses. Then, search a bit on the internet and you'll probably find instructions on how to do it more quickly in Vim. Note that Vim works best with the tools presented in this book. Not all text-based tools have Vim (or Emacs) support. For example, it is not available in Microsoft Office.

4.1 Commands

Use `Esc` to switch to normal mode. Then, the commands listed in Table 4.1 can be used.

Table 4.1: List of the most important Vim commands.

Keys	Description	Illustration
Esc	Cancel or switch to normal mode	
:q	Quit Vim	
:w	Write/save file	
:wq	Write and quit	
j	Down	Figure **??**
k	Up	Figure **??**
h	Left	Figure **??**
l	Right	Figure **??**
:e `file`	Open `file` for editing	
/ `text`	Search for `text`	

Moving the cursor in Vim. These keys have the benefit that you don't have to move your hand to the arrow keys on the keyboard.

4.2 Plugins

TOOD: Mention the thing I’m using to show tabs.

5 Converting text files

5.1 LaTeX

5.2 Pandoc

5.3 Presentations

6 Automating tasks

From this chapter on, the book uses the Julia programming language.

Contents:

- Scripting in Julia (to generate files with Pandoc).

7 Cleaning and preparing dirty data

8 Predicting the future

If you are like me and, basically, all scientists, you want to predict the future. With good predictions, you can do things like earning tons of money via stock markets or successfully treating patients, scaling up your favorite chemical process or getting your rocket into space. Even historians work like this, they meticulously investigate the past and use that knowledge to make predictions. For example, Harari (2014) is a book by Yuval Noah Harari describing a brief history of humankind and is superseded by Harari (2016) where Harari describes the future. These predictions are obtained by first reducing the complexity of the world by arguing that the world can be simplified in a certain way. This is also known as using a *model*. Then, after convicing the audience that the model is an accurate simplification of the real world, it can be applied to make a prediction. In some situations, particularity science, this approach is not considered precise enough. For example, to get new medication approved, it isn't sufficient to argue that the medication is effective because

- some patients say that they feel less sick after getting the medication,
- many patients found the medication tasty *and*
- you feel good after handing out the medication.

Instead, approval is obtained by applying models on data. A similar conclusion, based on a statistical model, would be

- patients who received the medication have significantly lower values of X which indicates that they are less sick.

So, it is expected that the world is simplified into a statistical model and to use that to prove the effectiveness of the medication. Unfortunately, this tradeoff between accuracy and complexity is a well-known problem in science. To see why this is the case, lets take a look at determinism which will lead us to causality and distributions.

Determinism ...

Different people group these models in different ways. Often, the groups are dichotomized which can simplify the discussion, but do note that there is usually a continuum, that is, not all models fall strictly into one category. For example, statistics can be separated into algorithms and inference (Efron & Hastie, 2016). Then, classifying which mail is spam can be seen as a purely algorithmic problem. If the algorithm works, then it doesn't really matter to know why it works. In science, the focus lies more on inference, that is, to know **why** a model works and what can be **inferred** from it.

Another common distinction is between frequentist and Bayesian statistics. For the purposes of this book, this is an useful distinction since different Julia libraries can clearly be classified to be either frequentist or Bayesian. Unfortunately, for the purposes outside this book, the reader is encouraged to learn about both paradigms because both have their strengths and weaknesses. Luckily, much knowledge is applicable to both paradigms like, for example, distributions, cross-validation and visualizations.

One core element of these models are distributions which are discussed in Section 8.1.

8.1 Distributions

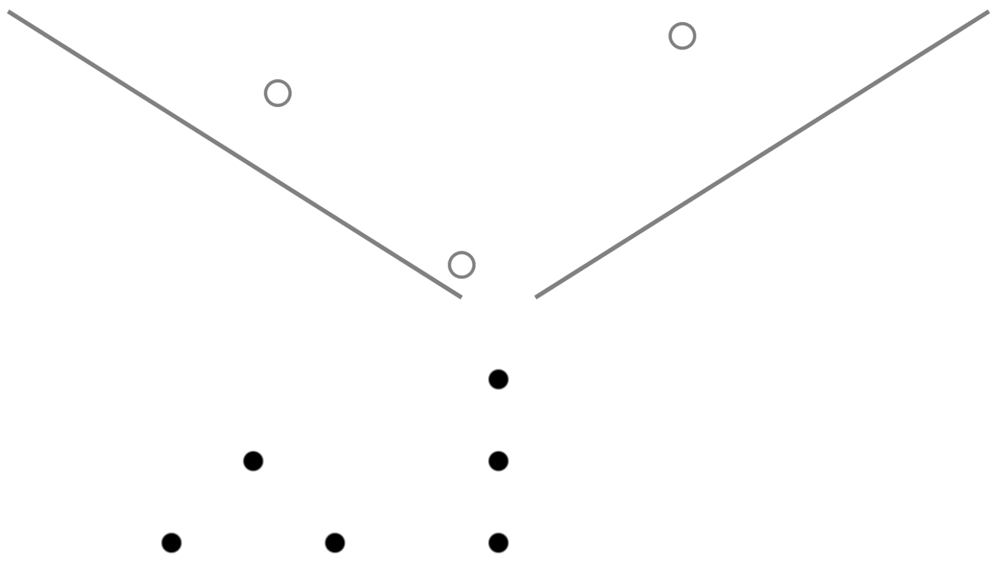

Figure 8.1: Schematic depiction of a Galton box; based on an image from Wikimedia Commons.

8.2 Frequentist

After introducing some common statistical notions in the previous sections, we can now focus on linear models and their corresponding statistical tests. From linear models, it is possible to formulate many common tests, or very close approximations, including Pearson correlations, t-tests and ANOVAs (Lindeløv, 2019). Note that linear models are as valid in the Bayesian paradigm as in the frequentist paradigm. However, here we focus on the combination with the frequentists tests, which is why these models are listed under the frequentist section.

In line with Bishop (2006), generally, we can write a linear model lm as

$$\text{lm}(X_v, W_m) = w_0 + w_1 x_1 + ... + w_m x_v,$$

with input variables X_v and weights W_m. Note that this means that the data with n measurements can be stored in a $n \times v$ matrix. For the rest of this chapter, we assume that the errors are Gaussian.

8.2.1 One sample

Say that you have a vector of sample values A and want to test whether the mean of the population is x. Lets denote the mean of the population and sample respectively by μ and μ_s and, for illustration purposes, lets generate some data:

```
function one_sample_data()
    n = 12μ
     = 10σ
     = 3
    Random.seed!(1)
    A = rand(Normalμ(, σ), n)μ₀
     = μ + 4
    (n, μ, σ, A, μ₀)
end
```

which is visualized in Figure 8.2.

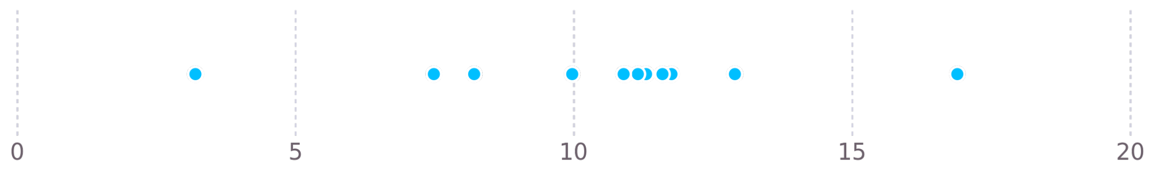

Figure 8.2: 12 points generated from a normal distribution with μ = 10 and σ = 3.

Okay, so let's say that we want to know whether this data was generated by a process having μ_0; this is our null hypothesis and we want to check whether the data is significantly different, that is, we want to test whether $\mu \neq \mu_0$. A simple way to guess μ is to take the mean of the sample. However, these means on their own don't provide much insight into **how** likely it is that $\mu = \mu_0$, see Figure 8.3.

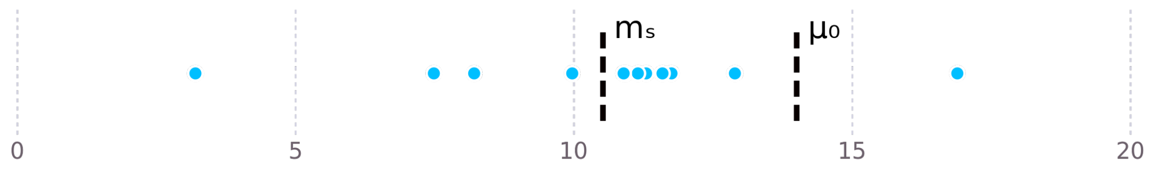

Figure 8.3: 12 points generated from a normal distribution with μ = 10 and σ = 3. Also, the sample mean m_s and the null-hypothesis mean μ_0 are shown.

To estimate this, we can guess the distribution of the sample. As already stated, we assume that data is generated from a normal distribution and we can use the sample mean and standard deviation to guess the distribution parameters. This distribution is depicted in Figure 8.4.

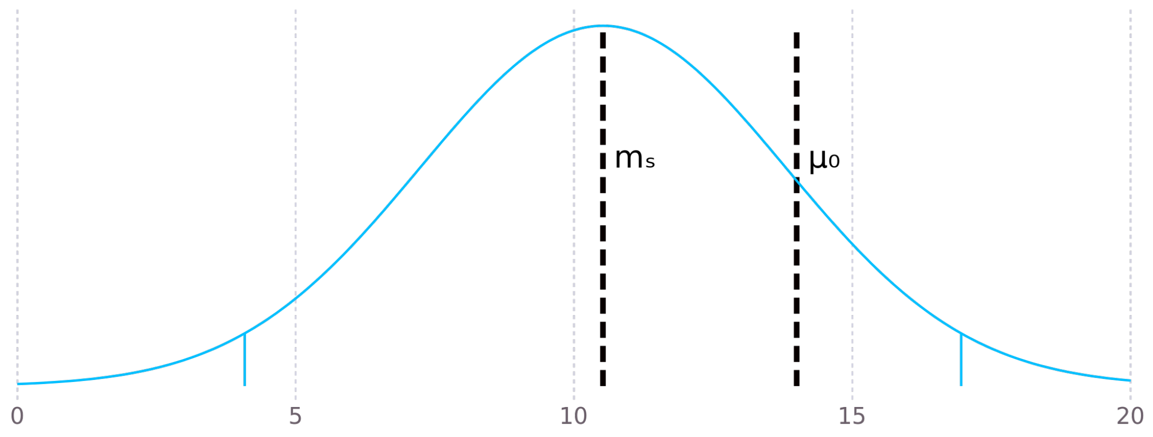

Figure 8.4: Normal distribution for the sample mean and standard deviation. The means for the sample mean m_s and null-hypothesis μ_0 are indicated by dashed black lines. The bounds for the 95% confidence interval are indicated by blue solid lines.

Now that we have this distribution, we could come up with a formal conclusion. For example, we can say that $\mu = \mu_s$ if μ_s lies within the 95% confidence interval of the estimated distribution. Then, in this case, we would now conclude that the means do not significantly differ.

Unfortunately, for small samples, the normal distribution isn't good enough. This is due to the fact that ...

$$\text{one\_sample}(X_v, W) = w_0$$

Table 8.1: Comparison of multiple models being fitted on our sample data with alternative mean $\mu_0 = \mu + 4 = 14$.

model	p	lower	upper
t-test	0.004	8.4	12.6

8.3 Bayesian

9 Running computations automatically

Continuous integrations (CI) and continous delivery (CD) allow software developers to make fewer mistakes by validating code changes automatically before applying them in the codebase. This is elaborated on by GitLab: CI/CD is about ensuring "the delivery of CI-validated code [...] via structured deployment pipelines."

This sounds complex, but actually isn't. CI arose from the need to validate code changes, in version control such as Git, automatically. Upon each code change, a server takes the code and runs all the tests. When the tests pass, you can be more sure that the changes are correct; when the tests fail, you know that something is broken. This is particulary helpful for code reviews when deciding whether the changes should be merged in the codebase, see Figure 9.1.

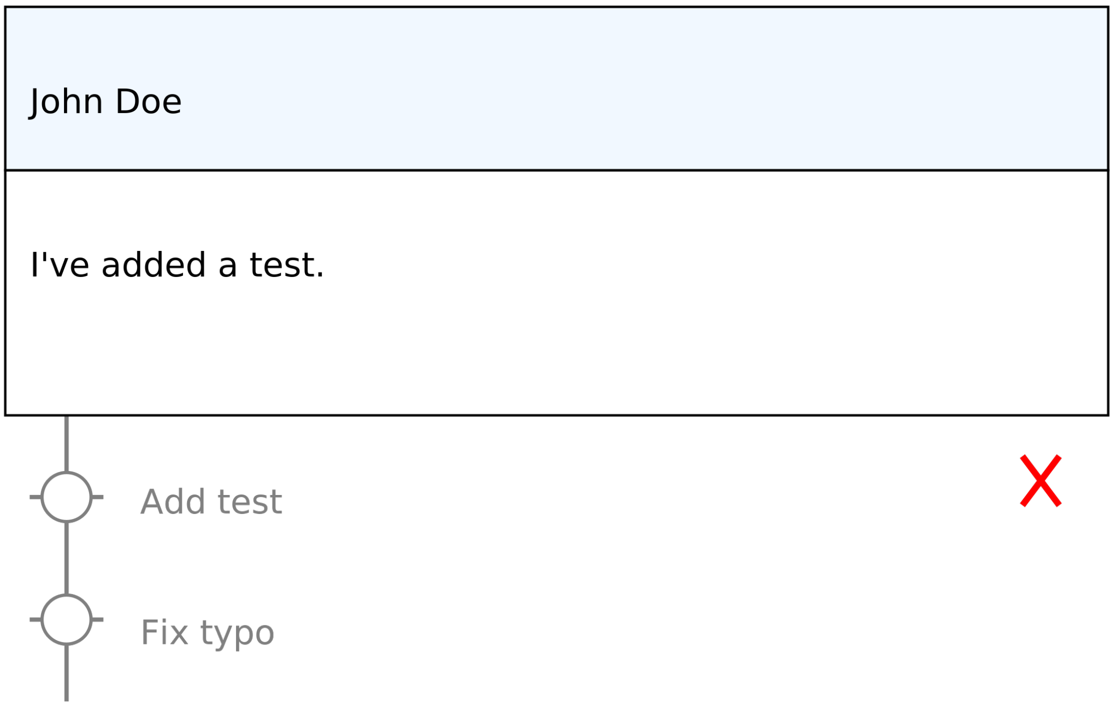

Figure 9.1: Simplified view of the GitHub interface for pull-requests. It shows that John Doe did a failing commit and fixed it afterwards.

CD is also about running computations upon code changes. This time however, the computations can do things like compiling the code into an executable. The executable can then continuously be delivered to the end-user, hence the name CD.

Interestingly enough, this idea is much more powerful than just CI/CD, which is why GitHub calls it GitHub Actions to "automate anything." (You can read this as: *automate anything which can be controlled, directly or indirectly, from a computer capable of running the GitHub runner.*) Like mentioned before, upon each code change we can

- run tests, and/or
- compile a program,

but we can also

- generate a website (like the one you're looking at right now),
- generate a book, or
- run a backup.

TODO: Add Compose.jl image here.

In essence, CI/CD and Workflows are about **linking computations to text**.

(Actually, you can automate anything which can be managed by a GitHub runner; this includes most, but not all, systems.)

10 Appendix

10.1 Versions

This website is built with Julia 1.6.0 and

```
Books v0.4.2 `https://github.com/rikhuijzer/Books.jl#main#main`
Colors v0.12.7
Compose v0.9.2
DataFrames v1.0.1
Distributions v0.24.18
Gadfly v1.3.2
HypothesisTests v0.10.3
Pkg
Random
Statistics
```

Specifically, it is generated by https://github.com/rikhuijzer/tools at commit `2fa8034758559bb93a19e4537501622d6455e05e`.

10.2 i3wm

References

Atkinson, R., & John, E. (1991). *Interview with Elton John*. Laughing Stock Productions.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

DeepMind. (2020). *AlphaGo*. https://deepmind.com/research/case-studies/alphago-the-story-so-far

Efron, B., & Hastie, T. (2016). *Computer age statistical inference* (Vol. 5). Cambridge University Press.

Greenberg, A. (2017). Hackers say they've broken face ID a week after IPhone x release. *Wired, November*, *12*.

Harari, Y. N. (2014). *Sapiens*. Bazarforlag AS.

Harari, Y. N. (2016). *Homo deus: A brief history of tomorrow*. Random House.

Hoare, G. (2014). *Always bet on text*. https://graydon2.dreamwidth.org/193447.html

Jordan, M. I. (2019). Artificial intelligence—the revolution hasn't happened yet. *Harvard Data Science Review*, *1*(1). https://doi.org/10.1162/99608f92.f06c6e61

Lindeløv, J. K. (2019). *Common statistical tests are linear models*. https://lindeloev.github.io/tests-as-linear/

Loeliger, J., & McCullough, M. (2012). *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.

Pickett, J. P. (2018). *The american heritage dictionary of the english language*. Houghton Mifflin Harcourt.

Smith, W. (2018). *The mathematical symbols used in statistics*. https://ocw.smithw.org/csunstatreview/statisticalsymbols.pdf

Torvalds, L. (2005). *Kernel SCM saga.*. https://marc.info/?l=linux-kernel&m=111288700902396

Wu, F., Lu, C., Zhu, M., Chen, H., Zhu, J., Yu, K., Li, L., Li, M., Chen, Q., Li, X., & others. (2020). Towards a new generation of artificial intelligence in china. *Nature Machine Intelligence*, *2*(6), 312–316. https://doi.org/10.1038/s42256-020-0183-4

Yegge, S. (2012). *A Programmer's Rantings: On Programming-Language Religions, Code Philosophies, Google Work Culture, and Other Stuff* (Note: also available at https://sites.google.com/site/steveyegge2/tour-de-babel, Ed.). Hyperink Inc.