Code Quality Handbook

By

Mikkel Freltoft Krogsholm

Compiled from GitHub

.

Why a Code Quality Handbook?

The aim of the Code Quality Handbook is to create a handbook, that you can use when you are starting on a project or include in a project you are already on.

Its goal is to provide you with a good overview of what to do in a project and how to do it in order to maximize the quality of your code, and in that regard the project as well.

We want to make sure that your code works every time, under every condition, and is written in a way that makes sense to you and to others. Because if it does not, then your code and project will eventually run into difficulties.

- If you are doing research with a company then it means doing research that can be reproduced - every time.
- If you are helping a client with their coding in a business setting, then it means writing production-ready code.
- If you are just writing code alone in the dark it means writing code your future self will understand.

The book is a work in progress and will evolve over time as new things get added and as we learn and improve on the different chapters.

The ambition of this book is quite big: to be a single point of reference when you start a new project.

That means that we will try to cover everything from documentation, and coding style to security and even ethics. We want to make good programmers out of good data scientists.

And please contact us if anything is missing from this book.

Table of Contents

Chapter 1

Documentation

AUTHOR: MIKKEL FRELTOFT KROGSHOLM

Code should be easy to understand. That means that it should be written to minimize the time it would take for someone else to understand it. You do that by thinking hard about the choices you take when you code, and what and when to comment your code. The purpose of commenting is to help the reader know as much as you did when you wrote the code, and everything was fresh in your memory.

1.1 Help People Understand

I think that a good metric for how extensive your documentation should be is to minimize the time needed for someone to understand it and your code. If one more line of comments will increase that, then write it - if it will not, then leave it out. Comments also take time to read, so if your comment is not helping people to understand your code better, then leave it out. Aim for your comments to have a high information-to-space ratio. But avoid "crutch comments" that make up for bad code (such as a bad function name) — fix the code instead. The best comment is clearly understandable code.

A good idea is to look at your code from a fresh perspective whenever you can. Step back and look at it as a whole - is everything making sense and is there any way another coder or the future you will misunderstand the code because understandability triumphs speed and clever code every time. Nothing dies faster than code that cannot be maintained.

One technique is to describe your code in plain English and use that description to help you write more natural code. This technique is deceptively simple but very powerful. Looking at the words and phrases used in your description can also help you identify which subproblems to break off. If you cannot describe the problem or your design in words, something is probably missing or undefined. Getting a program (or any idea) into words can really force it into shape. You have a lot of ideas and thoughts in your head when you are coding. If they can help other coders understand your work, then write them down as comments. These thoughts can be stuff like:

- Insights about why code is one way and not another ("director commentary").
- Flaws in your code, by using markers like TODO: or XXX:.
- The "story" for how a constant got its value.
- Put yourself in the reader's shoes: Anticipate which parts of your code will make readers say "Huh?" and comment those.
- Document any surprising behavior an average reader would not expect.
- Use "big picture" comments at the file/class level to explain how all the pieces fit together.
- Summarize blocks of code with comments so that the reader doesn't get lost in the details.

1.2 Make it Look Good

Everyone prefers to read code that is aesthetically pleasing. Just like when you are reading a book - or this text for that matter. What is easier on the eyes is also easier to understand. By "formatting" your code in a consistent, meaningful way, you make it easier and faster to read. Here are some tips for formatting your code:

- If multiple blocks of code are doing similar things, try to give them the same silhouette.
- Aligning parts of the code into "columns" can make code easy to skim through.
- If the code mentions A, B, and C in one place, don't say B, C, and A in another. Pick a meaningful order and stick with it.
- Use empty lines to break apart large blocks into logical "paragraphs". In R for instance, you are allowed as much white space as you want. Use it.

1.3 Break Down Big Blocks

Breaking a large function into smaller ones is also a form of documentation. It documents the code by describing the new subfunction with a succinct name and it helps the reader identify the main “concepts” in the code. On an aside, it also makes it easier to write tests and later debug your code. So be aggressive in breaking down complex logic wherever you see it. A simple technique for organizing your code is this: do only one task at a time. If you have code that is difficult to read, try to list all the tasks it is doing. Some of these tasks might easily become separate functions (or classes). Others might just become logical "paragraphs" within a single function. The exact details of how you separate these tasks are not as important as the fact that they are separated. The hard part is accurately describing all the little things your program is doing.

Write as little new code as possible. Each new line of code needs to be tested, documented and maintained. Further, the more code in your codebase, the "heavier" it gets and the harder it is to develop. You can avoid writing new lines of code by:

- Eliminating nonessential features from your code and not over-engineering
- Rethinking requirements to solve the easiest version of the problem that still gets the job done
- Staying familiar with standard libraries by periodically reading through their entire APIs

1.4 Document Outside Functions

By outside function I mean document what your function is doing. So this is looking at your function from the outside and documenting it as a whole and not the individual parts of the function.

1.4.1 In R

Use R’s ‘roxygen2‘ framework to thoroughly describe your function. You can see an example below:

```r
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
   x + y
}
```

This means giving the function a proper title, telling what goes into the function (@param) and what comes out (@return), and giving a few examples (@examples).

You can read much more about documenting your code in R here in Hadley Wickhams book on writing R packages: R Packages: Object Documentation.

1.4.2 In Python

TO DO

1.5 Document Inside Functions

By inside functions I mean document what each piece of code inside your function is doing.

1.5.1 In R

In R you can write comments in your code using the '#' like this:

```r
add <- function(x, y) {
# This part adds the numbers together.
    x + y
}
```

1.5.2 In Python

TO DO

1.6 Further Reading

- The Art of Readable Code
- R Packages: Organize, Test, Document, and Share your Code [Online Version]
- Tidyverse Styleguide: Code Documentation [Online version]

1.7 Udemy Resources

No courses at the moment that cover this.

Chapter 2

Git and Version Control Best Practices

AUTHOR: HARSH JAIN

This chapter is about best practices in git.

Git is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source code management in software development, but it can be used to keep track of changes in any set of files.

In *Think Big* we use it to keep track of the code we are developing - that includes everything from engineering to data science. Data science IS software development, and you need to use Git every time. That is why we have written this chapter on Git and Version Control Best Practices.

2.1 Commit Related Changes

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other team members to understand the changes and roll them back if something goes wrong. With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.

2.2 Commit Often

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it's easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard both to solve conflicts and to comprehend what happened.

2.3 Don't Commit Half-Done Work

You should only commit code when it's completed. This does not mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But don't commit just to have something in the repository before leaving the office at the end of the day. If you're tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git's "Stash" feature instead.

2.4 Test Before You Commit

Resist the temptation to commit something that you "think" is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing/sharing your code with others.

2.5 Write Good Commit Messages

Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions:

- What was the motivation for the change?
- How does it differ from the previous implementation?

Use the imperative, present tense ("change", not "changed" or "changes") to be consistent with generated messages from commands like git merge.

If you are using a requirement management/task tracking tool (like Jira) then always link your commit to a task, if you don't have a task then create one. For example (where 'DAT-281' is the task):

```
DAT-281: This is a new feature.
This feature supports Kerberos,
which will be used by JupyterHub.
```

If you are not using a requirement management/task tracking tool then define some categories for commit like below:

- feat: feature implementation
- fix: bug fix
- chore: small change
- doc: documentation change
- build: build related change

By doing that, we can easily filter relevant commits through a category, e.g.:

```
git log --all --grep='feat'
```

When you commit, use the first line as a summary and add a detailed description in the following paragraph, for example:

```
feat: this is a new feature
This feature supports Kerberos,
which will be used by JupyterHub.
```

2.6 Version Control is Not a Backup System

Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your version control system like it was a backup system. When doing version control, you should pay attention to committing semantically (see "related changes") – you should not just cram in files.

2.7 Use Branches

Branching is one of Git's most powerful features. Git provides quick and easy branching. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, experiments, and ideas.

Below is an example of a git branching model. You can read more about it at this link: A Successful Git Branching Model.

2.8 Agree on a Workflow

Git lets you pick from a lot of different workflows: long-running branches, topic branches, merge or rebase, git-flow, and so on. Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows, and (maybe most importantly) on your and your teammates' personal preferences. However, you choose to work, just make sure to agree on a common workflow that everyone follows.

Read more here: Workflow Comparison.

2.9 Recommended Git Flow

We have included a template for a recommended git flow that we suggest you use:

- Branch out of the main branch: git checkout -b '<feature_id>_title_description'
- Commit code: git commit -m 'my message'
- When done with code and testing:
  - Checkout latest main branch version: git checkout main_branch‘ and ‘git pull
  - Rebase your branch on the main branch: git checkout '<feature_id>_title_description' and git rebase main_branch
  - After fixing the rebasing, merge with squash the feature branch into the main branch: git checkout main_branch and git merge –squash '<feature_id>_title_description'
  - Push to upstream: git push origin main_branch

2.10 Useful Tools

- gitk (branch history visualization)
- nice git prompt (add this to your ‘ /.profile‘)

```
GIT_PS1_SHOWCOLORHINTS="yes"
source /usr/local/etc/bash_completion.d/git-completion.bash
source /usr/local/etc/bash_completion.d/git-prompt.sh

PS1='\n[\u@\h:\w$(__git_ps1)]\n$'
```

2.11 Do Enforce Standards

Having standards is a best practice and will improve the quality of your commits, codebase, and probably enhance git-bisect and archeology functionality, but what is the use of a standard if people ignore them? Checks could involve regression tests, compilation tests, syntax/lint checkers, commit message analysis, etc. Of course, there are times when standards get in the way of doing work, so provide some method to temporarily disable the checks when appropriate.

Traditionally, and in some people's views ideally, you would enforce the checks on the client side in a pre-commit hook (perhaps have a directory of standard hooks in your repo and might ask users to install them) but since users will often not install said hooks, you also need to enforce the standards on the server side. Additionally, if you follow the *commit-early-and-often-and-perfect-it-late* philosophy that is promoted in this chapter, initial commits may not satisfy the hooks. Enforcing standards in an update hook on the server allows you to reject commits that do not follow the standards.

Read more here: Git Hooks.

2.12 Miscellaneous Do's

- Copy/move a file in a different commit from any changes to it
- (Almost) Always name your stashes
- Protect your bare/server repos against history rewriting

2.13 Miscellaneous Don’ts

- Do not commit anything that can be regenerated (for example binaries, object files, jars, .class, etc.) from other things that were committed.
- Do not commit configuration files, specifically configuration files that might change from environment to environment or for any reason.

2.14 Further Reading

- Pro Git
- Git for Computer Scientists
- Other Resources
- Git Wiki

2.15 Udemy Resources

Git Complete: The Definitive, Step-by-Step Guide to Git

Chapter 3

R Coding Style

AUTHOR: FEBIYAN RACHMAN

Having a style that is followed by R-using Data Scientists or Engineers will benefit *Think Big Analytics* or *Teradata* in the long run. The style needs to be enforced to make the code more easily understandable and consistent.

It is good to keep in mind that someday there will be someone reading your code to solve similar problems, and it is important for them to understand it as soon as they read it.

There are multiple R-style standards available on the internet like Google and Tidyverse. This section heavily borrows from Tidyverse and is compacted to address the most common questions.

Let's start with naming conventions.

3.1 Naming Convention

Names that you give to R files, variables, constants, and functions should be meaningful, short enough to be concise, and yet long enough to be easily understood. Easily understood names are the key to quality code, and unless, performance/efficiency is critical, it should be the first priority.

3.1.1 Files

TODO / TO DISCUSS: Should file names start with nouns or verbs?

Don't use camel case in file names. Use underscores to separate words. Avoid capitalization of function names or packages since R is case-sensitive. Use a colon before enumerations or bulleted lists.

```
calculate_logins.R
```

When there are files that need to be executed sequentially, use numbers as a prefix.

```
00_calculate_logins.R
01_calculate_logouts.R
```

3.1.2 Identifiers

Variables

Generally speaking, variable names should be in the form of a noun. The preferred syntax for variable names is all lowercase letters and words separated with underscores (e.g. 'variable_name'). There are style guides that prefer names with camelCase, and that is acceptable too.

```
# Preferred
variable_name
# Okay
variableName
```

For Boolean variables, make sure you prefix them with 'is' followed by an adjective.

```
isGood <- true
```

Functions

Function names should start with a verb. Function names need to be in non-capitalized letters and are separated using underscores.

A 1 to 2-word-long function name is ideal. A 3-word-long function name is still okay. Rethink your function names if it exceeds 3 words or is too long in character length.

```r
# Preferrable
add_numbers <- function(a, b) {
  a + b
}
```

3.2 Syntax

3.2.1 Quotes

Quotes are used to surround a text string. Use ’"’, not ’’’, for quoting text. The only exception is when the text already contains double quotes and no single quotes.

```r
"Text"
'Text with "quotes"'
'Text with "double" and \'single\' quotes'
```

3.2.2 Spacing

Put a space before and after ‘=‘ when naming arguments in function calls. Most infix operators (‘==‘, ‘+‘, ‘-‘, ‘<-‘, etc.) are also surrounded by spaces, except those with relatively high precedence: ‘ˆ‘, ‘:‘, ‘::‘, and ‘:::‘.

Always put a space after a comma, and never before (just like in regular English). Prefer parentheses over irregular spacing to highlight operator precedence.

```
# Good
average <- mean((feet / 12) + inches, na.rm = TRUE)
sqrt(x^2 + y^2)
x <- 1:10
base::get

# Bad
average<-mean(feet/12 + inches,na.rm=TRUE)
sqrt(x ^ 2 + y ^ 2)
x <- 1 : 10
base :: get
```

Place a space before '(', except when it is part of a function call.

```
# Good
if (debug) show(x)
plot(x, y)

# Bad
if(debug)show(x)
plot (x, y)
```

3.2.3 Indentation

Your code statements, most of the time, will consist of more than one level of hierarchy. The indentation will make your code more readable if used correctly.

Curly braces, '{}', define the most important hierarchy of R code. To make this hierarchy easy to see, **always indent the code inside by two spaces**.

A symmetrical arrangement helps find related braces: the opening brace is the last, and the closing brace is the first non-whitespace character in a line. Code that is related to a brace (e.g., an if clause, a function declaration, a trailing comma, etc.) must be on the same line.

```
# Good
if (y == 0) {
  if (x > 0) {
    log(x)
  } else {
    message("x is negative or zero")
  }
} else {
  y ^ x
}

# Bad
if (y == 0)
{
    if (x > 0) {
      log(x)
    } else {
  message("x is negative or zero")
    }
} else { y ^ x }
```

It is OK to drop the curly braces if you have a very simple and short ’if statement’ that fits on one line. If you have any doubts, it is better to use the full form.

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

3.3 Documentation

Documentation increases the understandability of your code and usually comes in the form of comments alongside the code. The following guide makes sure your comments [missing text].

3.3.1 File/Function Comments

Always indent with one space after ‘#’. Use ‘roxygen2‘ to automatically create Rd files for you

```
#' Title contains the general idea of the file/function
#'
#' @description
#' This is some description, you can omit the tag above is the description is
#' quite short, but generally you will need to keep it around.
#'
#' @param key This is a parameter definition for a function. If it has multiple
#'   as column headings.
#'
#' @examples
#' sin(pi)
```

3.3.2 Code Comments

When commenting a line or a block of code, make sure the comment also answers the WHY. The WHAT can be explained by your code.

```
# Age of students we're interested in is between 15 and 18
if (age => 15 & age <= 18) {
  # Inner comments need to follow the indentation, and this is the
  # second line of the comment
}
```

3.3.3 Inline Comments

Inline comments can be used for variable declarations and library loading statements. Typically, inline comments should be used for short-length statements only. Inline comments should be as concise as possible, too.

```
y <- 10  # This is an okay inline comment
x <- 5     # Short comments only
```

3.4 Linting

Linting is the process of running a program that will analyze code for potential errors. It is sometimes called "static code analysis" and can be used to find style violations.

3.4.1 Lintr

A popular linting library in R is lintr. The library is configurable, but the defaults adhere to the aforementioned style guide.

Here's how one can use it:

```r
install.packages('lintr')
library(lintr)
# Open the file interactively
file_to_check <- file.choose()
lint(file_to_check)
```

3.4.2 Styler

The goal of styler is to provide non-invasive pretty-printing of R source code while adhering to the tidyverse formatting rules. Support for custom style guides is planned.

You can install the package from CRAN:

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

You can style a simple character vector of code with ‘style_text()‘:

```r
ugly_code <- "a<-function( x){1+1}"
style_text(ugly_code)
#> a <- function(x) {
#>   1 + 1
#> }
```

But there are many other ways of using the styler package. You can learn more here and here.

3.5 Further Reading

- TidyVerse Style Guide
- Google R Style Guide

3.6 Python Coding Style

New projects must follow pep8 style and naming conventions. Line length exceptions are allowed, while pep8 recommends a line length of 79 characters, a line length of a maximum 120 characters is acceptable.

To enforce compliance, before committing, the code needs to be run through flake8 tool; violations should be addressed.

As a good practice code submitted for code review should be pep8 compliant.

A short startup guide to flake8, please refer to the official website for complete documentation.

flake8 installation:

```
python<version> -m pip install flake8
```

pep8-naming installation:

```
python<version> -m pip install pep8-naming
```

Pre commit git hook installation (run this in the repo directory):

```
flake8 --install-hook git
git config --bool flake8.strict true
```

Sample config file for flake8 - for Mac and Linux systems ‘ /.config/flake8‘. For Windows use ‘ ˙flake8‘.

```
[flake8]
max-line-length=120
```

Troubleshooting:

```
flake8 --verbose
```

Chapter 4

Profiling

AUTHOR: MIKKEL FRELTOFT KROGSHOLM

This chapter is largely based on the Profiling and benchmarking chapter in Hadley Wickhams Advanced R book.

Profiling is about making your code run as fast as needed by optimizing it to do just that. Optimizing code to make it run faster is an iterative process:

- Find the biggest bottleneck (the slowest part of your code)
- Try to eliminate it (you may not succeed but that's OK)
- Repeat until your code is "fast enough"

4.1 Measuring Performance

You can measure the performance of your code in both R and Python.

4.1.1 Profiling in R

R uses a fairly simple type called a sampling or statistical profiler. A sampling profiler stops the execution of code every few milliseconds and records which function is currently executing (along with which function called that function and so on).

You can use the profvis package to profile your R code:

```
{r, eval=FALSE}
library(profvis)
library(ggplot2)

profvis({
  g <- ggplot(diamonds, aes(carat, price)) + geom_point(size = 1, alpha = 0.2)
  print(g)
})
```

The ‘profvis()‘ call returns an htmlwidget, which by default opens a web browser.

4.1.2 Profiling in Python

MISSING

4.2 Improving Performance

Below are some language-agnostic tips on how to make your code run faster. They apply to both R and Python.

4.2.1 Are you using the right tool?

First, it makes sense to consider whether you are even using the right tool or if there is a better solution. Are you for instance reading in a lot of data and then doing operations on it that might as well be done in a database like *postgresSQL*? Or are you trying to do advanced NLP on a lot of text data, where some of the heavy lifting might be better suited for ‘elasticsearch‘? If you are in doubt, ask a fellow data scientist or data engineer if they have an idea for an alternative solution - it might just be what supercharges your code.

4.2.2 Has someone already solved the problem?

Once you’ve profiled your code, it’s natural to see what others have done. You are part of a large community, and it’s quite possible that someone has already tackled the same problem. If your bottleneck is a function in a package, it’s worth looking at other packages that do the same thing.

Otherwise, the challenge is describing your bottleneck in a way that helps you find related problems and solutions. Knowing the name of the problem or its synonyms will make this search much easier. But because you don't know what it is called, it's hard to search for it! By reading broadly about statistics and algorithms, you can build up your own knowledge base over time. Alternatively, ask others. Talk to your colleagues and brainstorm some possible names, then search on Google and Stack Overflow. It's often helpful to restrict your search to language (R or Python) related pages. With R for instance, when you use Google, try rseek. For Stack Overflow, restrict your search by including the R tag, [R] in your search.

Once you have found a solution that's fast enough, then consider sharing your solution with the R or Python community.

4.2.3 Parallelise

Parallelization uses multiple cores or machines in a cluster to work simultaneously on different parts of a problem. It doesn't reduce the computing time, but it saves your time because you're using more computing resources in parallel.

You can also use tools like Spark (Sparklyr for R and pySpark for Python) to distribute your computations. But if you need to run "pure" R or Python, below are some ways you can do that in parallel:

Parallelization in R

In R you can use the future package for parallelization. It aims at being a **Unified Parallel and Distributed Processing in R for Everyone**.

Below is an overview of the possibilities you have for parallelizing your code in the future:

Name	OSes	Description
synchronous:		_non-parallel:_
‘sequential‘	all	sequentially and in the current R process
‘transparent‘	all	as sequential w/ early signaling and w/out local (for debugging)
asynchronous:		_parallel_:
‘multiprocess‘	all	multi-core, if supported, otherwise multi-session
‘multisession‘	all	background R sessions (on current machine)
‘multicore‘	not Windows	forked R processes (on current machine)
’cluster’	all	external R sessions on current, local, and/or remote machines
’remote’	all	Simple access to remote R sessions

Parallelization in Python

MISSING

Chapter 5

Testing Best Practice

AUTHOR: HARSH JAIN

This chapter is about best practices for testing or Test-driven development (TDD).

It is important to note that Test-driven development (TDD) is not solely a testing technique, but rather part of a holistic design, development, and testing process. The basic idea of TDD is that instead of writing your code first and then writing tests after to ensure the code works, you write your tests first and then write the code that will get all the tests to pass. This is known as a **Test-First Approach**.

There are two generally accepted views on how and why you should practice TDD in your software development:

- The first view sees TDD as a technique for specifying your requirements and design before writing the actual program code
- The second view takes a more pragmatic approach and sees TDD as a technique that helps programmers write better code

Regardless of the view one takes, what TDD practitioners all agree on is that TDD will not only improve your code, but it will also improve the overall design and implementation of your software system. Below are two types of tests that are important for us:

1. *Unit Test*: Verifies a single logic about the unit of work
2. *Integration Test*: Verifies the connection and data flow between components

Testing Frameworks/ Libraries

- **Python**: unittest, pytest, pymock, coverage
- **R**: testthat, RUnit, svUnit

5.1 Test Structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) Setup, (2) Execution, (3) Validation, and (4) Cleanup.

- **Setup**: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- **Execution**: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- **Validation**: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT.
- **Cleanup**: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.

5.2 Naming Conventions

Naming conventions help organize tests better so that it is easier for developers to find what they're looking for. Another benefit is that many tools expect that these conventions are followed:

- **Separate the implementation from the test code**: It avoids accidentally packaging tests together with production binaries; many build tools expect tests to be in a certain source directory.
- **Place test classes in the same package as implementation**: It helps finding tests, knowing that tests are in the same package as the code they test helps finding them faster.

- **Name test classes in a similar fashion as classes they test**: It helps finding tests. One commonly used practice is to name tests the same as implementation classes with suffix ’Test’. If, for example, implementation class is UserController, test class should be UserControllerTest. Often, the number of lines in test classes is bigger than the number of lines in the corresponding implementation class. There can be many test methods for each implementation method. To help locate methods that are tested, test classes can be split. For example, if StringCalculator has methods createUser and deleteUser, there can be test classes StringCalculatorAddTest and StringCalculatorRemoveTest.

- **Use descriptive names for test methods**: It helps to understand the objective of tests. Using method names that describe tests is beneficial when trying to figure out why some tests failed or when the coverage should be increased with more tests. It should be clear what conditions are set before the test, what actions are performed and what is the expected outcome. There are many different ways to name test methods. Our preferred method is to name them using the **Given/When/Then**syntax used in BDD scenarios. **Given** describes (pre)conditions, **When** describes actions, and **Then** describes the expected outcome. If some test does not have preconditions (usually set using @Before and @BeforeClass annotations), Given can be skipped.

5.3 Processes

TDD processes are the core set of practices. Successful implementation of TDD depends on the practices described in this section:

- **Write the test before writing the implementation code**: This ensures that testable code is written; ensures that every line of code gets tests written for it.

- **Write the test before writing the implementation code**: This ensures that testable code is written; ensures that every line of code gets tests written for it.

- **Integrate running tests in build pipeline**: If your project has a continuous integration(Jenkins) implemented then ensure that as soon as you commit your changes to VCS (git) a build is triggered and runs all the test cases

5.4 Miscellaneous Do's

- Separate common set-up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test focused on only the results necessary to validate its test.
- Is very limited in scope - If the test fails, it's obvious where to look for the problem. Use a few Assert calls so that the offending code is obvious. It's important to only test one thing in a single test.
- Runs fast, runs fast, runs fast - If the tests are slow, they will not be run often.
- Clearly reveals its intention - Another developer can look at the test and understand what is expected of the production code.
- Runs and passes in isolation - If the tests require a special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my system" excuse doesn't work.
- Often uses stubs and mock objects - If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
- Design time-related tests to allow tolerance for execution in non-real-time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Test the coverage of your code base and aim for maximum coverage various tools and plugins can be used depending on an IDE and the programming language of choice.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.

5.5 Miscellaneous Dont's

- Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).

- Dependencies between test cases - A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.

- Testing precise execution behavior timing or performance. Building "all-knowing oracles". An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.

- Slow running tests.

Chapter 6

Branding Your Code

It is important for your data science product to be used. If it is not used, then you might as well not have built it. This chapter is a little out of the ordinary, but it serves to get you thinking about how to maximize the use of the product you are now coding.

This chapter builds on a blog post called "Marketing for Data Science: A 7 Step 'Go-to-Market' Plan for Your Next Data Product" published on medium in March of 2018. The blog post lists seven steps that will boost the launch of your data science product:

- Naming/branding
- Documentation
- Champion identification
- Timing
- Outreach
- Demoing
- Tracking

Let's go through them one by one.

6.1 Naming your product

The blog post suggests getting an MBA name: *Memorable, Brandable, and Available.*

6.1.1 Memorable

Memorable names are often:

- **Pronounceable**: They start with a sharp sound and roll off the tongue. Research on English speakers suggests names with initial plosive consonants (p, t, k).
- **Plain**: They frequently repurpose common words (e.g. Apple), which help you combine rich mental images to your product. Be aware that discoverability through search may be limited when using common words. Slightly modifying the word can help overcome this (e.g. Lyft) as long as it is memorable.
- **Produced**: They can even be entirely new. Making up a new word is also a strategy (e.g. Google, Intel, Sony, or Garmin), but this requires substantially more initial seeding to establish the name. This may not be in line with the audience and timeframe of a/an (internal) data product launch.

6.1.2 Brandable

You want your new name to consistently represent the identity of the data product and reflect an overall positive attitude towards it. This way it can be incorporated seamlessly into the tool and documentation.

Available

Make sure no one else has called their data product the same thing. A quick google search should tell you whether that is the case.

6.2 Document Your Product

We have already covered documenting code in other places in this book, but the blog post also suggests going deep into detail about what problem your code solves. The reason for this is that you want your code/new app to stand alone - to be understandable by others than you.

In code documentation we want you to make your code understandable for other coders. In the description of your product, we want you to make its use understandable for future/potential users.

When writing your documentation, identify:

- The main problem your data product is solving
- Key features and how they solve the problem
- Key definitions
- Key technical aspects that need to be explained

Also, include your MBA name within the documentation to further establish the brand.

6.3 Identify Power Users

Nothing spreads your code more than enthusiastic users - ’so-called’ champions.

Seek out people who are affected by the problem you are trying to solve and share your work with them. Also, look to your own team members who have participated in the build or know your work. These champions can recommend your work to others who would also appreciate the solution.

These champions will both help spread your code by word-of-mouth and they will be able to answer user questions if you yourself are not around.

6.4 Time Your Launch

Before each launch, consider the current business environment and time your launch accordingly. The moment you have finished working on your data product is not necessarily the best time to launch it.

For example, a product team may be in the middle of fixing a major bug and not ready for a new idea. Conversely, an upcoming related communication activity (e.g., blog post) could be an opportune time for a release with cross-promotion.

Look at other recent data products: When were they released and how were they received? Stakeholders can feel inundated with too many new dashboards and models, and this may even contribute to ‘analysis paralysis.’

6.5 Cater to Your Audience

If your champions are not happy, your product will wither and die. Developing positive working relationships with your champions and users is important for the early and long-term success of your data product.

Identify and reach your audience - those who will be using what you have made and can benefit from it. With this target audience in mind, comment on tickets, post on Slack, chat, send emails to relevant groups, or go directly to talk to your audience.

Use your audience's preferred channels to communicate development progress, releases, and feedback. Establishing this communication will build early confidence in your data product. As iteration requests come in, you will have the opportunity to build this confidence with thoughtful acknowledgment of requests.

6.6 Demo It

Only you can see the picture in your mind of how something works. Demoing is a powerful way to communicate what your new data product does. A great way to do this is by getting a minimum viable data product, a prototype, out early to your champions.

Examples include creating a working application with minimal data, sketching a mockup of a dashboard, or taking screenshots.

These demos can be the starting point for recruiting the champions mentioned above and it is a great source of feedback for improving your code and app.

6.7 Track It

You may love the theoretical foundation and implementation of your data product, but ultimately the success of a data product comes down to the user.

Long-term marketing and retaining users depends on how much you can ensure reliability. Reliability is key to building your data product's brand, your reputation, and your technical credibility. This affects the marketing for your other current and future data products as well. It's worth noting that this does not mean perfection — it often just means dealing with problems quickly, fully, and transparently.

Monitor key metrics of your data product to see how it's working and what its impact is. Actively seek and be responsive to feedback. Evaluate if your data product is achieving its intended objectives and determine if features can be improved to better suit your audience.

If you are not achieving impact or the tool is not being used, revisit your initial assumptions about the problem you thought you were solving. Then, talk to your users (and non-users) about what might not be working. Be willing to destroy and start again, and create something even better with a new perspective. The initiative to iterate and improve your data product tools requires persistence but will raise the quality of your data products and enhance the rest of your marketing efforts.

You can use analytics tools like Google Analytics or the open source tool Matomo for this.

Chapter 7

Reproducible Data Science

This chapter is dedicated to the concept of reproducibility - that means creating code that can be deployed and run anywhere while at the same time giving the same results.

This chapter is opinionated. It promotes a certain way of doing things, that is not the usual way of doing things for a data scientist.

When you are done with this chapter you will have an idea of how to setup an environment that is geared towards reproducible data science based on the concepts of containerization and docker.

I recommend you do this for EVERY larger project you work on. You can spin up as many containers as you like so make it a habit of doing that when you are starting a new project. This way everything is documentable from the get-go.

7.1 What is Docker?

Docker is a computer program that performs operating-system-level virtualization also known as containerization. It is developed by Docker, Inc. Docker uses resource isolation to allow independent "containers" to run within a single OS, avoiding the overhead of starting and maintaining virtual machines (VMs).

It creates the possibility to set up isolated and reproducible environments for your data science work that leverage the host systems hardware but do not interfere with other software programs on the host. This is the concept of using docker images and containers.

A docker image is the boilerplate or template for a container. It can be created by using a Dockerfile that describes how the image is being built. This image, once built/compiled, can then be used to start new containers.

That means that you can write a Dockerfile that will create the exact environment that you need for your analysis. Once built, you can use this image every time you need to run the given analysis. The processes will be totally isolated and the software you run or update will not have any effect outside of the given container.

You can also share this Dockerfile with colleagues, check it in to git, and compile and store the docker image in a docker repository like Dockerhub. Besides sharing the same data science and having the exact same infrastructure for a project, you can also deploy the containers to external servers when your analysis needs to go into full production - and you can do that with minimum worry since you know it will work because everything is containerized.

7.2 A Dockerized Workflow

This section will describe how you can set up a dockerized workflow. It will focus on running R in a docker container, but the concepts are easily transferable to Python or other languages.

7.2.1 The Docker Image

Prebuild Image

To launch a docker container you need a docker image. This image can either be found on Dockerhub (or another docker repository) or created from a recipe of your own: a so-called dockerfile.

The Rocker Project on Dockerhub hosts a whole bunch of ready-to-go docker images for R. There are images for everything from base R to Rstudio with a whole lot of packages preinstalled.

For instance, if you want to get started with Rstudio running R version 3.5.0 with the Tidyverse already installed then you can get up and running, with the following command:

```
docker run -d --name rstudio -p 7878:8787 rocker/tidyverse:3.5.0
```

If the image is not already available on your local computer then it will download it when you run the command.

This is what the command means:

- ‘docker‘: tells the shell that we are executing a docker command
- ‘run‘: will start the image

- ‘-d‘: will start the image in the background (meaning you can use your terminal after the command) and will not display any logs.
- ‘–name rstudio‘: gives the container the name rstudio
- ‘-p 8787:8787‘: will map the internal port 8787 (Rstudio Servers default port) to the external port 7878. Meaning you can access it at localhost:7878
- ‘rocker/tidyverse:3.5.0‘: the image to be used is the tidyverse image from rocker version 3.5.0

Custom Image

The above is all well and gut if the image you are using has all the dependencies you need. If so, then it is a reproducible environment. If not, then you need to create your own image - for instance, if you are installing extra packages.

You create a custom image by building your own dockerfile. Below I am creating a dockerfile that bases itself on the ‘rocker/tidyverse:3.5.0‘ from above but adds two new extra packages. Please notice that the file must be named **Dockerfile**.

```
FROM rocker/tidyverse:3.5.0
# Install statsDK and realestateDK packages from CRAN
RUN R -e "install.packages(c('statsDK', 'realestateDK'), repos =
    'https://cloud.r-project.org/')"
```

Next, you need to build this dockerfile into an image. You do this by running this command in the folder where the ‘Dockerfile‘ resides:

```
docker build -t my_custom_image:v1
```

This will create an image called ‘my_custom_image‘ with ‘v1‘ as the version number. You can change that to whatever you want.

7.2.2 Docker Compose

Compose is a tool for defining and running multi-container Docker applications. It is also a way to create a "recipe" for starting up your data science environment, even if you only have one container.

Below are two Docker compose scripts for each of the two approaches described above.

Docker compose with prebuild image

The compose script below runs a "service" called rstudio. This service starts up a container called ‘rstudio‘ based on the ‘rocker/tidyverse:3.5.0‘ image and exposes the 8787 port to 7878 just like we did above.

```
version: "3"

services:

  rstudio:
    container_name: rstudio
    image: rocker/tidyverse:3.5.0
    ports:
      - 7878:8787
```

Docker compose with custom Built Image

The compose script below runs a "service" called rstudio. This service starts up a container called ‘rstudio‘ but unlike the compose script above, this one builds the image from the Dockerfile in the same folder (hence the .). It then exposes the 8787 port to 7878 just like the other one.

```
version: "3"

services:

  rstudio:
    container_name: rstudio
    build: .
    ports:
      - 7878:8787
```

7.2.3 Commit to Git

With your dockerfiles and docker compose script, you now have an entire description of your environment. You can commit these files to git along with the scripts you used in your analysis and a readme file that describes how to set up the environment and run the analysis.

With this, you and everyone else should be able to reproduce your analysis anywhere at any time.

7.3 Further Reading

An Introduction to Docker for Reproducible Research, with Examples from the R Environment

7.4 Udemy Resources

Docker Mastery: The Complete Toolset From a Docker Captain

Chapter 8

Data Science Pipelines

- openfaas
- streamsets

Chapter 9

Defensive Programming

This chapter is more or less copied from Defensive Programming.

9.1 What is There to Defend Against?

Functions sometimes fail. It's inevitable. Defensive programming is not about preventing your functions from failing; rather, it's about ensuring that any failures are quick to surface, hard to miss, and easy to understand.

Defend your code from three threats:

- **Unanticipated user inputs** are usually function arguments that don't conform to your function's assumptions. For example, a user might pass you a vector where you expected a scalar, a data frame that lacks an essential column, or 'true' instead of 'TRUE'. For scientific programming, some of the most important unanticipated inputs will be formatted correctly but wrong in more subtle ways. A user might supply discharge data in cubic feet per second while your function expects cubic meters per second, or they might request 'hyperbolic' when the only available options are 'linear"' and 'polynomial'.

- **Unanticipated results** can come from functions that your function uses. Your function might call sapply expecting a vector, but on certain datasets, the output could be a list instead. And 'diff(as.POSIXct(c('2014-03-01','2014-04-01')))' will return a time difference of '31 days' if your computer is in Arizona but '30.95833 days' if it's in Colorado.

- **Unreliable processes** usually involve the internet. Does your function download a file or send an email? These processes are prone to random failures. Although you'll probably devote more keystrokes to defending against unanticipated inputs and results, unreliable processes can fail in especially frustrating and unreproducible ways.

9.2 Principles of Defensive Programming

The key to defensive programming is to know what your function requires and to make formal assertions about those requirements. These assertions take the form of code-based tests of user inputs and function outputs, followed by warnings, error messages, or preventive actions if something is about to go wrong. When your functions can't succeed, try to make them fail.

- Conspicuously - The worst failure is a silent one
- Fast - If a function is going to fail eventually, it might as well fail right now
- Informatively - Provide messages and context that help the user understand and/or correct the problem

9.2.1 Fail Conspicuously

R provides several ways to communicate with the user when things are not going according to plan. The three methods you'll use most often are:

- Errors, produced with 'stop()', are best when your function can't reasonably proceed. For example, 'weighted.mean(1:3, 4:5)' returns an error because the values and weights need to have the same lengths.
- Warnings, produced with 'warning()', are best when your function can mostly achieve what was asked, but the output might not be fully what the user expected. For example, 'log(-3:3)' gives the warning 'NaNs produced' to indicate that you have asked for the (impossible) log of negative values and so will see 'NaN' in those positions in the output vector.
- Messages, produced with 'message()', are best for giving the user status updates as a long-running function makes progress, or for telling the user about a decision your function has made for the user. For example, 'dplyr::full_join(data.frame(x=1, y=2), data.frame(y=2, z=3))' guesses that it should join on the '"y"' column and tells you what it guessed.

You may be tempted to use 'print()' or 'cat()' to keep users informed, but it's best to reserve these functions for standard and expected outputs such as model summaries or reports. Errors, messages, and warnings (collectively called 'conditions') have special features that make them better for handling the unexpected. These include:

- RStudio prints conditions in a bright color to attract the user's attention. They are appropriately conspicuous.
- You can call 'traceback()' on any condition to find out where it originated. This can be very helpful for debugging.
- You can control which conditions you see: 'suppressWarnings()' and 'suppressMessages()' hide warnings and messages from specific function calls, and 'options(warn = 2)' treat warnings like errors (again, helpful for debugging).
- The 'tryCatch()' function automatically recognizes conditions and lets you choose how to handle them. You can add information to an error message, convert a warning to an error or a message, ignore specific warnings, and even retry the failed operation. See the Retries section below for an example.

9.2.2 Fail Fast

It's almost always better for a function to fail right away than to wait and keep trying. Nobody wants to wait through a long computation only to find out that the starting conditions were unacceptable. Similarly, if your function produces 3 output files but it sometimes fails after producing just the first file, the user is left with messy partial outputs. To avoid awkward situations like these, check the user inputs early in your function and check the outputs of subroutines as soon as they have been run.

The simplest tests are 'if' statements combined with 'stop()', 'warning()', or 'message()'. For example:

```r
cool_computation <- function(dat, method) {
  if(!is.data.frame(dat) || any(names(dat) != c('x', 'y'))) {
    stop("dat must be a data.frame with columns 'x' and 'y' for this cool
        computation to continue")
  }
  if(!(length(method) == 1 && method %in% 1:3)) {
    stop("method must be a single integer with a value of 1, 2, or 3")
  }
  # cool part goes here...
}
cool_computation(data.frame(x=1, y=2), method=5)

## Error in cool_computation(data.frame(x = 1, y = 2), method = 5): method must
   be a single integer with a value of 1, 2, or 3
```

The ‘if-stop‘ combination requires you to write your own error message. You can sometimes save typing with the ‘stopifnot()‘ function:

```r
cool_computation_2 <- function(dat, method) {
  yvals <- sapply(seq_len(nrow(dat)), function(i) {
    dat[i,'y']
  })
  stopifnot(!is.list(yvals))
  stopifnot(is.numeric(yvals))
  return(yvals)
}
cool_computation_2(data.frame(x=1))

## Error: !is.list(yvals) is not TRUE
```

R also provides helpful built-in error handling for some common input problems. In these cases, you can probably rely on R to catch the problem and generate a useful error message for you:

- If a user fails to supply an argument ‘x‘ that has no default, then as soon as your function tries to use ‘x‘, the user will see ‘argument "x" is missing, with no default‘. If ‘x‘ isn't used until late in your function and you want to check for ‘x‘ right away, you can get a ‘TRUE/FALSE‘ from ‘missing(x)‘ and then throw your own error.

- If a user supplies an extra argument ‘y=3‘ that isn't listed in the function declaration, the user will see ‘unused argument (y = 3)‘.

- For character arguments, the ‘match.arg()‘ function can check the user's input against a pre-defined list of options. ‘match.arg()‘ is especially nice because it helps with the Don't Repeat Yourself (DRY) principle: You only need to type a vector of options once, in the function definition. Then the vector will appear in the function help file, will get picked up automatically by ‘match.arg()‘, and will appear in the error message if the user's selection isn't one of the valid options. (‘match.arg‘ has several other nifty features - check them out at ‘?match.arg‘!)

```
apply_method <- function(method=c('linear','polynomial')) {
  method <- match.arg(method)
  return(method)
}
apply_method('linear') # normal functionality

## [1] "linear"

apply_method('hyperbolic') # useful error message

## Error in match.arg(method): 'arg' should be one of "linear", "polynomial"

### The exception to fast failure: Retries
```

Fast failure is usually the best option, but there are cases where retries are better. These arise most often with internet data transfers, which are the flakiest thing we do with computers these days. For other failures, we can usually rely on the user to fix a problem by supplying different inputs, but in the case of internet transfers our function can sometimes solve the problem just by trying again. If using the httr package, you can identify a problem using a built-in test 'stop_for_status()', which throws an error if the transfer was unsuccessful:

```
library(httr)
flaky_GET <- function() {
  good_get <- GET("http://httpbin.org/get")
  stop_for_status(good_get)
  return(good_get)
}
```

For this demonstration, let's also invent an unreliable function that pretends to do an internet transfer but fails even more often:

```
flaky_process <- function() {
  success <- runif(1, min=0, max=1) > 0.7
  if(!success) stop("darn! this 'internet transfer' failed")
  return("this is my successful result")
}
```

To add in retries, wrap the call to your unreliable process in a call to 'tryCatch', then put it in a loop that keeps iterating until 'flaky_process()' returns successfully or we run out of 'attempt's. The 'error' argument to 'tryCatch' is a function you define to control what happens if 'expr' returns an error; in this case, we simply return the error as an object to be inspected on the following line. If that inspection shows that the output is not an 'error' object, we conclude that the attempt was successful and we exit the ‘for‘ loop immediately (without doing any more iterations) with ‘break‘.

```
set.seed(4433)
for(attempt in 1:10) {
  message("attempt number ", attempt)
  output <- tryCatch(
    expr={ flaky_process() },
    error=function(e) { return(e) }
  )
  if(!is(output, "error")) {
    message("success! exiting the retry loop now")
    break
  }
}

## attempt number 1

## attempt number 2

## attempt number 3

## attempt number 4

## attempt number 5

## attempt number 6

## attempt number 7

## attempt number 8

## success! exiting the retry loop now

output

## [1] "this is my successful result"
```

9.2.3 Fail Informatively

When your function is about to fail and retries won’t help, the most important thing you can do is communicate clearly to the user about what went wrong. Your time is well spent on crafting informative error messages that explain what's wrong and what the user can do right now to fix the problem. Consider these alternatives:

```
quick_and_dirty <- function(dat, status) {
  suggestion <- switch(
    status,
    "red sky at night"="sailors, delight!",
    "red sky at morn"="sailors, be warned..."
  )
  return(sprintf("On %s, %s", dat$date, suggestion))
}
quick_and_dirty(data.frame(Date=as.Date("2017-06-05")), status="Red Sky at
    Night")

## character(0)
thoughtful_and_sweet <- function(dat, status=c("red sky at night", "red sky at
    morn")) {
  if(!('date' %in% names(dat))) {
    stop("'dat' should include a column for 'date'")
  }
  status <- match.arg(status)
  suggestion <- switch(
    status,
    "red sky at night"="sailors, delight!",
    "red sky at morn"="sailors, be warned..."
  )
  return(sprintf("On %s, %s", dat$date, suggestion))
}
thoughtful_and_sweet(data.frame(Date=as.Date("2017-06-07")), status="Red Sky at
    Night")

## Error in thoughtful_and_sweet(data.frame(Date = as.Date("2017-06-07")), :
    'dat' should include a column for 'date'
thoughtful_and_sweet(data.frame(date=as.Date("2017-06-07")), status="Red Sky at
    Night")

## Error in match.arg(status): 'arg' should be one of "red sky at night", "red
    sky at morn"
thoughtful_and_sweet(data.frame(date=as.Date("2017-06-07")), status="red sky at
    night")

## [1] "On 2017-06-07, sailors, delight!"
```

As a user, which function would you rather encounter?

You can do all the checking and communication that's required with 'if()' and 'stop()' alone. But if you're passionate about writing less code while still producing informative error messages, check out the checkmate, assertive, assertr, and assertthat packages. Each of these packages provides a slightly different approach to a common problem. Most of them provide:

- Pre-packaged tests for common requirements, e.g., whether a variable falls within some range of values or dates, whether a file has some specific extension, or whether a list has some specific length.
- Nicer default messages than 'stop' (which has no defaults) and 'stopifnot' (which just prints out the code of the test).
- A choice of what action to take when a test is not passed. Most of these packages let you choose among throwing an error, receiving a 'TRUE' or 'FALSE', receiving a character string describing the test failure, or defining your own action.

assertive provides a huge number of pre-defined tests; assertthat is concise and quick to learn; assertr works elegantly with piping workflows; checkmate is optimized for computational speed. If one of these packages sounds like a good fit for your needs, have at it!

9.3 Balancing Defensiveness with Efficiency

Defensive programming is an art. Not only does it require great imagination to think up all the crazy inputs that might enter your function, but it also requires your judgment to know how many tests are enough. When you're deciding which tests to create in your functions, consider the following:

- What are the most likely forms of bad input to this function?
- What might a confused user try, and which tests could save users from nonsensical or misleading outputs?
- Which forms of bad input would cause the most catastrophic, slow, or frustrating problems?
- What values could a code chunk produce that would cause the biggest problems later in the function (or after the function returns)?

- Will users be calling this function directly, or can you control the range of inputs by keeping this function internal to your package?

It is OK not to test for every possible edge case - in fact, you cannot. But you can and should test strategically for the cases with the highest probabilities and the highest risks.

Chapter 10

Ethics

In Spider-Man, Peter Parker is told that "With great powers come great responsibilities." This is also true for data science superheroes. As the algorithms, we develop and use get more powerful and capable and their use becomes more pervasive, we have to think about how and for what we are using these new powers.

This dilemma is also dawning on national governments and large corporations. Some of them are now starting to write up and enforce guiding principles on how and for what AI should be used. See for instance Google's AI principles here.

This chapter looks into some of the aspects of why it is important to be aware of when designing and using AI.

10.1 Unfair Bias

Is the AI you are building reinforcing an old unfair bias or maybe even creating a new one? Our societies and cultures can have unfair biases regarding gender, race, sexuality, or other traits that our AI might unintentionally inherit. If we are not on the lookout for this, we might create an AI that recreates or even reinforces these biases.

10.2 What is in the Model?

Our models need to be as explainable as possible. Especially if they are used to make decisions about real people's lives and livelihoods. We need to be able to look under the hood of the model. Both to see what is going on, but also so we can explain it to people who are affected by the model. Making a model accountable and interpretable to people not only makes us build better models (because we understand them better) but also creates trust in the models we use (because the laypersons may better understand them).

There are many tools out there now that aim at this (such as LIME and more are coming.

10.3 Privacy by Design

What data are you using in your model and are you allowed to use it? These questions are especially relevant since GDPR has come into full effect. But they are also relevant from an ethical perspective. If we are using personal data then it is important that the people know what their data is being used for and that they have given their consent.

If we are handling sensitive personal information (like medical history) it is especially important to be vigilant about privacy. Even if the user has given consent it is still very important to keep that data from spreading or leaking. A leak of someone's very private information can lead to stigmatization and discrimination of that person.

Chapter 11

Final Words

We have finished a nice book. For further details, visit here.