R Notes

John Doe

2025-04-04

Contents

Chapter 1

About

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports; for example, a math equation $a^2 + b^2 = c^2$.

1.1 Usage

Each **bookdown** chapter is an .Rmd file, and each .Rmd file can contain one (and only one) chapter. A chapter *must* start with a first-level heading: `# A good chapter`, and can contain one (and only one) first-level heading.

Use second-level and higher headings within chapters like: `## A short section` or `### An even shorter section`.

The `index.Rmd` file is required, and is also your first book chapter. It will be the homepage when you render the book.

1.2 Render book

You can render the HTML version of this example book without changing anything:

1. Find the **Build** pane in the RStudio IDE, and
2. Click on **Build Book**, then select your output format, or select "All formats" if you'd like to use multiple formats from the same book source files.

Or build the book from the R console:

```
bookdown::render_book()
```

To render this example to PDF as a `bookdown::pdf_book`, you'll need to install XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.org/tinytex/.

1.3 Preview book

As you work, you may start a local server to live preview this HTML book. This preview will update as you edit the book when you save individual .Rmd files. You can start the server in a work session by using the RStudio add-in "Preview book", or from the R console:

```
bookdown::serve_book()
```

Chapter 2

Rstudio

Rstudio shortcuts

Command Palette: + +P, all shortcuts can be accessed via the Command Palette.

keyboard combination	function
opt + _	insert assignment operator `<-`
ESC or ctrl + C	exit `+` prompt
ctrl + shift + m	add pipe operator "%>%"
ctrl + `[/]`	indent or unindent
cmd + D	delete one row
cmd + 1	move cursor to console window
cmd + 2	move cursor to editor window
ctrl + shift + S	add 80 hyphens `---` to signal a new chapter (Addin)
ctrl + shift + =	add 80 equals `===` to signal a new Chapter (Addin)
shift + cmd +N	new R script
cmd + ↑ / ↓	in console, get a list of command history
shift + ↑ / ↓	select one line up/down
fn + F2	`view()` an object, don't select the object
cmd + shift + 1	activate X11() window
ctrl (+ shift) + tab	next (last) tab in scriptor (this applies to all apps); hit ctrl first, then shift if necessary, last tab

Source

keyboard combination	function
cmd + return	Run current line/selection
opt + return	Run current line/selection (retain cursor position)

`Rmd` related

keyboard combination	function
cmd + shift + K	**Knit** rmd
cmd + opt + C	run current code chunk in `Rmd`
cmd + opt + I	insert code chunks in `Rmd`, i.e., ```` ```{r} ```` and ```` ``` ````

Q: How to print output in console rather than inline in Rmd?

A: Choose the gear in the editor toolbar and choose "Chunk Output in Console".

Q: How to insert Emojis in Rmd?

A: There are several options:

- You can type directly a lot of Emojis, such as and . Try this first, if it doesn't show properly, then try the following solutions.
  - If the emoji can show in the script, then you can use it directly.
- Using a html tag, e.g., `<span>` `</span>` will show like this

  This seems to be the most straightforward solution to me.

  Note that the emoji won't disply correctly in your Rmd file, but when you render the Rmd and deploy to html pages, the emoji will show properly.
- Using Hexadecimal code. (You need to look up the code somewhere, which is a hassle.)

  We can add emojis to an HTML document by using their hexadecimal code. These code starts with `&#x` and ends with `;` to specify browser that these are hexadecimal codes. For example,

  ```
  <p>Smily face <span>&#x1F600;</span> </p>
  ```

  will give you

  Smily face

  Go to this site: https://emojipedia.org/emoji/

Grab the **codepoint** for the emoji you want (e.g., `U+1F600` for grinning face)

Replace `U+` with `&#x` so it becomes `&#x1F600`, and add a semicolon `;` at the end.

Finally, enclose that into an html tag, e.g., `<span>`.

- With RStudio Visual mode. (You need to change mode back and forth.)

  First change to the Visual mode. To insert an emoji, you can use either the `Insert` menu or the requisite markdown shortcut plus auto-complete:

  I am personally NOT a fan of Visual Mode because it changes your source code silently …

Set working directory

```
# get the dir name of the current script
dir_folder <- dirname(rstudioapi::getSourceEditorContext()$path)
setwd(dir_folder) # set as working dir
```

RStudio projects are associated with R working directories. You can create an RStudio project:

- In a brand new directory
- In an existing directory where you already have R code and data
- By cloning a version control (Git or Subversion) repository

Why using R projects:

1. I don't need to use `setwd` at the start of each script, and if I move the base project folder it will still work.
2. I have a personal package with a custom project, which creates my folders just the way I like them. This makes it so that the basic locations for data, outputs and analysis is the same across my work.

Double-click on a `.Rproj` file to open a fresh instance of RStudio, with the working directory and file browser pointed at the project folder.

Q: What is an **R session**? And when do I use it?

A: Multiple concurrent sessions can be useful when you want to:

- Run multiple analyses in parallel
- Keep multiple sessions open indefinitely
- Participate in one or more shared projects

Launch a new project-less RStudio session

```
# run in console
rstudioapi::terminalExecute("open -n /Applications/RStudio.app", show = FALSE)
```

`-n` Open a new instance of the application(s) even if one is already running.

`rstudioapi::terminalExecute(command, workingDir = NULL, env = character(), show = TRUE)` tells R to run the system command in quotes.

- `command` System command to be invoked, as a character string.
- `workingDir` Working directory for command
- `env` Vector of name=value strings to set environment variables
- `show` If FALSE, terminal won't be brought to front

The `rstudioapi` package provides an interface for interacting with the RStudio IDE with R code. Using`rstudioapi`, you can:

- Examine, manipulate, and save the contents of documents currently open in RStudio,
- Create, open, or re-open RStudio projects,
- Prompt the user with different kinds of dialogs (e.g. for selecting a file or folder, or requesting a password from the user),
- Interact with RStudio terminals,
- Interact with the R session associated with the current RStudio instance.

Set up Development Tools

https://cran.r-project.org/bin/macosx/tools/

- install Xcode command line tools

  ```
  sudo xcode-select --install
  ```

- install GNU Fortran compiler

  Using **Apple silicon** (aka arm64, aarch64, M1) Macs Fortran compiler

- Go to https://www.xquartz.org/, download the .dmg and run the installer.
- Verify that build tools are installed and available by opening an R console and running

  ```
  install.packages("pkgbuild")
  pkgbuild::check_build_tools()
  ```

Insert Code Session

To insert a new code section you can use the **Code** -> **Insert Section** command. Alternatively, any comment line which includes at least four trailing dashes (`-`), equal signs (`=`), or pound signs (`#`) automatically creates a code section.

Define your own shortcuts

https://www.statworx.com/ch/blog/defining-your-own-shortcut-in-rstudio/

https://www.r-bloggers.com/2020/03/defining-your-own-shortcut-in-rstudio/

Install the shortcut packages.

Add code session separators, `---` or `===`.

```
install.packages(
    # same path as above
  "~/Downloads/shoRtcut_0.1.0.tar.gz",
  # indicate it is a local file
  repos = NULL)
install.packages(
    # same path as above
  "~/Downloads/shoRtcut2_0.1.0.tar.gz",
  # indicate it is a local file
  repos = NULL)
```

Now go to Tools > Modify Keyboard Shortcuts and search for "dashes". Here you can define the keyboard combination by clicking inside the empty Shortcut field and pressing the desired key-combination on your keyboard. Click Apply, and that's it!

2.1 Dark Theme

https://community.rstudio.com/t/fvaleature-req-word-background-highlight-color-in-find-and-spellcheck/18578/3

https://rstudio.github.io/rstudio-extensions/rstudio-theme-creation.html

https://docs.posit.co/ide/user/ide/guide/ui/appearance.html#creating-custom-themes-for-rstudio

`.ace_marker-layer .ace_selection` Changes the color and style of the highlighting for the currently selected line or block of lines.

`.ace_marker-layer .ace_bracket` Changes the color and style of the highlighting on matching brackets.

Recommended highlight color: `rgba(255, 0, 0, 0.47)`

`RStudio` editor theme directory on Mac:

right click `RStudio.app`, "Show Package Contents" to navigate to the application folder.

`/Applications/RStudio.app/Contents/Resources/resources/themes/ambiance.rstheme`

Custom theme (user-defined) folder:

- `~/.config/rstudio/themes/idle_fingers_2.rstheme` on mac
- viridis-theme

```
/* yaml tag */
.ace_meta.ace_tag {
  color: #2499DA;
}
/* quoted by $...$ and code chunk options */
.ace_support.ace_function {
  color: #55C667;
}
```

2.2 Update R

Q: How to tell which version of R you are running?
A: In the R terminal, type `R.version`.

The key thing to be aware of is that when you update R, if you just download the latest version from the website, you will lose all your packages!

The easiest way to update R and not cause yourself a huge headache is to use the `installr` package. When you use the `updateR()` function, a series of dialogue boxes will appear. These should be fairly self-explanatory but there is a full step-by-step guide available for how to use `installr`, the important bit is to select "Yes" when it asked if you would like to copy your packages from the older version of R.

```
# Install the installr package
install.packages("installr")

# Load installr
library(installr)

# Run the update function
updateR()
```

2.3 Packages Management

Load packages

Q: What is the difference btw `library(package)` and `require(package)`?
A:

- `library(package)` returns an error if the package doesn't exist.
- `require(package)` returns `FALSE` if the package is not found and `TRUE` if the packages is loaded. `require` is designed for use inside other functions, such as using the value it returns in some error checking loop, as it outputs a warning and continues if the package is not found.

Q: How to reload a package after updating?
A: Call `detach(package:pkg, unload = TRUE)` or `unloadNamespace` first, then use `library(pkg)` to reload. If you use `library` on a package whose namespace is loaded, it attaches the exports of the already loaded namespace. So detaching and re-attaching a package may not refresh some or all components of the package, and is inadvisable. The most reliable way to completely detach a package is to restart R.

For example, if we want to detach `ggplot2` package, we can use

```
detach(package:ggplot2, unload=TRUE)
```

`requireNamespace` can be used to *test* if a package is installed and loadable because it comes back with either `TRUE` (if found the pkg) or `FALSE` (if failed to find the pkg).

```
> !requireNamespace("ggplot2")
[1] FALSE
> !requireNamespace("ggplot3")
Loading required namespace: ggplot3
Failed with error:  'there is no package called 'ggplot3''
[1] TRUE
```

To see whether need to install some packages:

```
if (!requireNamespace("devtools")) install.packages("devtools")
```

Alternatively,

```
# short command
"ggplot2" %in% installed.packages()
# full command
"ggplot2" %in% rownames(installed.packages())
```

`installed.packages()` Finds details of all packages installed in the specified library path `lib.loc`. Returns a matrix of package names, library paths and version numbers.

```
> installed.packages() %>% class()
[1] "matrix" "array"

> installed.packages() %>% str()
 chr [1:355, 1:16] "abind" "alphavantager" "anytime" "askpass" "assertthat" "backports" "base" ..
 - attr(*, "dimnames")=List of 2
```

```
  ..$ : chr [1:355] "abind" "alphavantager" "anytime" "askpass" ...
  ..$ : chr [1:16] "Package" "LibPath" "Version" "Priority" ...
```

The following code can be used to load packages for your project and set up the working environment.

```
# load the pkg, if not found, install then load
require(dplyr) || {install.packages("dplyr"); require(dplyr)}
require(odbc) || {install.packages("odbc"); require(odbc)}
require(DBI) || {install.packages("DBI"); require(DBI)}
```

If using `library()`, will return error if some package is not installed and interrupt the program.

If it is a list of packages you want to check, use `lapply` to loop through all packages.

```
## First specify the packages of interest
packages = c("MASS", "nlme")

## Now load or install&load all
package.check <- lapply(
  packages,
  FUN = function(x) {
    if (!require(x, character.only = TRUE)) {
      install.packages(x, dependencies = TRUE)
      library(x, character.only = TRUE)
    }
  }
)
```

You can then use `search()` to determine whether all the packages have loaded.

```
search()
 [1] ".GlobalEnv"        "package:nlme"      "package:MASS"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:datasets"  "renv:shims"        "package:utils"
[10] "package:methods"   "Autoloads"         "package:base"
```

Install R packages from source

```
# From local tarball
install.packages(
  # indicate path of the package source file
  "~/Documents/R/UserPackages/shoRtcut2_0.1.0.tar.gz",
  # indicate it is a local file
  repos = NULL)
```

```
# From github
install.packages("Rcpp", repos="https://rcppcore.github.io/drat")
```

Install from GitHub

```
devtools::install_github(repo, ref="HEAD", subdir = NULL)
```

- `repo` repository address in the format `username/repo[/subdir][@ref|#pull]`. Alternatively, you can specify `subdir` and/or `ref` using the respective parameters. If both are specified, the values in `repo` take precedence.
- `ref` Desired git reference. Could be a commit, tag, or branch name, or a call to `github_pull()` or `github_release()`. Defaults to `"HEAD"`, which means the default branch on GitHub and for git remotes.

Ex

```
# install version 3.5.1
install_github("tidyverse/ggplot2", ref="ggplot2 3.5.1")
```

Check installed packages

```
# print all installed packages
rownames(installed.packages())
# check if `ggplot2` is installed
"ggplot2" %in% rownames(installed.packages())
```

Check package version

```
packageVersion("ggplot2") # check package version
```

Q: How do I know if I have the latest version?
A: You can go to GitHub repo to check release notes. You will find the latest version of packages there.

Update packages

- Update an individual package
  - Using `install.packages`

    ```
    install.packages("ggplot2") # update one specific package
    ```

  - Using `update.packages`

    ```
    update.packages(oldPkgs = "ggplot2")
    ```

    Note that you need to specify `oldPkgs` explicily as it is a named argument.

- Update ALL outdated packages

```
## update all installed packages in a stated library location, default to `.libPat
update.packages(lib.loc = .libPaths(), ask = TRUE)
```

`update.packages` updates ALL outdated packages in a stated library location. That library location is given by the first argument (if not supplied it works on all known library locations for the current R session).
It will ask you for every package if you want to update.
To just say `yes` to everything, use `ask = FAlSE`.

```
update.packages(ask = FALSE)
```

Unfortunately this won't update packages installed by `devtools::install_github()`

Troubleshooting

Q: I ran `update.packages("ggplot2")`, but nothing happened. No output on console, no error, nothing.
A: The first argument specifies the library location you want to search through (and update packages therein). `update.packages("ggplot2")` means you want to update the packages in library location `ggplot2`, which is most unlikely to exist on your R installation.

Q: I tried to update `ggplot2` with `install.packages("ggplot2")`, but nothing happened.
A: If `ggplot2` is already loaded, then you can't install `ggplot2` in the current session now. If you need to, save any objects you can't easily recreate, and quit out of R. Then start a new R session, immediately run `install.packages("ggplot2")`, then once finished, load the package and reload in any previously saved objects.

More about `update.packages`:

- `update.packages(lib.loc = NULL, repos = getOption("repos"), ask = TRUE)`: First a list of all packages found in `lib.loc` is created and compared with those available at the repositories. If `ask = TRUE` (the default) packages with a newer version are reported and for each one the user can specify if it should be updated. If so the packages are downloaded from the repositories and installed in the respective library path (or `instlib` if specified).
- You can specify one specific package to update using `update.packages(oldPkgs = "ggplot2")`. It will check updates only for that package and ask you if you want to update.

The easiest way to update an individual package is just to use `install.packages`. It is a one step command, compared to `update.packages`, which first checks and then asks.

- `update.packages` returns NULL invisibly.
- Be aware that some package updates may cause your previous code to stop working. For this reason, we recommend updating all your packages once at the beginning of each academic year (or semester) – don't do it before an assessment or deadline just in case!

Updating all Packages after R update

R packages are missing after updating. So you have to save the installed packages and re-install them after updating.

- Alternatively, `installr::updateR()` automatically updates R and installs your packages.

Here is how to do it manually.

```
## get packages installed
packs <- as.data.frame(installed.packages(.libPaths()[1]), stringsAsFactors = F)
# Save to local
f_name <- "~/Documents/R/packages.csv"
rownames(packs)
write.csv(packs, f_name, row.names = FALSE)
packs <- read_csv(f_name)
packs
## Re-install packages using install.packages() after updating R
install.packages(packs$Package)
```

R library path `/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/library`

- use `find.package("ggplot2")` to find the location to where the given package is found.
- alternatively, you can run `.libPaths()`
  - `.libPaths()` without an argument will return a list of all the places R will currently look for loading a package when requested.
  - `.libPaths("a/b/c")` with an argument will add that new directory (`"a/b/c"`) to the ones R was already using. If you use that directory often enough, you may wish to add that call to `.libPaths("a/b/c")` in your `.Rprofile` startup file in your home directory.

Put your R package on GitHub

Reference: https://jennybc.github.io/2014-05-12-ubc/ubc-r/session2.4_github.html

- Change to the package directory
- Initialize the repository with `git init`
- Add and commit everything with
    1. `git add .` stage changes;
    2. `git status` optional check staged changes, but yet to submit;
    3. and `git commit` submit staged changes.
- Create a new repository on GitHub
- Connect your local repository to the GitHub one

```
# add repo name "origin" to the remote repo at the URL
git remote add origin https://github.com/username/reponame
```

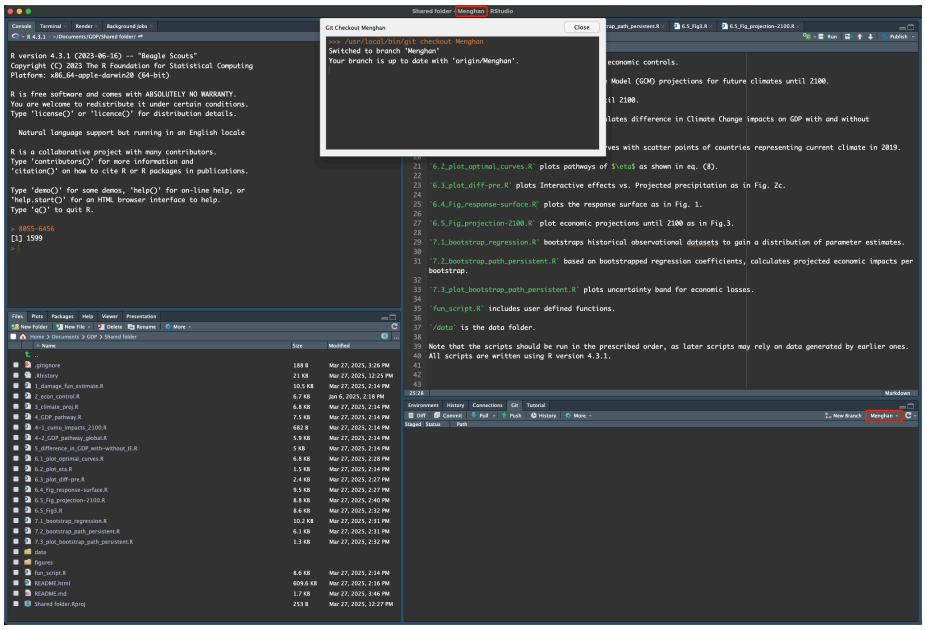
- Push everything to github

```
# rename the current local branch to "main"
git branch -M main
# creates a remote branch "origin" and sets it upstream of the "main" branch
git push -u origin main
```

2.4 Using Git with RStudio

Before you start coding, make sure that you are on the correct branch. You may check

- from the Git tab on the Environment, History, Connections, … pane
- you can also see from the status bar on the very top of the window. The words are formatted as “Projection Name – Branch – RStudio”.



Choose a License for your repo

Q: Which open source license is appropriate for my project?
A: See https://opensource.guide/legal/#which-open-source-license-is-appropriate-for-my-project.

Q: How to add a license to my repo?
A: Follow the instructions here.

2.5 Save R Workspace

If you want to saves all objects in your work space, use `save.image()`. It will creates an image of your current variables and functions, and saves them to a file called `.RData`. When R next loads, objects stored in this image are by default restored.

This sounds convenient, however, you do NOT want to do this because this corrupt reproducibility of your project.

You want to start from a clean slate very time.

It is suggested change RStudio Global Options to

- *not* "restore `.RData` into workspace at startup", and
- *never* "save workspace to `.RData` on exit".

In case you do feel the need to save the workspace, use the following cmd.

```
save.image(file = ".RData", version = NULL, ascii = FALSE, compress
= !ascii, safe = TRUE)
```

```
## save current workspace ##
f_name <- "RImage/TCR_2023-05-09.RData"
f_name
save.image(f_name)
# load(f_name)
```

Q: Can I save the loaded packages in the current session/workspace?
A: The workspace is for *objects* like data and functions. Starting R with particular packages loaded is what your `.Rprofile` file is for, and you can have a different one in each directory. But I'd recommend not saving anything between r sessions and instead recreate it all using code. This is much more likely to lead to reproducible results.

History

When you quit a project, `.Rhistory` is automatically written to the project directory unless you opt out to. It contains a history of all of the commands that you have sent to the R console in this session.

Pop out an editor

Click the `Show in New Window` button in any source editor tab.

To return a document to the main window, click the `Return to Main Window` button on the editor toolbar.

Environment Pane

By default, the Environment pane is located in the top-right and includes the Environment, History, Connections, Build, and Version Control System (VCS) tabs.

Version Control System (VCS)

The VCS tab will change based on the version control system you have enabled for that session. For example, using Git will change the tab name to Git and provide some common commands for viewing diffs, committing changes, pull and push … Output pane

The Output pane displays various outputs such as plots, HTML content, or on-disk files. It contains the Files, Plots, R Packages, Help, Viewer, and Presentation tabs.

Ref: RStudio Pane Layout

Global Options that make coding easier

- Syntax highlight and matched parentheses.

  Under "Tools -> Global Options -> Code -> Display", under **Syntax section**, check the boxes for **highlight R function calls** and **use rainbow parentheses**. The second is especially useful to mark matching opening and closing brackets.

- Show whitespace characters.

  In "Tools -> Global Options -> Code -> Display", check "Show whitespace characters". This will let you see spaces and newlines in the editor.

Chapter 3

Knit Rmd

R Markdown is a powerful tool for combining analysis and reporting into the same document. R Markdown has grown substantially from a package that supports a few output formats, to an extensive and diverse ecosystem that supports the creation of books, blogs, scientific articles, websites, and even resumes.

Nice documentations

- R markdown: The definitive guide. provides detailed references
- R markdown cookbook concise and covers essential functions, with examples.

Q: What is the difference between Rmd and R script?
A:

- An R script (`.R`) is used for developing and troubleshooting code; a place where you can store reusable code fragments.
- An R Markdown file (`.Rmd`) is used to integrate R commands with explanatory text and output, making it useful for creating reports.

Quick takeaways:

- Can still use horizontal separator ctrl + shift + S for dashed lines and ctrl + shift + = for equals
- Headers must have one empty line above and below to separate it from other text

---

YAML metadata

Q: What is YAML?

A: YAML is a human-friendly data serialization language for all programming languages.

Q: What does YAML do?

A: It is placed at the very beginning of the document and is read by each of Pandoc, **rmarkdown**, and **knitr**.

- Provide metadata of the document.
- located at the top of the file.
- adheres to the YAML format and is delimited by lines containing three three dashes (`---`).

It can set values of the template variables, such as `title`, `author`, and `date` of the document.

- The `output` field is used by rmarkdown to apply the output format function `rmarkdown::html_document()` in the rendering process.

  There are two types of output formats in the **rmarkdown** package: documents (e.g., `pdf_document`), and presentations (e.g., `beamer_presentation`).

  Supported output format examples: `html_document`, `pdf_document`.

  R Markdown documents (`html_documents`) and R Notebook documents (`html_notebook`) are very similar; in fact, an R Notebook document is a special type of R Markdown document. The main difference is using R Markdown document (`html_documents`) you have to knit (render) the entire document each time you want to preview the document, even if you have made a minor change. However, using an R Notebook document (`html_notebook`) you can view a preview of the final document without rendering the entire document.

- Many aspects of the LaTeX template used to create PDF documents can be customized using *top-level* YAML metadata (note that these options do not appear underneath the `output` section, but rather appear at the top level along with `title`, `author`, and so on). For example:

```
---
title: "Crop Analysis Q3 2013"
output: pdf_document
fontsize: 11pt
geometry: margin=1in
---
```

  A few available metadata variables are displayed in the following (consult the Pandoc manual for the full list):

Variable	Description
`lang`	Document language code
`fontsize`	Font size (e.g., `10pt`, `11pt`, or `12pt`)
`documentclass`	LaTeX document class (e.g., `article`)
`classoption`	Options for documentclass (e.g., `oneside`)
`geometry`	Options for geometry class (e.g., `margin=1in`)
`mainfont`, `sansfont`, `monofont`, `mathfont`	Document fonts (works only with `xelatex` and `lualatex`)
`linkcolor`, `urlcolor`, `citecolor`	Color for internal links (cross references), external links (link to websites), and citation links (bibliography)
`linestretch`	Options for line spacing (e.g. 1, 1.5, 3).

– In PDFs, you can use code, typesetting commands (e.g., `\vspace{12pt}`), and specific packages from LaTeX.

1. The `header-includes` option loads LaTeX packages.

```
---
output: pdf_document
header-includes:
- \usepackage{fancyhdr}
---

\pagestyle{fancy}
\fancyhead[LE,RO]{Holly Zaharchuk}
\fancyhead[LO,RE]{PSY 508}

# Problem Set 12
```

2. Alternatively, use `extra_dependencies` to list a character vector of LaTeX packages. This is useful if you need to load multiple packages:

```
---
title: "Untitled"
output:
  pdf_document:
    extra_dependencies: ["bbm", "threeparttable"]
---
```

f you need to specify options when loading the package, you can add a second-level to the list and provide the options as a list:

```yaml
---
title: "Untitled"
output:
  pdf_document:
    extra_dependencies:
      caption: ["labelfont={bf}"]
      hyperref: ["unicode=true", "breaklinks=true"]
      lmodern: null
---
```

Here are some examples of LaTeX packages you could consider using within your report:

* pdfpages: Include full PDF pages from an external PDF document within your document.
* caption: Change the appearance of caption subtitles. For example, you can make the figure title italic or bold.
* fancyhdr: Change the style of running headers of all pages.

– Some options are passed to Pandoc, such as `toc`, `toc_depth`, and `number_sections`. You should consult the Pandoc documentation when in doubt.

```yaml
---
output:
  pdf_document:
    toc: true
        keep_tex: true
---
```

* `keep_tex: true` if you want to keep intermediate TeX. Easy to debug. Defaults to `false`.

We can include variables and R expressions in this header that can be referenced throughout our R Markdown document. For example, the following header defines `start_date` and `end_date` parameters, which will be reflected in a list called `params` later in the R Markdown document.

Thus, if we want to use these values in our R code, we can access them via `params$start_date` and `params$end_date`.

Should I use quotes to surround the values?

- Whenever applicable use the unquoted style since it is the most readable.
- Use quotes when the value can be misinterpreted as a data type or the value contains a `:`.

```yaml
# values need quotes
foo: '{{ bar }}' # need quotes to avoid interpreting as `dict` object
foo: '123'       # need quote to avoid interpreting as `int` object
```

```
foo: 'yes'           # avoid interpreting as `boolean` object
foo: "bar:baz:bam" # has colon, can be misinterpreted as key

# values need not quotes
foo: bar1baz234
bar: 123baz
```

3.1 Chunk Options

If you want to set chunk options globally, call `knitr::opts_chunk$set()` in a code chunk (usually the first one in the document), e.g.,

````
```{r, label="setup", include=FALSE}
knitr::opts_chunk$set(
  comment = "#>", echo = FALSE, fig.width = 6
)
```
````

Full list of chunk options: https://yihui.org/knitr/options/

Chunk options can customize nearly all components of code chunks, such as the source code, text output, plots, and the language of the chunk.

Other languages are supported in `Rmd`

You can list the names of all available engines via:

```
names(knitr::knit_engines$get())
##  [1] "awk"          "bash"         "coffee"
##  [4] "gawk"         "groovy"       "haskell"
##  [7] "lein"         "mysql"        "node"
## [10] "octave"       "perl"         "php"
## [13] "psql"         "Rscript"      "ruby"
## [16] "sas"          "scala"        "sed"
## [19] "sh"           "stata"        "zsh"
## [22] "asis"         "asy"          "block"
## [25] "block2"       "bslib"        "c"
## [28] "cat"          "cc"           "comment"
## [31] "css"          "ditaa"        "dot"
## [34] "embed"        "eviews"       "exec"
## [37] "fortran"      "fortran95"    "go"
## [40] "highlight"    "js"           "julia"
## [43] "python"       "R"            "Rcpp"
## [46] "sass"         "scss"         "sql"
## [49] "stan"         "targets"      "tikz"
## [52] "verbatim"     "theorem"      "lemma"
## [55] "corollary"    "proposition"  "conjecture"
```

```
## [58] "definition"   "example"      "exercise"
## [61] "hypothesis"   "proof"        "remark"
## [64] "solution"     "marginfigure"
```

The engines from `theorem` to `solution` are only available when you use the **bookdown** package, and the rest are shipped with the **knitr** package.

To use a different language engine, you can change the language name in the chunk header from `r` to the engine name, e.g.,

````
```python
x = 'hello, python world!'
print(x.split(' '))
```
````

For engines that rely on external interpreters such as `python`, `perl`, and `ruby`, the default interpreters are obtained from `Sys.which()`, i.e., using the interpreter found via the environment variable `PATH` of the system. If you want to use an alternative interpreter, you may specify its path in the chunk option `engine.path`.

For example, you may want to use Python 3 instead of the default Python 2, and we assume Python 3 is at `/usr/bin/python3`

````
```{python, engine.path = '/usr/bin/python3'}
import sys
print(sys.version)
```
````

- All outputs support markdown syntax.
- If the output is html, you can write in html syntax.

The **chunk label** for each chunk is assumed to be unique within the document. This is especially important for cache and plot filenames, because these filenames are based on chunk labels. Chunks without labels will be assigned labels like `unnamed-chunk-i`, where `i` is an incremental number.

- Chunk label doesn't need a `tag`, i.e., you only provide the `value`.
- If you prefer the form `tag=value`, you could also use the chunk option `label` explicitly, e.g.,

  ````
  ```{r, label='my-chunk'}
  # one code chunk example
  ```
  ````

You may use `knitr::opts_chunk$set()` to change the default values of chunk options in a document.

Commonly used chunk options

- Complete list here. Or `?opts_chunk` to get the help page.

Options	Definitions
`echo=TRUE`	Whether to display the **source code** in the output document.Use this when you want to show the output but not the code itself.
`eval=TRUE`	Whether to evaluate the code chunk.
`include=TRUE`	Whether to include the chunk **output** in the output document. If `FALSE`, nothing will be written into the output document, but the code is still evaluated and plot files are generated if there are any plots in the chunk, so you can manually insert figures later.
`message=TRUE`	Whether to preserve messages emitted by `message()`
`warning=TRUE`	Whether to show warnings in the output produced by `warning()`.
`results='markup'`	Controls how to display the text results. When `results='markup'` that is to write text output as-is, i.e., write the raw text results directly into the output document without any markups.Useful when priting `stargazer` tables.
`comment='##'`	The prefix to be added before each line of the text output. Set `comment = ''` remove the default `##`.
`fig.keep='high'`	How plots in chunks should be kept. `high`: Only keep high-level plots (merge low-level changes into high-level plots). `none`: Discard all plots. `all`: Keep all plots (low-level plot changes may produce new plots). `first`: Only keep the first plot. `last`: Only keep the last plot. If set to a numeric vector, the values are indices of (low-level) plots to keep.If you want to choose the second to the fourth plots, you could use `fig.keep = 2:4` (or remove the first plot via `fig.keep = -1`).
`fig.align="center"`	Figure alignment.
`fig.pos="H"`	A character string for the figure position arrangement to be used in `\begin{figure}[]`.
`fig.cap`	Figure caption.

`results='markup'` note plural form for result**s**.

- `markup`: Default. Mark up text output with the appropriate environments depending on the output format. For example, for R Markdown, if the text output is a character string `"[1] 1 2 3"`, the actual output that **knitr** produces will be:

  ```
  ```
  [1] 1 2 3
  ```
  ```

  In this case, `results='markup'` means to put the text output in fenced

code blocks (“‘).

- `asis`: Write text output as-is, i.e., write the raw text results directly into the output document without any markups.

  ```
  ```{r, results='asis'}
  cat("I'm raw **Markdown** content.\n")
  ```
  ```

  Sometime, you encounter the following error messages when you have R codes within `enumerate` environment.

  > You can't use `macro parameter character #` in horizontal mode.

  By default, knitr prefixes R output with `##`, which can't be present in your TeX file.

  Solution:

  – specify `results="asis"` in code chunks.

- `hold`: Hold all pieces of text output in a chunk and flush them to the end of the chunk.
- `hide` (or `FALSE`): Hide text output.

3.2 Print Verbatim R code chunks

Including verbatim R code chunks inside R Markdown

One solution for including verbatim R code chunks (see below for more) is to insert hidden inline R code (`` `r ''` ``) immediately before or after your R code chunk.

- The hidden inline R code will be evaluated as an inline expression to an empty string by knitr.

Then wrap the whole block within a markdown code block. The rendered output will display the verbatim R code chunk — including backticks.

R code generating the four backticks block:

```
output_code <-
"````markdown
```{r}
plot(cars)
``` \n````"
cat(output_code)
```

Write this code in your R Markdown document:

````
````markdown
`r ''`````{r}
plot(cars)
```
````
````

or

````
````markdown
```{r}`r '''`
plot(cars)
```
````
````

Knit the document and the code will render like this in your output:

````
```{r}
plot(cars)
```
````

This method makes use of **Markdown Syntax** for code.

Q: What is the Markdown Syntax for code?
A:

- Inline code use a pair of backticks, e.g., `` `code` ``. To use $n$ literal backticks, use at least $n+1$ backticks outside. Note that use a space to separate your outside backticks from your literal backtick(s). For example, to generate `` `code` ``, you use `` ``␣`code`␣`` `` (i.e., two backticks + space + one backtick + `code` + one backtick + space + two backticks). Note that you need to write sequentially.
- Plain code blocks can be written either
  - After three or more backticks (fenced code blocks), or

    Can also use tildes (`~`)
  - Indent the blocks by four spaces (indented code blocks)

    Special characters do not trigger special formatting, and all spaces and line breaks are preserved. Blank lines in the verbatim text need not begin with four spaces.
- Note that code blocks must be separated from surrounding text by blank lines.

If the code itself contains a row of tildes or backticks, just use a longer row of tildes or backticks at the start and end:

```
~~~~~~~~~~~~~~~~
~~~~~~~~~~
code including tildes
```
```

~~~~~~~~~~
~~~~~~~~~~~~~~~~

These begin with a row of three or more tildes (~) and end with a row of tildes that must be at least as long as the starting row.

A shortcut form (without braces) can also be used for specifying the language of the code block:

```
```haskell
qsort [] = []
```
```

This is equivalent to:

```
``` {.haskell}
qsort [] = []
```
```

`haskell` is the language class.

You can add more classes, such as `numberLines` for adding line numbers.

This shortcut form may be combined with attributes:

```
```haskell {.numberLines}
qsort [] = []
```
```

Which is equivalent to:

```
``` {.haskell .numberLines}
qsort [] = []
```
```

and

```
<pre id="mycode" class="haskell numberLines" startFrom="100">
  <code>
  primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, x `mod` p /= 0]
  </code>
</pre>
```

If highlighting is supported for your output format and language, then the code block above will appear highlighted, with numbered lines starting with 100, 101, and go on.

---

**References**:

https://yihui.org/en/2017/11/knitr-verbatim-code-chunk/

https://support.posit.co/hc/en-us/articles/360018181633-Including-verbatim-R-code-chunks-inside-R-Markdown

https://themockup.blog/posts/2021-08-27-displaying-verbatim-code-chunks-in-xaringan-presentations/

Pandoc's Markdown: https://pandoc.org/MANUAL.html#fenced-code-blocks

# Chapter 4

# Basic R

Save & Load R objects

`save(..., f_name)` and `saveRDS()`

`save()` When loaded the named object is restored to the current environment (in general use this is the global environment — the workspace) **with the same name it had when saved**.

`save` writes an external representation of **R** objects to the specified file. The objects can be read back from the file at a later date by using the function `load` or `attach` (or `data` in some cases).

```
save(..., list = character(),      file = stop("'file' must
be specified"),      ascii = FALSE, version = NULL, envir =
parent.frame(),      compress = isTRUE(!ascii), compression_level,
eval.promises = TRUE, precheck = TRUE)
```

- `...` The names of the objects to be saved (as symbols or character strings).
- `list` A character vector containing the names of objects to be saved.
  - The names of the objects specified **either** as symbols (or character strings) in `...` or as a character **vector** in `list` are used to look up the objects from environment `envir`.
- `file` the name of the file where the data will be saved.

`saveRDS()` doesn't save the both the object and its name it just saves a representation of the object. As a result, the saved object can be loaded into a named object within R that is different from the name it had when originally serialized.

> Serialization is the process of converting a data structure or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and "resurrected" later in the same or another computer environment.

`saveRDS` works only for saving a single R object, `save()` can save multiple R objects in one file. A workaround for `saveRDS` is to save all target objects in a single R object (e.g., in a **list**), and then use `saveRDS()` to save it at once.

```
datalist = list(mtcars = mtcars, pressure=pressure)
saveRDS(datalist, "twodatasets.RDS")
rm(list=ls())

datalist = readRDS("twodatasets.RDS")
datalist
```

---

### Load R objects

`load(f_name)` to load `.rda` file.

`readRDS(f_name)` to load `.rds` file.

Naming conventions:

- `rda` and `rds` for selected objects
- `.RData` for all objectes in your workspace
- The file extensions are up to you; you can use whatever file extensions you want.

An example

```
> require(mgcv)
Loading required package: mgcv
This is mgcv 1.7-13. For overview type 'help("mgcv-package")'.
> mod <- gam(Ozone ~ s(Wind), data = airquality, method = "REML")
> mod

Family: gaussian
Link function: identity

Formula:
Ozone ~ s(Wind)

Estimated degrees of freedom:
3.529  total = 4.529002

REML score: 529.4881
> save(mod, file = "mymodel.rda")
> ls()
[1] "mod"
> load(file = "mymodel.rda")
> ls()
```

```
[1] "mod"


> ls()
[1] "mod"
> saveRDS(mod, "mymodel.rds")
> mod2 <- readRDS("mymodel.rds")
> ls()
[1] "mod"  "mod2"
> identical(mod, mod2, ignore.environment = TRUE)
[1] TRUE
```

---

**Save figures in a list**

```
p_list <- list(p_ano=p_ano, p_tr=p_tr)
# p_list[[name]] <- p_obj
p_list[[1]]

f_name <- paste0(fig_dir, sprintf("trend_analysis/image_list_%s.rds", con_name))
# saveRDS(p_list, f_name)

# plot in a panel grid
p_allCON <- plot_grid(plotlist=p_list, align="vh", labels=sprintf("(%s)", letters[1:length(p_list
p_allCON
```

---

## 4.1 Data Input & Output

### 4.1.1 Read Data

**Read Fortran**

`read.fortran(file, format, ..., as.is = TRUE, colClasses = NA)`

- `format` Character vector or list of vectors.

**Read dta**

`haven::read_dta()` read Stata data file.

```
data <- read_dta("climate_health_2406yl.dta")
# retrieve variable labels/definitions
var_dict <- tibble("name" = colnames(data),
                   "label" = sapply(data, function(x) attr(x, "label")) %>%
                                as.character()
```

```
                    )
var_dict

var_label(data$gor) # get variable label
val_labels(data$gor) # get value labels
```

### Read fixed width text files

`read.fwf(file, widths)`

- `widths` integer vector, giving the widths of the fixed-width fields (of one line), or list of integer vectors giving widths for multiline records.

`read.table(f_name, header=FALSE, row.names, col.names, sep="", na.strings = "NA")` a very versatile function. Can be used to read `.csv` or `.txt` files.

- `f_name` path to data.
- `header=FALSE` defaults to `FALSE`, assumes there is no header row in the file unless specified otherwise.
  - If there is a header in the first row, should specify `header=TRUE`.
- `row.names` a vector of row names. This can be
  - *a vector* giving the actual row names, or
  - *a single number* giving the column of the table which contains the row names, or
  - *character string* giving the name of the table column containing the row names.
- `col.names` a vector of optional names for the variables. The default is to use `"V"` followed by the column number.
- `sep` use white space as delimiter.
  - if it is a `csv` file, use `sep=','` to specify comma as delimiter
- `na.strings = "NA"` a character vector of strings which are to be interpreted as `NA` values.
  - A useful setting: `na.strings = c("", "NA", "NULL")`

`read.csv(f_name, header = TRUE, sep = ",", na.strings = "..", dec=".")`

- `header = TRUE` whether the file contains the names of the variables as its first line.
- `sep` the field separator string. Values within each row of `x` are separated by this string.
- `na` the string to use for missing values in the data.
- `dec` the string to use for decimal points in numeric or complex columns: must be a single character.
- `fileEncoding` `UTF-8`

When reading data from github, you need to pass in the raw version of the data in `read.csv()`,

R cannot read the display version.

You can get the URL for the raw version by clicking on the Raw button displayed above the data.

```
# read.table can be used to read txt and csv. Need to specify sep=',' when reading csv.
data <- read.table("https://raw.githubusercontent.com/my1396/course_dataset/refs/heads/main/boned
data

data <- read.table("https://raw.githubusercontent.com/my1396/course_dataset/refs/heads/main/boned

# can use read_csv or read.csv
data <- read_csv("https://raw.githubusercontent.com/my1396/course_dataset/refs/heads/main/boneden
data
```

`read_delim(f_name, delim=";")` allows you to specify the delimeter as `;`.

`readr::read_csv(f_name, na = c("..", NA, ""), locale = locale(encoding = "UTF-8"), col_types = cols(Date = col_date(format = "%m/%d/%y")) )` read comma separated values.

- `col_types` specify column types. Could be created by `list()` or `cols()`.

  `read_csv` will automatically guess, if you don't explicitly specify column types. You can override column types by providing the argument `col_types`. You don't need to provide all column types, just the ones you want to override.

- By default, reading a file without a column specification will print a message showing what `readr` guessed they were. To remove this message,
  - set `show_col_types = FALSE` for one time setting, or
  - set `options(readr.show_col_types = FALSE)` for the current sessions' global options setting. If want to change permanently everytime when R starts, put `options(readr.show_col_types = FALSE)` in `.Rprofile` as global options.

`read_tsv()` read tab separated values.

`read_csv2(f_name, na = c("..", NA, ""))` use semicolon `;` to separate values; and use comma `,` for the decimal point. This is common in some European countries.

- `locale` The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
- `locale(date_names = "en", date_format = "%AD", time_format = "%AT", decimal_mark = ".", grouping_mark = ",", tz = "UTC", encoding = "UTF-8", asciify = FALSE)`
  - `decimal_mark` indicate the decimal place, can only be `,` or `.`

  - `encoding` This only affects how the file is read - readr always converts the output to UTF-8.

---

### 4.1.2 Write Data

Save data in `uft8` encoding with special language characters

`write_excel_csv()` include a UTF-8 Byte order mark which indicates to Excel the csv is UTF-8 encoded.

`write.csv(x, f_name, row.names=TRUE, fileEncoding ="UTF-8")`

- `x` a matrix or data frame. If not one of the types, it is attempted to coerce `x` to a data frame.
  - `write_csv(x)` `x` can only be data frame or tibble. Doesn't support matrix.

```
mat %>% as_tibble(rownames = "rowname") %>% write_csv("mat.csv")
mat %>% write.csv("mat.csv")
```

- `row.names` wether to write row names of `x`. Defaults to `TRUE`.

# Chapter 5

# Machine Learning

Parametric models such as generalized linear regression and logistic regression has advantages and disadvantages.

**Strength**:

- The effects of individual predictors on the outcome are easily understood
- Statistical inference, such as hypothesis testing or interval estimation, is straightforward
- Methods and procedures for selecting, comparing, and summarizing these models are well-established and extensively studied

**Disadvantages** in the following scenarios:

- Complex, non-linear relationships between predictors and the outcome
- High degrees of interaction between predictors
- Nominal outcome variables with several categories

In these situations, non-parametric or algorithmic modeling approaches have the potential to better capture the underlying trends in the data.

Here we introduce three models: classification and regression trees (CART), random forests, k-nearest neighbors.

- Classification and regression trees (CART) are "trained" by recursively partitioning the  -dimensional space (defined by the explanatory variables) until an acceptable level of homogeneity or "purity" is achieved within each partition.
- A major issue with tree-based models is that they tend to be high variance (leading to a high propensity towards over-fitting). Random forests are a non-parametric, tree-based modeling algorithm that is built upon the idea that averaging a set of independent elements yields an outcome with lower variability than any of the individual elements in the set.

This general concept should seem familiar. Thinking back to your introductory statistics course, you should remember that the sample mean, $\overline{x}$, of a dataset has substantially less variability ($\frac{\sigma}{\sqrt{n}}$) than the individual data-points themselves ($\sigma$).

Q: What is Bias-Variance Trade-Off in Machine Learning?
A:

- Bias refers to error caused by a model for solving complex problems that is over simplified, makes significant assumptions, and misses important relationships in your data.
- Variance error is variability of a target function's form with respect to different training sets. Models with small variance error will not change much if you replace couple of samples in training set. Models with high variance might be affected even with small changes in training set. High variance models fit the data too well, and learns the noise in addition to the inherent patterns in the data.

## 5.1 Random Forest

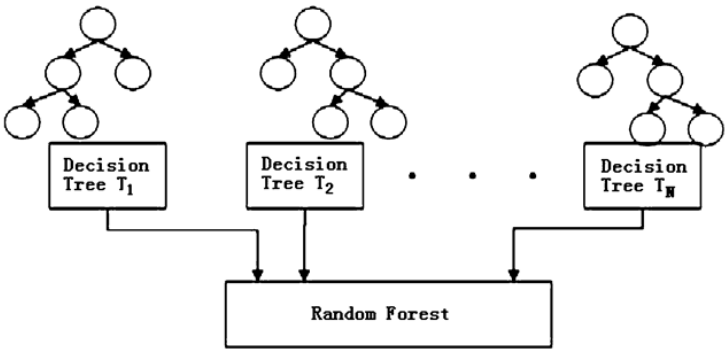
**Averaging of independent trees**

The goal of bagging is to produce $B$ separate training datasets that are independent of each other (typically  is in the hundreds). The model of interest (in this case classification and regression trees) is trained separately on each of these datasets, resulting in $B$ different estimated "models". These are then averaged to produce a single, low-variance estimate.

Bagging is a general approach, but its most well-known application is in the random forest algorithm:

1. Construct $B$ bootstrap samples by sampling cases from the original dataset with replacement (this results in $B$ unique datasets that are similar to the original)
2. Fit a classification and regression tree to each sample, but randomly choose a subset of $m$ variables that can be used in the construction of that tree (this results in $B$ unique trees that are fit to similar datasets using different sets of predictors)
3. For a given data-point, each of the $B$ trees in the forest contributes a prediction or "vote", with the majority (or average) of these votes forming the random forest's final prediction, $\widehat{y}_i$

```
knitr::include_graphics("images/rf.png")
```




A downside of both the CART and random forest algorithms (as well as many other algorithmic modeling approaches) is an inability to clearly quantify the roles played by individual variables in making predictions. However, the importance of individual variables in a random forest can still be expressed using a measure known as variable importance.

The random forest algorithm requires the following tuning parameters be specified in order to run:

- `ntree` - the number of bagged samples, $B$, onto which trees will be grown
- `mtry` - the number of variables that are randomly chosen to be candidates at each split
- Some sort of stopping criteria for individual trees, this can be:
  - `nodesize`, which sets the minimum size of terminal nodes
    - larger `nodesize` leads to shallower trees
    - smaller node size allows for deeper, more complex trees
  - `maxnodes`, which sets the maximum number of terminal nodes an individual tree can have.

### Applications of Random Forest

Some of the applications of Random Forest Algorithm are listed below:

- Banking: It predicts a loan applicant's solvency. This helps lending institutions make a good decision on whether to give the customer loan or not. They are also being used to detect fraudsters.
- Health Care: Health professionals use random forest systems to diagnose patients. Patients are diagnosed by assessing their previous medical history. Past medical records are reviewed to establish the proper dosage for the patients.
- Stock Market: Financial analysts use it to identify potential markets for stocks. It also enables them to remember the behaviour of stocks.
- E-Commerce: Through this system, e-commerce vendors can predict the preference of customers based on past consumption behaviour.

**When to Avoid Using Random Forests?**

Random Forests Algorithms are not ideal in the following situations:

- Extrapolation: Random Forest regression is not ideal in the extrapolation of data. Unlike linear regression, which uses existing observations to estimate values beyond the observation range.
- Sparse Data: Random Forest does not produce good results when the data is sparse. In this case, the subject of features and bootstrapped sample will have an invariant space. This will lead to unproductive spills, which will affect the outcome.

**FAQ**

Q: Is RF a linear or non-linear model?
A: RF can capture complex, non-linear relationships.

Q: Is RF sensitive to Imbalanced Data?
A: Yes. It may perform poorly if the dataset is highly imbalanced like one class is significantly more frequent than another.

Q: What is the loss function?
A: Entropy/gini or any other loss function you want.

Q: Difference btw RF and a linear model?
A: A major difference is that a decision tree does not have "parameters", whereas the linear models need to create a functional form and find the optimal parameters.

---

## Implementation in R

`ranger` package offers a computation efficient function for RF.

```
RF_ranger <- ranger(formula = formula,
                    data = data_before[idx,],
                    probability = TRUE,
                    importance = "permutation",
                    scale.permutation.importance = TRUE,
                    )
    # print(RF_ranger)

rf.pred.test <- predict(RF_ranger, data=data_before[-idx,])$predictions
```

Parameters controlling the general process of RF:

- `probability=FALSE`: Whether to forecast a probability forest.

The hyperparameters `mtry`, `min.node.size` and `sample.fraction` determine the degree of randomness, and should be tuned.

- `mtry=500`: Number of variables to possibly split at in each node in one tree. In plain language, it indicates how many predictor variables should be used in each tree.
  - Default is the (rounded down) square root of the number variables. Alternatively, a single argument function returning an integer, given the number of independent variables.
  - Range btw 1 to the number of predictors.
  - If all predictors are used, then this corresponds in fact to bagging.
- `min.node.size`: The number of observations a terminal node should at least have.
  - Default 1 for classification, 5 for regression, 3 for survival, and 10 for probability. For classification, this can be a vector of class-specific values.
  - Range between 1 and 10
- `sample.fraction`: Fraction of observations to be used in each tree. Default is 1 for sampling with replacement and 0.632 for sampling without replacement. For classification, this can be a vector of class-specific values.
  - Smaller fractions lead to greater diversity, and thus less correlated trees which often is desirable.
  - Range between 0.2 and 0.9

Parameters controlling what and how intermediate results are saved:

- `keep.inbag = FALSE`: Whether to save how often observations are in-bag in each tree.

  Set to `TRUE` if you want to check sample composition in each tree.

- `importance = 'none'|'impurity'|'impurity_corrected'|'permutation'`: Variable importance mode.

- `scale.permutation.importance = FALSE`: Whether to scale permutation importance by standard error as in (Breiman 2001). Only applicable if `'permutation'` variable importance mode selected.

- `write.forest = TRUE`: Whether to save `ranger.forest` object, required for prediction. Set to `FALSE` to reduce memory usage if no prediction intended.

  - Set to `FALSE` when you do parameter tuning.

Q: How to tune hyperparameters?
A: Check out `mlr3` package. Here is an example.

---

## Imbalance Classification

You can balance your random forests using case weights. Here's a simple example:

```r
library(ranger)

# Make a dataste
set.seed(43)
nrow <- 1000
ncol <- 10
X <- matrix(rnorm(nrow * ncol), ncol=ncol)
CF <- rnorm(ncol)
Y <- (X %*% CF + rnorm(nrow))[,1]
Y <- as.integer(Y > quantile(Y, 0.90))
table(Y)

# Compute weights to balance the RF
w <- 1/table(Y)
w <- w/sum(w)
weights <- rep(0, nrow)
weights[Y == 0] <- w['0']
weights[Y == 1] <- w['1']
table(weights, Y)

# Fit the RF
data <- data.frame(Y=factor(ifelse(Y==0, 'no', 'yes')), X)
model <- ranger(Y~., data, case.weights=weights)
print(model)
```

Code Source: https://stats.stackexchange.com/a/287849

Fixed proportion sampling: https://github.com/imbs-hl/ranger/issues/167

---

**References**:

https://remiller1450.github.io/m257s21/Lab10_Other_Models.html

## 5.2 Neural Network

Neural networks are made up objects called "layers" and "neurons" and these things connect to each other in a specific way. Each layer has some number of neurons. For example, the first layer might have 10 neurons, the second might have 15, and so on. The number of layers and the number of neurons in each layer is a "hyperparameter", the user picks how many of each. Let's take a look at a single neuron.

$$v_3^{(1)} = g(w_3^{(1)}x + b_3^{(1)})$$

- The LHS, $v_3^{(1)}$, will be the output. The superscript (1) refers to the layer number; the subscript 3 refers to the neuron.

  An output here means just a single number. If we have say 15 neurons (subscript) for this first layer (superscript), then we will have 15 numbers come out of this first layer: $v^{(1)} = \{v_1^{(1)}, v_1^{(1)}, ..., v_{15}^{(1)}\}$ where the bolded $v$ means a vector.

- $x$ is our input vector.
- $w$ is the weight/coefficient vector.
- $b$ is a bias or the intercept term, shifting the value of $w \cdot x$ up or down.
- $g$ refers to a "non-linear" function, often called as the "activation function".