Invertible Reproducible Documents

Eric Lim

October 16, 2014

Motivation

Knitr(REF) is a wonderful package that enables dynamic report generation with R. It allows integration of R code into LaTeX, LyX, HTML, Markdown, AsciiDoc, and reStructuredText documents through the concepts of literate programming (REF), which involves interaction between code and documentation for report generation. The main purpose of **knitr** follows an important idea in academic research that the ideal research paper or report must encompass the entire computational environment used to produce the results in the paper such as the code so that the same results can be reproduced using the same computational environment (reproducible research REF). The importance of reproducibility can be further extended to be applied in the entire area of academia (REF).

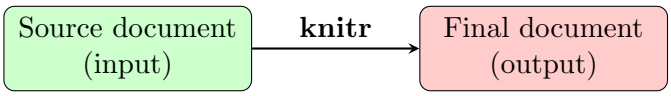


Figure 1.1: Unidirectional document generation by **knitr**

While there are tremendously important ideas to consider and many advantages, the process of generating R code embedded documents using **knitr** is almost always a one-way trip (Figure 1.1), meaning source documents (as input) can only generate final documents (as output), not the other way around. This is simply due to the fact that **knitr** is designed with intention to dynamically generate reports, not to extract displayed R code in final documents in order to generate source documents.

Consider a situation where a *consultant* provides reports as reproducible documents and a *client* is to read or review the reports. In this situation, it is often difficult for the consultant to receive feedback from the client efficiently. Since it is impossible to convert back from final documents to source documents, the client would have to provide his or her feedback by either writing physically on the printed copy of the final report or by electronic means such as through exchanging e-mails. The document provider, then, has to rectify the source documents accordingly, only to repeat the process of generating and presenting the report to the client. This process often has to be repeated until final correction can be achieved. As we can see, it can quickly become tedious.

We believe this is relevant to the field of statistics as similar situations

mentioned above can often arise. Interaction between clients and consultants is crucial for statisticians and any possible factor to deteriorate the relationship with clients is best avoided. Therefore, we want to avoid this issue by a more efficient document generation workflow.

A possible solution is to allow clients to interact directly with the final documents and edit them. Then consultants can merge the changes and make final adjustments to generate new reports efficiently. In order to achieve this, reports must be *invertible*, as well as reproducible, that is, final reports must be able to be converted into the source document format without introducing any change from the inversion process itself.

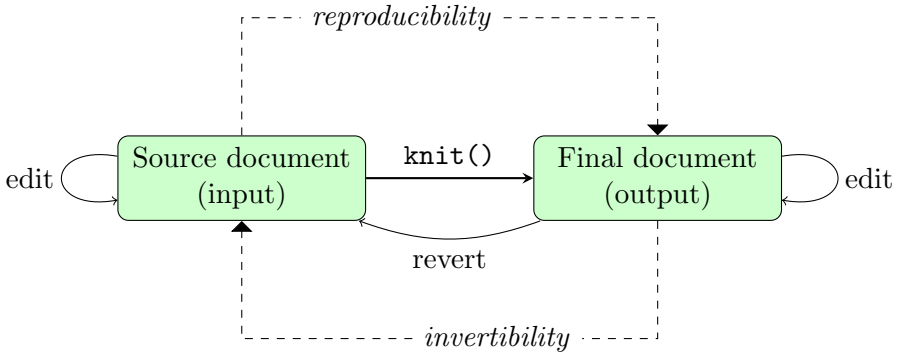
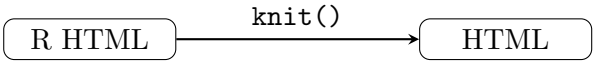


Figure 1.2: Invertible reproducible workflow

This report introduces a possible workflow that can be utilised effectively in the aforementioned situation. The main focus of the project is in achieving this hypothetical round trip and exploring possible issues associated with it. Generalising the workflow by improving the robustness and efficiency of the functions involved has only been considered to a level that has been manageble under the time given for the project. Hence, it has been possible to devise two separate strategies to deal *specifically* with two different document types under the allowed time. The phase 1 of this report will discuss primarily on dealing with HTML-based documents and the second phase will focus on Markdown-based documents.

Phase 1: R HyperText Markup Language



Introduction

The primary objective of the initial phase is to design and achieve a complete reproducible, invertible workflow. There are many obstacles that limit the possibility of devising an invertible workflow. A sizable one of these problems is that converting (or inverting) between two different document formats requires a tremendous amount of a particular resource that we were limited on; *time*.

Out of the many formats that **knitr** supports, R HTML is the format we have chosen for the objective. There are two main reasons for this choice; R HTML does not require conversion between two document formats and it allows the use of powerful text processing tools, all of which increase our chance of accomplishing a reproducible, invertible workflow.

Support of unified format

In **knitr**, R HTML-based documents are *knitted* to produce HTML-based documents. The *only* difference between these two document formats is that R HTML-based documents contain sections of lines of R code, known as R Code Chunks. Here is an example of a simple R Code Chunk:

```
<!--begin.rcode
plot(cars)
end.rcode-->
```

By using the function, `knit()`, in **knitr**, the R Code Chunk is run in R to produce output (a plot for the previous example). The R code, `plot(cars)`, and the output plot are, then, enclosed separately in nested `div` elements in the final HTML-based document.

Knitr generates HTML documents, as output, from *HTML-based* R HTML documents, as input. In other words, the document generation process does

not involve conversion between two distinct document formats. The source document format is essentially the same as the final document format.

By this property, difficulties associated with inverting a document format back into a different format can be minimised and the chance of inverting is increased.

Availability of tool sets

HTML is markup language very similar to XML. This means powerful XML-based tool sets, such as XPath (REF), are available for use to manipulate the HTML documents effectively. The R package, **XML** (REF), is used to provide these XML-based tool sets in R.

Demonstration

Here is a brief demonstration of how an R HTML document can be used to generate a HTML document.

Listing 2.1 is the code structure of the source document, `example1.Rhtml`. The highlighted text (lines 8–10) is an R Code Chunk. The option, `echo=FALSE`, is used to hide the R code in the final document.

Listing 2.1: `example1.Rhtml`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  </head>
5    <body>
6      <h1>Example</h1>
7      <p>Summary of the cars data:</p>
8      <!--begin.rcode echo=FALSE
9      summary(cars)
10     end.rcode-->
11   </body>
12 </html>
```

The following R code is used to *knit* the source document, `example1.Rhtml`, to generate the final document, `example1.html`:

```
> library(knitr)
> knit("example1.Rhtml")
```

Listing 2.2 shows the code structure of the final document, `example1.html`. The R code from line 9 of Listing 2.1 is run in R to produce the result seen

in lines 13–19 of Listing 2.2. Cautions must be taken as the code in Listing 2.2 has been tidied up with line endings to enhance readability. The actual untidied code consists of long lines that usually require text-wrap to fit into the page.

Listing 2.2: (tidied) `example1.html`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <style type="text/css"> ... </style>
5  </head>
6    <body>
7      <h1>Example</h1>
8      <p>Summary of the cars data:</p>
9      <div class="chunk" id="unnamed-chunk-1">
10       <div class="rcode">
11       <div class="output">
12         <pre class="knitr r">
13           ##     speed          dist
14           ##  Min.   : 4.0  Min.   :  2
15           ##  1st Qu.:12.0  1st Qu.: 26
16           ##  Median :15.0  Median : 36
17           ##  Mean   :15.4  Mean   : 43
18           ##  3rd Qu.:19.0  3rd Qu.: 56
19           ##  Max.   :25.0  Max.   :120
20         </pre></div>
21       </div></div>
22   </body>
23 </html>
```

Figure 2.1 shows what `example1.html` will look like in a web browser.

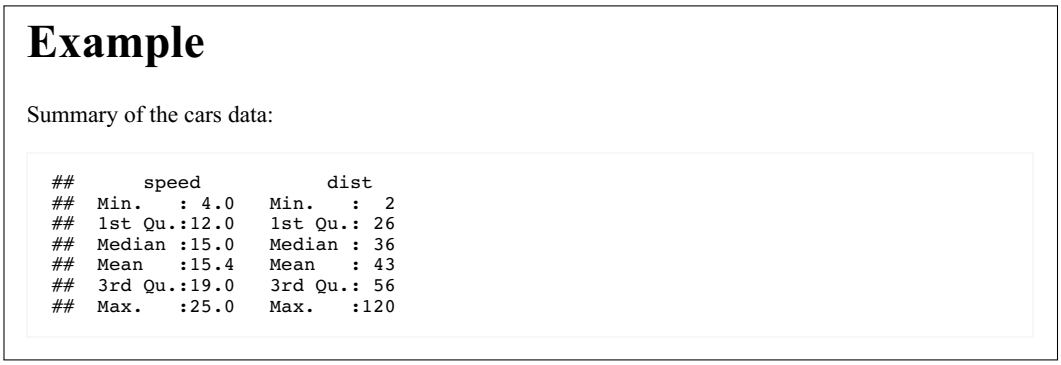


Figure 2.1: `example1.html` in a browser

Overview

A complete cycle of the invertible workflow consists of six stages. Each stage is involved primarily with text processing, along with an occasional use of XPath.

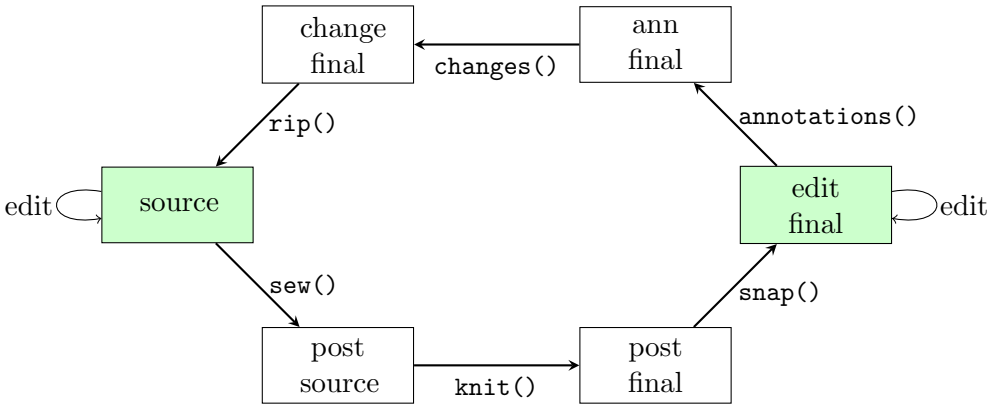


Figure 2.2: The reproducible, invertible workflow

Loss of information

Processing of a source document into a final document can be associated with loss of information from the source document. For example, the original R code in line 9 of Listing 2.1 does not appear in the final document (Listing 2.2). As a solution, a source document undergoes a pre-processing step where each R Code Chunk is copied in such a way that the original R Code Chunks are perfectly preserved throughout the stages of the workflow and are retainable after a complete cycle.

The pre-processing task is carried out by the function, `sew()`.

Gain of information

As opposed to losing information, the document generation process can result in gain of new information. For example, lines 9–19 of Listing 2.2, which correspond to the output generated from the original R code in line 9 of Listing 2.1, only exist in the post-processed (final) document. The sections of newly gained information, i.e., the output of R Code Chunks, may seem editable to the client.

However translating changes made on the gained information in the final document to the source document where the new information is non-existent

(the output is not generated yet) is problematic. And more technical side of the report, such as manipulating R code, is usually best left for the consultant.

A solution chosen for this project is to restrict these R-created sections of the final document to be annotated only. In this way, the client can still pass on feedbacks to the consultant without directly changing the R code or the output of the R code.

Editing final document

Allowing the client to modify the final document is achieved by adding JavaScript code to the final document. The JavaScript code loads two libraries, **CKEditor** (REF) and **Annotator** (REF), that provide relatively user-friendly GUIs.

The JavaScript code is added by the function, `snap()`.

Merging changes

Annotations and changes made on the final document are merged by the functions, `annotations()` and `changes()`, respectively.

The final stage of the invertible workflow involves text processing to remove any artifacts added by **knitr** and is carried out by the function, `rip()`.

Functions detail

`sew()`

The function carries out a pre-processing step, where R Code Chunks are copied. The copied R Code Chunks are delimited by `<!--begin.keepcode` and `end.keepcode-->` markup, and each copy is added *after* its original counterpart, correspondingly.

A simple example of a copied R Code Chunk is as below:

```
<!--begin.rcode echo=FALSE
summary(cars)
end.rcode-->
<!--begin.keepcode echo=FALSE
summary(cars)
end.keepcode-->
```

Inline R Code Chunks are delimited differently, by `<!--rinline.keep` and `-->` markup to differentiate them from the ordinary R Code Chunks.

The default suffix of the output file is `.post.Rhtml`, converted from the source suffix, `.Rhtml`.

`knit()`

The function is directly from the package, **knitr**, and carries out the usual knitting procedure. It should be noted that the copies of R Code Chunks generated from the previous step are *not* processed by `knit()`, and thus remain hidden in the subsequent steps.

The default suffix of the output file is `.html`, converted from the input suffix, `.Rhtml`.

`snap()`

The function is primarily involved with adding pieces of code to the (knitted) HTML document; it adds a piece of JavaScript code along with a piece of HTML script to the document and HTML attributes to the appropriate elements of the HTML document.

The JavaScript code contains `scripts` for loading the libraries, **CKEditor** and **Annotator**, responsible for providing the editable and annotatable GUIs, and **jQuery** (REF), on which the functionality of the former two libraries depends. The HTML script contains the layout of the save `button`.

The attributes, `contenteditable="true"` and `id="Editor"`, are added to

each element in the document that should be editable, that is, any top level element that is *not* a `div` element with `class="chunk"` attribute. **CKEditor** detects elements with `contenteditable="true"` attribute and provides the editable GUI for *each* of them. The other attribute, `id="Editor"`, is used to allow reliable matching of the elements when changes are merged.

Correspondingly, **Annotator** detects `div` elements with `class="chunk"` attribute, which denote the sections of R-created content, and provides the annotatable GUI on *each* of these sections.

Once the final editable document is prepared using `sew()`, `knit()` and `snap()`, it must be hosted on a web server to save changes and annotations from **CKEditor** and **Annotator**. A test server has been set up at `http://stat220.stat.auckland.ac.nz/cke/` and can be accessed with the username and password, `"cke"`. Instructions can be followed on the web site to upload the editable final document prepared by `snap()`.

Figures 2.3 and 2.4 present the editable and annotatble GUIs. The line , *"Summary of the cars data:"* (Figure 2.1), is edited to *"summary(cars)"* in the Teletype (REF) font family. And the annotation, *"need units"*, is made on the text, *"speed"*, of the output. The "save" button on top of the page can be clicked to store the change and annotation separately in text files on the test server.

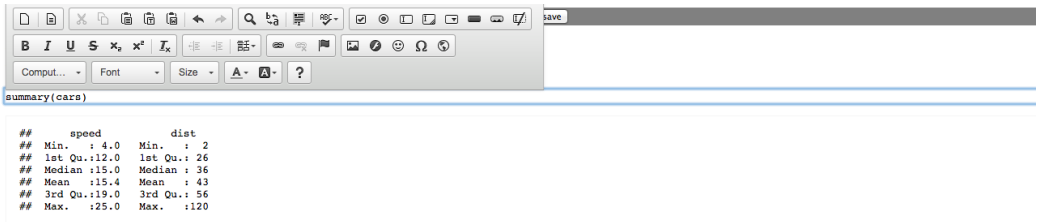


Figure 2.3: CKEditor GUI

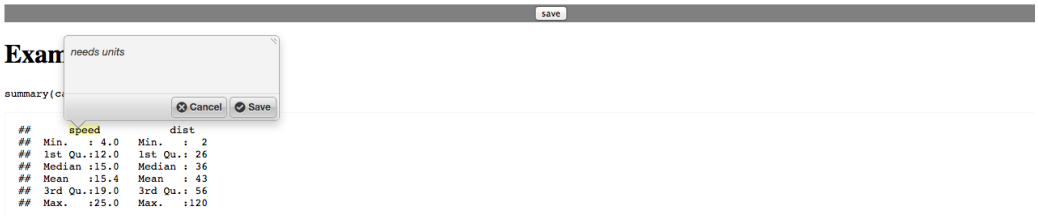


Figure 2.4: `Annotator` GUI

The annotation is saved on a plain text file, `test-annotations.txt`, in **JSON** (REF) format and the change is saved on `test-changes.txt` as sim-

ple text. The code structure of these two files can be seen in Listings 2.3 and 2.4.

Listing 2.3: `test-annotations.txt`

```
[
  {
    "ranges": [
    {
      "start":         "/div[1]/div[1]/pre[1]",
      "startOffset":    8,
      "end":           "/div[1]/div[1]/pre[1]",
      "endOffset":     13
      }
    ],
    "quote":"speed",
    "text":"needs units",
    "id":"61430634697899221413241555634"
  }
]
```

Listing 2.4: `test-changes.txt`

```
EDITOR Editor-1 NOT MODIFIED

EDITOR Editor-2:
<code>summary(cars)</code>
```

The output of `snap()` consists of one file, e.g., the editable final document with `.edit.html` suffix, converted from the input suffix, `post.html`. The editable document is, then, uploaded onto the test server where editing occurs.

When the editing is completed, cliking on the "save" button creates the two files, `test-annotations.txt` and `test-changes.txt`, on the test server for downloading and retrieval by other means.

`annotations()`

The function merges annotations from `test-annotations.txt` into the final document. The R package, **jsonlite** (REF), is the tool used to convert the JSON data (the annotations in `test-annotation.txt`) into an R object, so that they can be merged into the final document.

Since the R-created sections are restricted to be annotated only, i.e., clients are not allowed to edit them, they require manual adjustments. For this rea-

son, annotations are merged as (HTML) paragraphs with highlighted text and are inserted at the top of their original locations (in the R-created sections), to ease the burden of looking for them and manual merging them afterwards.

The default suffix of the output file is `.anns.html`, converted from the input suffix, `.edit.html`.

Listing 2.5 is the code structure of the final document merged with the annotation. The highlighted text, lines 10–12, displays the merged annotation.

Listing 2.5: (tidied) `example1.anns.html`

```
1  <!DOCTYPE html>
2  <html>
3  <head></head>
4    <body>
5      <p id="savebutton" style="background-color:grey; text-align:
           center">
6        <button onclick="savechanges()">save</button>
7      </p>
8      <h1 contenteditable="true" id="Editor-1">Example</h1>
9      <p contenteditable="true" id="Editor-2">Summary of the cars data
           :</p>
10     <p class="annotation" style = "background-color:coral">
11     <em>Annotation:  The text "speed" was annotated with the
12     message "needs units"</em></p>
13     <div class="chunk" id="unnamed-chunk-1">
14       <div class="rcode">
15       <div class="output">
16         <pre class="knitr r">
17           ##      speed          dist
18           ##  Min.   : 4.0   Min.   :  2
19           ##  1st Qu.:12.0   1st Qu.: 26
20           ##  Median :15.0   Median : 36
21           ##  Mean   :15.4   Mean   : 43
22           ##  3rd Qu.:19.0   3rd Qu.: 56
23           ##  Max.   :25.0   Max.   :120
24         </pre></div>
25       </div></div>
26   </body>
27 </html>
```

`changes()`

The function merges *changes* in the file, `test-changes.txt`, into the final document.

The *old* content, that has been edited, is replaced by the *new* content, in `test-changes.txt`.

The default suffix of the output file is `.save.html`, converted from the input suffix, `.anns.html`.

Listing 2.6 is the code structure of the final document merged with the change. The highlighted text, line 10, shows the modified text.

Listing 2.6: (tidied) `example1.save.html`

```
1  <!DOCTYPE html>
2  <html>
3  <head></head>
4    <body>
5      <p id="savebutton" style="background-color:grey; text-align:
           center">
6        <button onclick="savechanges()">save</button>
7      </p>
8      <h1 contenteditable="true" id="Editor-1">Example</h1>
9      <p contenteditable="true" id="Editor-2">
10 <code>summary(cars)</code>
11     </p>
12     <p class="annotation" style = "background-color:coral">
13     <em>Annotation: The text "speed" was annotated with the
14     message "needs units"</em></p>
15     <div class="chunk" id="unnamed-chunk-1">
16       <div class="rcode">
17       <div class="output">
18         <pre class="knitr r">
19           ##     speed           dist
20           ##  Min.   : 4.0   Min.   :  2
21           ##  1st Qu.:12.0   1st Qu.: 26
22           ##  Median :15.0   Median : 36
23           ##  Mean   :15.4   Mean   : 43
24           ##  3rd Qu.:19.0   3rd Qu.: 56
25           ##  Max.   :25.0   Max.   :120
26         </pre></div>
27       </div></div>
28   </body>
29 </html>
```

Figure 2.5 shows how the *annotated* and *changed* final document, `example1.save.html` looks like in a web browser.

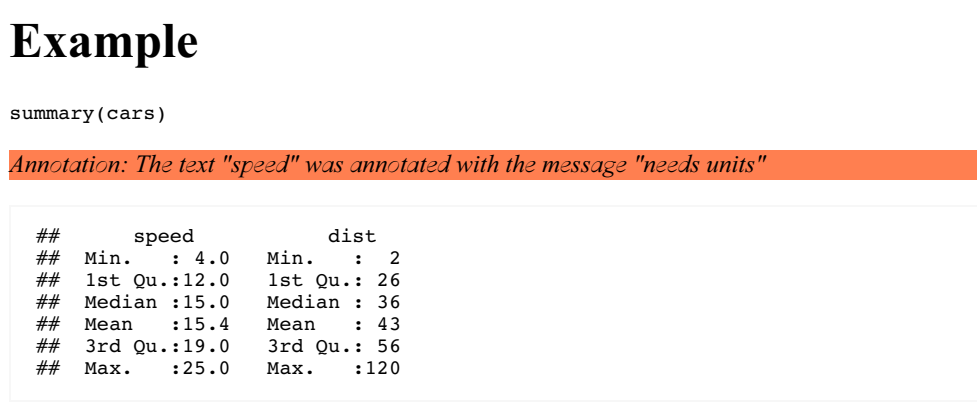


Figure 2.5: `example1.save.html`

`rip()`

The function inverts final HTML-based documents, merged with changes and annotations, back into R HTML-based source documents.

More specifically, all unwanted contents added from the previous steps are removed. These include; the HTML `scripts` and `styles` added by `knit()` and `snap()`, the HTML attributes added by `snap()` and all R-created sections from `knit()`. The copies of R Code Chunks are reverted back to their original R HTML syntax, in place of the removed R-created sections, to become reproducible.

The default suffix of the output file is `.return.Rhtml`, converted from the input suffix, `.save.html`.

Listing 2.7 is the code structure of the inverted document, `example1.return.Rhtml`.

Listing 2.7: `example1.return.Rhtml`

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 </head>
5   <body>
6     <h1>Example</h1>
7     <p>
8 <code>summary(cars)</code>
9     </p>
```

```
10      <p class="annotation" style = "background-color:coral">
11      <em>Annotation: The text "speed" was annotated with the message
            "needs units"</em>
12      <!--begin.rcode echo=FALSE
13      summary(cars)
14      end.rcode-->
15    </body>
16  </html>
```

Rather than manually comparing the original source and inverted documents, we can use `diff` utility in UNIX system. The command line below is to compare the original source file, `example1.Rhtml` and the inverted source file, `example1.return.Rhtml`:

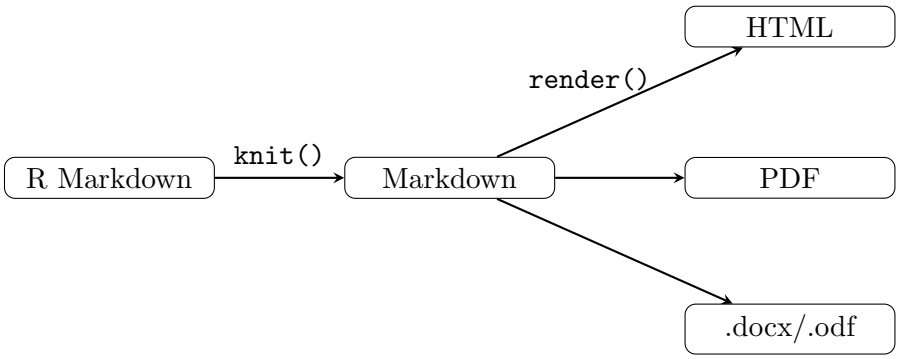
```
$ diff example1.Rhtml example1.return.Rhtml
```

Listing 2.8 is the output from the previous `diff` command. It says line 7 of the original source document (Listing 2.1) is changed with the lines 7–11 of the inverted file (Listing 2.7).

Listing 2.8: Output from `diff`

```
1  7c7,11
2  <     <p>Summary of the cars data:</p>
3  ---
4  >     <p>
5  > <code>summary(cars)</code>
6  >     </p>
7  >     <p class="annotation" style = "background-color:coral">
8  >     <em>Annotation: The text "speed" was annotated with the
       message "needs units"</em>
```

Phase 2: R Markdown



3.1 Introduction

Phase 1 workflow is strictly limited to work on R HTML-based documents in conjunction with **knitr**. It is of our interest to generalise the workflow so that a similar workflow can be applicable on other formats.

The format we have chosen for the second phase is R Markdown. There are two main reasons behind our choice; R Markdown is popular and good tools exist for it.

Popularity

R Markdown is becoming popular amongst many, which makes it a strong candidate for the second phase of this project. There are two main reasons behind the popularity; its syntax is simple and easy for increased usability and supports flexible conversion into different document formats.

R Markdown-based documents can be flexibly converted into different document formats, such as PDF, HTML and word (ODF/DOCX). Apart from adding on its popularity, what this flexibility allows us to implement in the workflow is the ability to produce final documents in different formats. For example, a final document can be converted to HTML and PDF, where the former can be published on the Web while the latter can be suitable for printing.

Tools for R Markdown

The latest version, R Markdown v2, has many improvements, one of which, for example, is the support for interactive document generation using **Shiny** (REF). This is a particularly useful feature in regards to the current global trend.

With the vast advancement in technology, the world demands evidence, often through means of visualising, before acceptance and acknowledgement. In the field of statistics, a particularly effective solution to this demand is via data visualisation, whose effectiveness can be exponentially enhanced by *interactivity*. An interactive visualisation method provides fun and exciting ways to visualise data through which client perception and learning can be greatly improved.

R Markdown's support for **Shiny**, as well as other improvements, is something that is very promising in securing its already popular usage. It also suggests that R Markdown is relatively active and responsive to meet global demands, thus giving us a reason to choose this document format for phase 2.

Demonstration

Listing 3.1 is the internal code structure of a simple R Markdown document, `example2.Rmd`. The highlighted text, lines 6–8, is a simple R Code Chunk.

Listing 3.1: `example2.Rmd`

````
1  ---
2  title: ""
3  output:
4    html_document:
5      toc: false
6  ---
7
8  Example
9  ========================
10
11 Summary of the cars data:
12
13 ```{r}
14 summary(cars)
15 ```
````

There is a crucial difference between the two document generation processes of R HTML and R Markdown. R HTML-based documents are knit-

ted *directly* into HTML, whereas R Markdown-based documents are knitted *intermediately* into Markdown, which is then *rendered* into a final format. Because there are *two* stages involved in one complete document generation process for R Markdown, the function `knit()` has to be used twice; firstly on the source R Markdown document and secondly on the knitted Markdown document. The difference is significant but can be overlooked. **Knitr** implements the use of the function, `render()`, in **rmarkdown** (REF) package which knits and renders in a single command.

The R code below calls `render()` to generate the HTML-based final document, `example2.html`:

```
rmarkdown::render("example2.Rmd", output_format="html_document")
```

Listing 3.2 is the code structure of the final document generated from the source document. The original R Code Chunk in lines 6–8 of Listing 3.1 is processed to produce the result seen in lines 15–21 of Listing 3.2. In general, the HTML document generated from R Markdown has more content, such as `meta` elements, than the one from R HTML. Note that there are more than one `meta`, `script` and `style` elements inside `head` but these are limited to one each for readability.

Even though the two final documents seen in Listings 2.2 and 3.2 are both in the HTML format, they are structurally quite different. Markup attributes such as `class="container-fluid main-container"` in line 10 of Listing 3.2 is different to `class="chunk"` markup in line 9 of Listing 2.2.

Listing 3.2: (tidied) `example2.html`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta ... />
5    <script> ... </script>
6    <style> ... </style>
7  </head>
8    <body>
9      <style type="text/css"> ... </style>
10     <div class="container-fluid main-container">
11       <div id="example" class="section level1">
12         <h1>Example</h1>
13         <p>Summary of the cars data:</p>
14         <pre><code>
15           ##     speed          dist
16           ##  Min.   : 4.0   Min.   :  2
17           ##  1st Qu.:12.0  1st Qu.: 26
```

```
18          ##  Median :15.0  Median : 36
19          ##  Mean   :15.4  Mean   : 43
20          ##  3rd Qu.:19.0  3rd Qu.: 56
21          ##  Max.   :25.0  Max.   :120
22        </code></pre>
23      </div>
24    </div>
25    <script> ...bootstrap table styles... </script>
26    <script> ... mathjax ... </script>
27  </body>
28 </html>
```

Figure 3.1 shows what `example2.html` will look like in a web browser. Although nearly identical, we can see minor differences between the two final documents, seen in Figures 2.1 and 3.1, such as different font styles for the headings and spacing used for the R Code Chunk output. The differences are due to different `styles` used in both cases.

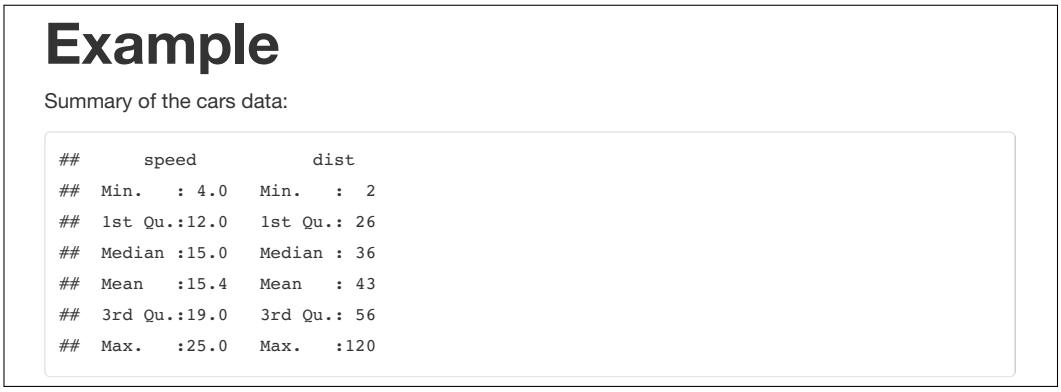


Figure 3.1: `example2.html` in a browser

3.2 Overview

Three additional steps are introduced to phase 2 workflow as a pre-processing stage. Once the source document is *readied* by the three steps of pre-processing, it becomes reproducible and invertible.

Some of the function names are kept identical to those in phase 1 workflow to minimise confusion and maintain consistency. But it must be noted that there are extra features in these functions that work *exclusively* for R Markdown.

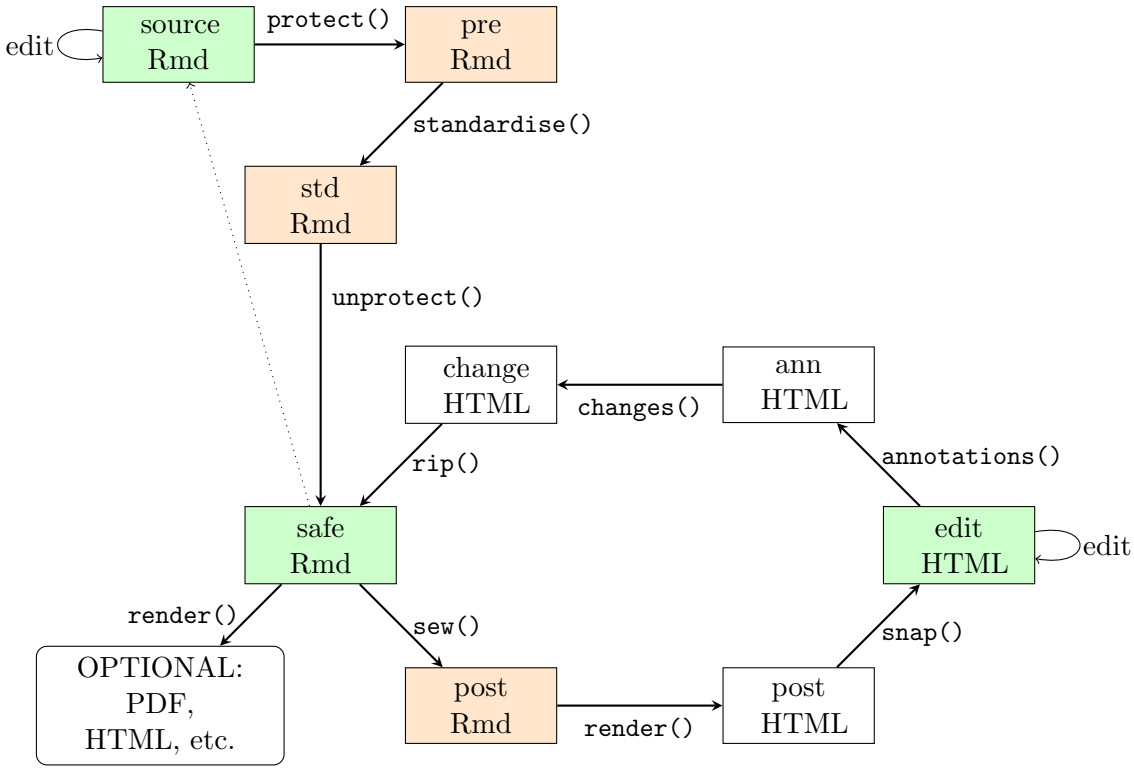


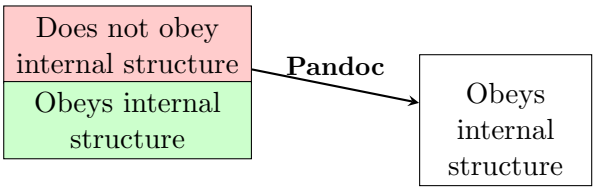
Figure 3.2: Revised reproducible, invertible workflow

Invertibility of Pandoc

Pandoc does not support flawless conversion between document formats, that is, the source document format is not the same as the final document format.

There is an internal structure in **Pandoc** that determines the structure of the output documents. This means that whether the source document obeys

the internal structure or not, **Pandoc** will always produce a final document that obeys the internal structure. This is best understood with a simple diagram: (**Fix diagram**)



We have decided to use the internal structure as a means of *standardising* the source document so that any subsequent conversion by **Pandoc** does not interfere with its structure. The structural aspect of standardised documents will be consistent throughout the workflow.

The standardisation step is carried out by the function, `standardise()` with the use of **Pandoc**.

Loss of information

R Markdown has syntax for specifying metadata, such as author, title and date information. An example of a metadata section is as below:

```
---
title: "Example"
output:
  html_document:
    toc: true
    theme: united
---
```

Metadata sections are *exclusive* to R Markdown. As a result, **Pandoc** discards them in its output documents. This means that the metadata information cannot be carried over to subsequent steps after the standardisation step. **Pandoc** also interferes with R Code Chunks in R Markdown, because R Code Chunks are not known to **Pandoc**. The solution we have chosen is to implement a pre-processing step to protect the metadata sections and R Code Chunks.

The usual delimiters, `---`, for metadata are changed to `<!--rmd_metadata` and `rmd_metadata--> rmd-rmLines`. The protected version of the previous metadata example is as below:

```
<!--rmd_metadata
title: "Example"
output:
```

```
  html_document:
    toc: true
    theme: united
rmd_metadata--> rmd-rmLines
```

R Code Chunks are protected by `<!--begin.keepcode` and `end.keepcode-->` `rmd-rmLines` delimiters.

The function, `protect()`, carries out the protection of metadata and R Code Chunks.

Conversion of formats

Pandoc is used to convert HTML into Markdown, as well as in standardising source documents. This conversion by **Pandoc** is *one* of the tasks carried out by the function, `rip()`.

3.3 Functions detail

`protect()`

The function protects metadata sections and R Code Chunks in the source document.

Metadata sections are delimited by `<!--rmd_metadata` and `rmd_metadata--> rmd-rmLines`.

R Code Chunks are delimited by `<!--begin.keepcode` and `end.keepcode--> rmd-rmLines`.

The use of `rmd-rmLines` is to correct for **Pandoc**'s tendency of removing empty lines after the protected lines.

The default suffix of the output file is `.pre.Rmd`, converted from the source suffix, `.Rmd`.

Listing 3.3 is the code structure of the protected document, `example2.pre.Rmd`.

Listing 3.3: `example2.pre.Rmd`

````
<!--rmd_metadata
title: ""
output:
  html_document:
    toc: false
rmd_metadata--> rmd-rmLines

Example
=========================

Summary of the cars data:

<!--begin.keepcode```{r, echo=FALSE}
summary(cars)
```end.keepcode--> rmd-rmLines
````

`standardise()`

The function, `standardise()`, calls **Pandoc** to *standardise* the protected source document. In this function, `system()` is used to invoke the command line necessary for calling **Pandoc**.

The default suffix of the output file is `.std.Rmd`, converted from the input suffix, `.pre.Rmd`.

Listing 3.4 is the code structure of the standardised document, `example2.std.Rmd`.
````

A couple of things to note is that the equal signs in line 10, which correspond to R Markdown's syntax for specifying a heading, is shortened and the delimiters, `rmd-rmLines`, are broken into the next lines in lines 7 and 17. These are some of the examples of **Pandoc**'s internal structure.

Listing 3.4: `example2.std.Rmd`

```
1  <!--rmd_metadata
2  title: ""
3  output:
4    html_document:
5      toc: false
6  rmd_metadata-->
7  rmd-rmLines
8
9  Example
10 =======
11
12 Summary of the cars data:
13
14 <!--begin.keepcode```{r, echo=FALSE}
15 summary(cars)
16 ```end.keepcode-->
17 rmd-rmLines
```

`unprotect()`

The function removes the protection added by `protect()`.

The special delimiters used to protect metadata and R Code Chunks are reverted back to their normal R Markdown syntax.

The pushed down lines for `rmd-rmLines` delimiters are simply removed.

The default suffix of the output file is `.safe.Rmd`, converted from the input suffix, `.std.Rmd`.

Listing 3.5 is the code structure of the invertible document, `example2.safe.Rmd`, that is fully prepared to enter phase 2 invertible workflow.

Listing 3.5: `example2.safe.Rmd`

```
1  ---
2  title: ""
3  output:
4    html_document:
5      toc: false
6  ---
```

```
7
8  Example
9  =======
10
11 Summary of the cars data:
12
13 ```{r, echo=FALSE}
14 summary(cars)
15 ```
```

We can use the `diff` command below to compare the source document, `example2.Rmd`, and the invertible source document, `example2.post.Rmd`:

```
$ diff example2.Rmd example2.safe.Rmd
```

Listing 3.6 is the output form the previous `diff` command. The only difference between the two source documents is line 9, where the number of equal signs is reduced by the internal structure of **Pandoc**.

Listing 3.6: Output from `diff`

```
1 9c9
2 < ========================
3 ---
4 > =======
```

`sew()`

The function carries out a similar pre-processing step as the phase 1 version with one addition; it copies metadata sections (as well as R Code Chunks) and append the copies after their originals.

The default suffix of the output file is `.post.Rmd`, converted from the input suffix, `.safe.Rmd`.

`render()`

The function is directly from the R package, `rmarkdown`. This function is what **knitr** uses to convert R Markdown-based source documents into final documents.

The default suffix of the output file is `.post.html`, converted from the input suffix, `post.Rmd`.

`snap()`

The function carries out conceptually the same tasks as the phase 1 version; add pieces of HTML and JavaScript code to the final document, and add HTML attributes to editable nodes but *differently*.

Two final (HTML-based) documents each generated from `knit()` and `render()` are structurally different; the latter is richer with more contents such as pieces of JavaScript code for **MathJax** (REF) library and using Bootstrap tables.

These HTML `scripts` added by `render()` interfere with the JavaScript code added by `snap()`, which means **CKEditor** and **Annotator** do not work. The solution implemented in phase 2 `snap()` is to

`annotations()`

`changes()`

`rip()`