Supporting the Executability of R Markdown Files

by

MD ANAYTUL ISLAM

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE FACULTY OF GRADUATE STUDIES
OF LAKEHEAD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

2023
Lakehead University
Thunder Bay, Ontario, Canada

ABSTRACT

R Markdown files are examples of literate programming documents that combine R code with results and explanations. Such dynamic documents are designed to execute easily and reproduce study results. However, little is known about the executability of R Markdown files which can cause frustration among its users who intend to reuse the document. This thesis aims to understand the executability of R Markdown files and improve the current state of supporting the executability of those files.

Towards this direction, a large-scale study has been conducted on the executability of R Markdown files collected from GitHub repositories. Results from the study show that a significant number of R Markdown files (64.95%) are not executable, even after our best efforts. To better understand the challenges, the exceptions encountered while executing the files are categorized into different categories and a classifier is developed to determine which Markdown files are likely to be executable. Such a classifier can be utilized by search engines in their ranking which helps developers to find literate programming documents as learning resources. To support the executability of R Markdown files a command-line tool is developed. Such a tool can find issues in R Markdown files that prevent the executability of those files. Using an R Markdown file as an input, the tool generates an intuitive list of outputs that assist developers in identifying areas that require attention to ensure the executability of the file. The tool not only utilizes static analysis of source code but also uses a carefully crafted knowledge base of package dependencies to generate version constraints of involved packages and a Satisfiability Modulo Theories (SMT) solver (i.e., Z3) to identify compatible versions of those packages. Findings from this research can help developers reuse R Markdown files easily, thus improving the productivity of developers.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my supervisor, Dr. Muhammad Asaduzzaman, for his continual advice, insightful recommendations, encouragement, and exceptional patience throughout this thesis effort. Without him, this thesis would not have been possible.

I would like to express my gratitude to Dr. Zubair Fadlullah (Western University) and Dr. Garima Bajwa (Lakehead University) for their contributions to my thesis committee. Their guidance, suggestions, evaluation, and insightful remarks helped this thesis to be completed successfully.

I appreciate Dr. Shaowei Wang (University of Manitoba) for generously dedicating his time and sharing his knowledge, which has enhanced the quality of my work.

I also want to express my gratitude to the Lakehead University Department of Computer Science for their kind assistance in the form of scholarships, awards, and bursaries, which allowed me to focus more intently on my thesis work.

I want to express my gratitude to all of my friends and fellow Department of Computer Science staff who have supported me along this journey. I am especially grateful to Dr. Sheikh Moniruzzaman, Subrata Das, Pronab Ghosh, and Shouvik Paul.

I want to convey my gratitude to my family and relatives, particularly expressing thanks to my mother, Mst. Auleda Begum, and my brothers, Md. Wasequl Islam Anup and Md. Rezoyanul Islam Ananno.

I would like to thank my friends in Thunder Bay for their constant support and encouragement. I would especially like to thank Mostansir Nayeem, Shofi Ahmed, Ragib Raonok, Prithu Aldrin Costa, Hasib Kamal, and Nahiyan Al Mahmood.

I am grateful to everyone for their participation in this thesis, even if they are not named specifically in this thesis. Your participation has been really helpful to my growth, both personally and professionally.

This thesis is dedicated to the memory of my father, Md Saiful Islam, who has consistently served as my source of inspiration. He is no longer with us. May Allah grant him eternal peace in Jannah.

PUBLICATIONS

Parts of this thesis have been submitted for peer-review or accepted to be published:

- M. A. Islam, M. Asaduzzaman, and S. Wang, **"On the Executability of R Markdown Files"**, accepted to be published in the 21st International Conference on Mining Software Repositories, 2024. (Chapter 3)

Contents

List of Tables

List of Figures

Chapter 1

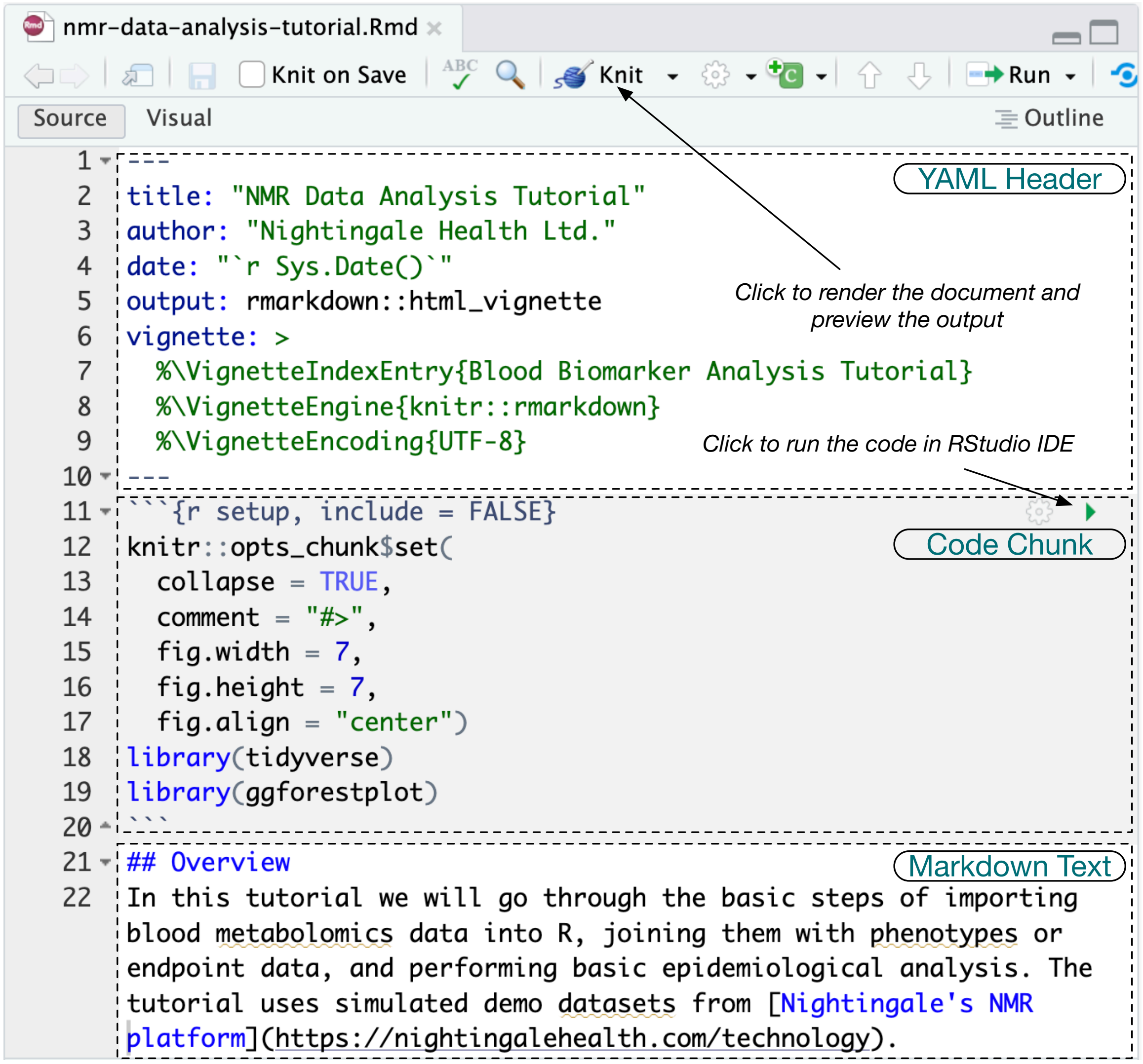
Introduction

This chapter provides a short introduction to the thesis. Section 1.1 describes the motivation for this thesis. Section 1.2 explains the research problem. The contributions of this thesis are described in Section 1.3. Finally, Section 1.4 discusses the outline of the remaining chapters.

1.1 Motivation

The literate programming paradigm was developed by Donald Knuth, who emphasized explaining the process that goes into writing code rather than just writing code to accomplish a task [1]. This is accomplished by writing documentation which is dotted with computation results and code. The need to share easily comprehensible and replicable data analysis results has led to an increase in the popularity of the literate programming paradigm. It is simpler to reuse because the code's design is self-documenting and offers a clear style throughout the file. This documentation offers programmers a better understanding of where updates are needed as well as what to do during any extension. R Markdown files are illustrations of literate programming documents used by researchers to share study results.

An R Markdown file consists of three different components. These are metadata, text, and code. The metadata (known as the YAML metadata or YAML header) consists of a set of key-value pairs that provide information about the document and rendering configuration. The metadata is specified within a pair of three dashes (i.e., —). The text component (also known as prose or narrative) contains the markdown text that describes the needed operations to be performed by the computer. A code component can be a code chunk or an inline R code that can be executed to get the output. Figure 1.1 shows an example of an R Markdown file. R Markdown files depend on the Knitr package for the purpose of


nmr-data-analysis-tutorial.Rmd
Knit on Save
Knit
Run
Source
Visual
Outline
YAML Header
Click to render the document and preview the output
Click to run the code in RStudio IDE
Code Chunk
Markdown Text

~~~
---
title: "NMR Data Analysis Tutorial"
author: "Nightingale Health Ltd."
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Blood Biomarker Analysis Tutorial}
  %\VignetteEngine{knitr::rmarkdown}
  %\VignetteEncoding{UTF-8}
---
```{r setup, include = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>",
  fig.width = 7,
  fig.height = 7,
  fig.align = "center")
library(tidyverse)
library(ggforestplot)
```
## Overview
In this tutorial we will go through the basic steps of importing
blood metabolomics data into R, joining them with phenotypes or
endpoint data, and performing basic epidemiological analysis. The
tutorial uses simulated demo datasets from [Nightingale's NMR
platform](https://nightingalehealth.com/technology).
~~~

Figure 1.1: An example of an R Markdown file.

rendering [2]. The Knitr package executes the code chunks, generates the output, and combines that together with the narratives. When we use the render function in R Markdown, it takes the initial .Rmd file and execute all of the code chunks, creating a new markdown (.md) document that includes both the code and its output. This markdown file is then processed by pandoc to produce the final formatted document (e.g., pdf, word, and html5). Although this process may seem complex, R Markdown simplifies the process by combining all of the necessary steps into a single render function.

To reproduce a study result, it is necessary to first understand the process that transforms the input data into the final result in successive steps. R Markdown files make it easier to share reproducible study results because the result, code, and narratives explaining the computational steps (i.e., the process) and explanation of the result are all in one place. Although R Markdown files are becoming increasingly popular for sharing reproducible results from research, not much is known about how executable they are. It's a vital initial step because the ability to run these files is essential for replicating the results of the research. A solid understanding of the difficulties involved in executing R Markdown files is crucial because it can assist us in developing potential solutions for the problem.

1.2 Research objectives and significance

An empirical study is performed to determine how executable are R Markdown files. This is done by collecting 191,966 R Markdown files from GitHub repositories that are publicly available. Each R Markdown file is executed in a separate execution environment to determine its executability status. In case a file cannot be executed, the error messages are collected for further investigation. A quantitative study is conducted that compares executable and non-executable groups of files to understand their differences. If the two groups of files are different from each other we can leverage features collected from those files to build a classifier to predict their executability without even executing them. A qualitative study is performed on the set of non-executable R markdown files to identify the reasons that prevent their executability. Thus, the first part of this thesis focuses on the following two research questions.

- **How executable is R Markdown files? Are there differences between the executable and non-executable groups?** Despite our best efforts, 64.95% of R Markdown files are not executable, according to the results of our study. By considering different groups of features, we examine the differences between the two groups of files (executable vs. non-executable). The findings of the investigation reveal that the two groups of files differ significantly from each other. As an illustration, the non-executable group of files depends on more libraries than the executable group does.

- **What are the reasons that hinder the executability of R Markdown files?** We find three primary reasons that prevent the executability of R Markdown files. These are a) Missing libraries (69.76%) b) Missing data (19.12%) c) Erroneous Code (11.12%).

Since a large number of R Markdown files available online suffer from executability issues, not all of those files can be reused easily. Users who wish to search for R Markdown files online to support their development tasks may be unaware of the issues that prevent the executability of R Markdown files. Existing search engines also do not consider the executability of those files in their ranking and it is also not feasible to test the executability by executing those files because of resource and time constraints. As a result, we develop classifiers taking into account characteristics associated with R Markdown files across seven different dimensions. An extensive evaluation using a large number of R Markdown files reveals that the classifier achieves a median AUC of 83.02%, indicating the feasibility of leveraging such classifiers.

Results from our qualitative analysis revealed that library-related issues are the most dominant factor that prevents the executability of R Markdown files (e.g., missing libraries, and incompatible library versions). Existing package management tools are not very helpful in solving these problems. For example, such tools are not able to able to identify required versions of packages or libraries. While it is possible to install the latest versions of required packages but latest versions of packages are not always used by R Markdown files. Moreover, users of R Markdown files do not always use package management tools and document the versions of required packages. Thus, a tool is developed that utilizes static analysis of source code and a carefully crafted knowledge base of package dependencies to generate version constraints of involved packages. The tool then uses SMT solvers (i.e., Z3 [3]) to identify compatible versions of those packages. The tool can also able to identify missing files and syntax errors in code.

1.3 Contributions of the Thesis

The contributions of the thesis are as follows:

- A large-scale empirical study to understand how executable are R Markdown files. A quantitative comparison is performed to understand the differences between executable and non-executable groups of files.

- A qualitative analysis of non-executable R Markdown files to understand the reasons that prevent the executability of those files.

- The development and evaluation of classifiers that can predict the executability of R Markdown files.

- A knowledge base of package dependencies created by mining R packages stored in the Comprehensive R Archive Network (CRAN [4]). CRAN is the central repository that contains different versions of packages along with their documentation.

- A tool that can automatically identify executability issues in R Markdown files and provide solutions to fix those issues.

1.4 Outline of the Thesis

This chapter outlines the research problems and briefly describes the contributions of the thesis to address those problems. The remaining chapters of this thesis are organized as follows:

- Chapter 2 describes prior studies related to the thesis.
- Chapter 3 provides a detailed description of a study that investigates the executability of R Markdown files.
- Chapter 4 presents a technique that can automatically identify executability issues in R Markdown files and assists developers in fixing those issues.
- Chapter 5 concludes the thesis by summarizing research contributions and providing future research directions.

Chapter 2

Related Work

The foundation of this thesis is laid by the prior studies on executability, reproducibility and dependency management in the literature, which are summarized in this chapter along with the terminologies related to this thesis.

2.1 Executability

To check executability, Yang et al. [5] examined the Python snippets that are available in Stack Overflow. They found that among the available snippets, 75% are parseable and only 25% are executable. Similar to this work, Eric et al. [6] investigated the feasibility of executing Python gists. A gist is a code snippet that can be shared with others. Using python:2.7.13 and python:3.6.5 images as the base, they tried running gists in isolated Docker containers to conduct a baseline analysis. A gist that runs without any errors is considered an executable gist. On the other hand, the name of the error that was raised during the execution is used to code any unsuccessful gist. Their findings indicate that the majority of the gists are not directly executable in a default Python environment, primarily due to insufficiently configured environments. Although in certain cases a naive inference algorithm (collect imported library names and install the latest version of those libraries) can automatically recover the correct application environment configurations, it is not effective in most cases. They faced mainly two challenges while configuring environments. One of them is identifying dependencies with non-obvious names and the other one is installing dependencies with transitive dependence on system modules. However, they introduced a technique, called Gistable, which can automatically configure and execute approximately 5,000 publicly available Python gists. Each gist has its own docker file, based on the python:2.7.13 image containing gist and its dependencies which ensures re-executibility on their docker image. Our study differs in various ways from the research of Yang et al. [5] and Eric et al. [6]. Comparing our work to earlier studies, we found that Yang et al. [5] focused on Python code, whereas Eric et al. [6] examined the executability of publicly accessible Python gists. Our approach examines the executability of R Markdown files. While Eric et al. [6] determined library names from import statements and attempted installation of those libraries, Yang et al. [5] didn't consider any dependency installation. On the other hand, we utilized the renv tool [7] to automatically identify and install dependencies, guaranteeing a more accurate and efficient evaluation of R Markdown files' executability.

Apart from that, Samim et al. [8] conducted, a comprehensive mixed-methods investigation involving an empirical analysis of tutorials sourced from diverse online platforms. They used a naive execution strategy, in other words, copy and paste strategy, to establish a baseline executability level. Then they examined the code blocks with tutorials and formulated a set of annotations to enhance executability. Although there were still a significant number of failures, these annotated tutorials with Docable resulted in an improved executability rate. Upon qualitative examination of tutorial failures and holding peer debriefing sessions with tutorial creators, various issues were identified that hindered successful execution. These problems affected tutorial quality in addition to providing accessibility challenges. The research proposes various approaches for enhancing software tutorials, such as offering easily accessible substitutes for tutorial participants and introducing automated testing to ensure the continuous quality of the tutorials. Monir et al. [9] proposed a scalable Docker-based evaluator designed for determining whether Python code snippets are executable or not. Surprisingly, their research result indicates that executability does not significantly impact the likelihood of an answer whether it is excepted. They argue that explanations are more important than code that can be executed quickly. However, they mark a noteworthy increase in executability for code snippets which was referenced in GitHub projects, implying that practical usability is more valued in such contexts. Additionally, Horton at al. [10] describes a novel approach for automatically inferring and configuring a computing environment capable of executing any Python code snippet without encountering import errors. Their process involves constructing a knowledge base of known Python packages, leveraging static and dynamic analysis, and association rule mining to find package dependencies. The authors introduced DockerizeMe, which generates an environment configuration in the form of a Dockerfile for the Docker container system which can be easily reproducible. According to their experimental results, DockerizeMe achieves a 30% improvement in environment configuration efficiency over a baseline method.

2.2 Reproducibility

To the best of our knowledge, there has been no large-scale study conducted on the quality and executability of R Markdown files. In contrast, numerous studies have explored the quality and reproducibility of Jupyter Notebooks which is similar to R Markdown files because both are designed to support literate programming. As an illustration, Pimentel et al. [11] did a reproducibility study, identified the best and worst practices for reproducibility in Jupyter Notebooks for the Python language, and suggested ways to enhance reproducibility. However, the authors acknowledge the need for further work, including the development of tools that can identify hidden states, out-of-order cells, and undeclared dependencies in order to address bad practices.

However, Rudolf at al. [12] thoroughly examined publicly available workflows, particularly those that are written in the Taverna 2 language, which is used for creating workflow, and hosted on the myExperiment platform [13]. Their investigation was focused on finding weaknesses in different Taverna processing components. They conducted a static analysis of processor usage and configuration. Afterwards, automated attempts were performed to execute each workflow, with manual inspections added in case of issue. Based on these observations, they presented a series of suggestions to increase workflow reproducibility. Their results point to several ways to address problems, from improving the publishing and monitoring procedures to fine-tuning workflow definitions and insertion of auxiliary metadata such as Research Objects which allows for a semantic description and linking of other objects employed in the research investigation. Interestingly, a large percentage of workflows had execution issues, which were frequently caused by seemingly insignificant errors such as the absence of example values for input parameters and missing libraries for Java programs. Saikat et al. [14] examined the reproducibility of issues in 400 posted questions to identify how software developers use Stack Overflow to solve problems. To find out how replicable the code is, the researchers spent 200 man-hours parsing, compiling, executing, and even carefully examining the code segments. The observation reveals a two-fold outcome: First, 22% of the code segments totally fail in replication attempts, and second, 68% of the code segments require adjustment to reproduce issues that have been reported. After investigating the causes of non-reproducibility, they proposed evidence-based recommendations for writing effective code examples in Stack Overflow questions. In addition, they found that questions with reproducible issues are three times more likely to have accepted answers than questions with non-reproducible issues.

Ana et al. [15] used publicly available replication datasets which were hosted on the Harvard Dataverse repository to examine research code quality and performance. they conducted their study by examining around 2000 replication datasets and 9000 unique R files from 2010 to 2020. The investigation determines the code's usability by executing it in a clean runtime environment to detect common coding errors. Their analysis shows that 74% of the R files encountered errors during the initial execution, while 56% of the issues were found even after code cleaning, emphasizing the importance of strong coding practices. Our work is different from this work in several ways. Firstly, they considered the replication dataset hosed in the Harvard Dataverse repository while we considered R Markdown files that are publicly available in Github. Our dataset is bigger than theirs. Secondly, they tried the installation of 200 common libraries and in the data cleaning part, they tried to install the library which is not installed. On the contrary, we used renv to identify the all dependencies and install them. Finally, they didn't predict the executability of a file while developed classifiers to predict the executability of R Markdown files.

2.3 Dependency Conflict

German et al. [16] point out challenges in managing dependencies, focusing on problems pertaining to the downloading, building, and satisfying interdependent artifacts that might not have explicit documentation. They introduce a framework for classifying dependency types and introduce a technique to create and visualize an interdependency graph for packages. Horton et al. [17] presented Version 2 (V2), an advanced version of DockerizeMe [10] that goes above and beyond by adding the ability to intelligently analyze dependency versions and validate multiple environments. They mentioned that despite code snippets' abundance, they face obstacles to reuse because of a lack of accompanying environment configuration. The absence of active maintenance aggravates the issue, leading to discrepancies between the code snippet and the most recent configuration. Recent efforts emphasize the vital role of validating and detecting out-of-date code snippets for effective reuse. However, distinguishing between correctness and mere obsolescence is a difficult task. In the best-case scenario, well-documented breaking changes facilitate manual detection of out-of-date API usage, whereas, in the worst-case scenario, exhaustive searches through previous dependency versions are required. With the release of V2, their method seeks to detect isolated cases of configuration drift, especially if the snippet makes use of an API that encountered a breaking change. Every drift instance that has been found is characterized by a validation failure along with a configuration patch containing the required dependency version changes to fix the underlying problem. By leveraging feedback-directed search, V2 reduces the number of configurations that need to be validated by methodically exploring the possible configuration space. When V2 is used on a collection of publicly available Python snippets, it effectively detects 248 cases of configuration drift. After that, Wang et al. [18] performed an empirical investigation to assess the impact of dependency-resolving strategy, which revealed changes in one DC (Dependency Conflict) manifestation pattern (Pattern A) while another (Pattern B) remained consistent. They found inefficiency issues in Pattern A and time/space waste in Pattern B. To address these issues, They developed smartPip, a tool that uses a knowledge base for Pattern A and improves the virtual environment solution for Pattern B. SmartPip outperformed existing tools in open source projects, obtaining speedups of 1.19X - 1.60X for Pattern A while decreasing storage space by 34.55% - 80.26%, with improvements of 2.26X - 6.53X for Pattern B compared to the built-in Python virtual environment (venv).

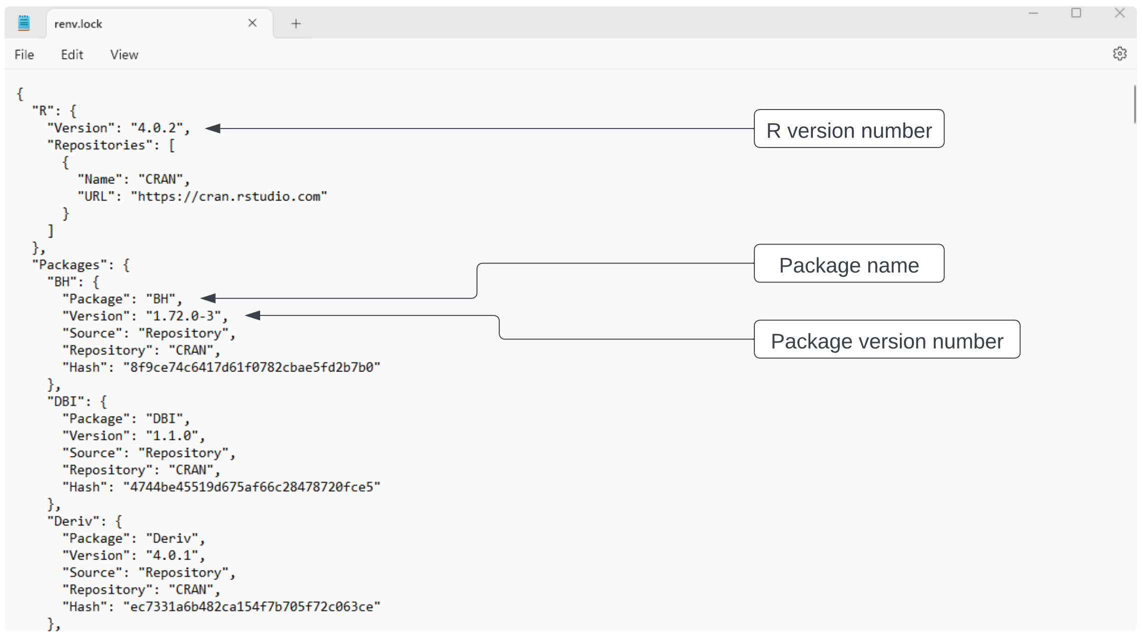
In a similar spirit, Lungu at al. [19] identify dependencies extending beyond specific projects within a software ecosystem. By introducing a model that can capture inter-project dependencies, they contribute to a deeper understanding of dependencies within the context of software development.

2.4 Dependency Management

Among the R package management tools, renv [7], packrat [20], devtools [21], checkpoint [22], and pacman [23] are management tools that offer different functionalities to meet the requirements of different users in the R programming community. The comparison table (Table 2.1) clarifies important characteristics. Additionally, we provide an example of a lock file in figure 2.1 to demonstrate how they are stored. To begin with, tools that support lockfiles include renv [7], packrat [20], and checkpoint [22]. These tools ensure reproducibility by keeping track of package versions. Renv [7] and Packrat [20] are excellent in dependency isolation because they allow users to create environments that are specific to their projects, which prevents dependencies from interfering with one another. Renv [7], Checkpoint [22], and Pacman [23] all have the handy feature of automatically installing dependencies, which makes setup easier. Renv [7] and Packrat [20] support project-specific configurations, which allow users to install and maintain versions of packages specific to a project. Notably, automatic snapshot creation is an extra benefit offered by renv [7] and packrat [20], which further streamlines the process. The selection of a tool is contingent upon the particular requirements and inclinations of the R user, as each tool offers distinct advantages.

Table 2.1: Comparison of R Package Management Tools.

Feature	**renv**	**packrat**	**devtools**	**checkpoint**	**pacman**
Lockfile	Yes	Yes	No	Yes	No
Dependency Isolation	Yes	Yes	No	Yes	No
Dependency Installation	Automatic	Manual	Manual	Automatic	Manual
Project-Specific	Yes	Yes	No	Yes	No
Automatic Snapshots	Yes	Yes	No	No	No



```
{
  "R": {
    "Version": "4.0.2",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://cran.rstudio.com"
      }
    ]
  },
  "Packages": {
    "BH": {
      "Package": "BH",
      "Version": "1.72.0-3",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "8f9ce74c6417d61f0782cbae5fd2b7b0"
    },
    "DBI": {
      "Package": "DBI",
      "Version": "1.1.0",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "4744be45519d675af66c28478720fce5"
    },
    "Deriv": {
      "Package": "Deriv",
      "Version": "4.0.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "ec7331a6b482ca154f7b705f72c063ce"
    },
```

Figure 2.1: An example of a lock file.

2.5 Restoring Executability

Several techniques have been proposed to address these challenges and enhance reproducibility in Jupyter Notebooks. Osiris [24], the first technique to restore the executability of Jupyter Notebooks, is focused on automatically restoring the execution cell order. Incorrect execution order of notebook cells can prevent the executability of the notebook. Following up on this work, the author of Osiris, Jiawei et al. [24] also presented SnifferDog [25] to restore the executability. Jiawei et al. [25] have taken a major step towards this goal in addressing the challenges of restoring the execution environments of Jupyter Notebooks. They pointed out a common problem wherein dependencies are not stated clearly, which prevents the execution of a notebook. They introduced SnifferDog which overcomes this problem by analyzing imports and usages of APIs in notebooks, matching them with libraries in different versions. SnifferDog then finds library candidates that were used to create the notebook and searches for configurations to make the notebook fully reproducible.

Apart from that to restore the executability of Jupyter Notebooks, Chenguang et al. [26] introduced a technique, called RELANCER, that restores the executability by automatic upgrade of deprecated APIs. It uses an iterative runtime-error-driven approach to detect and resolve individual API problems in almost real-time. Based on runtime error messages, the system uses a machine-learned model to predict which API repairs are required, encompassing updates in API or package names, parameters, or parameter values. RELANCER combines information from API migration examples on GitHub and API documentation for generating a search space for candidate repairs. They also employed a second machine-learned model to rank these candidates. RELANCER outperforms a baseline without its machine learning models, which can only fix 24% of subjects, by restoring execution for 49% of them in a real-time scenario with a 5-minute time limit. However, their result is based on the evaluation of a dataset of 255 notebooks which might not represent the whole scenario for all kinds of data.

In case of R Markdown files, however, such efforts are still lacking. Therefore, a large-scale study exploring the quality and reproducibility of R Markdown files is necessary to identify best practices and develop tools that can enhance their executability. This study will be beneficial to researchers who use R Markdown files for their computational research and can provide them with guidelines on how to enhance the quality and reproducibility of their work.

2.6 Conclusion

In this section, we examine relevant research and delve into background material to strengthen the reasoning behind the thesis. We introduce the notion of executability and discuss several works that deal with this facet. Furthermore, we also briefly discuss research on reproducibility. An explanation of several dependency management solutions is presented after a thorough review of methods for resolving dependence conflicts. Finally, we examine methods for resolving executability-related concerns.

Chapter 3

On the Executability of R Markdown Files

3.1 Introduction

Donald Knuth introduced the literate programming paradigm where the goal is not to write code to complete a task but to focus on explaining the process that leads to the development of that code [1]. This is done by writing documentation interspersed with code and computation results. Literate programming paradigm gains popularity due to the necessity of sharing data analysis results that can be understood and reproduced easily. As the design of the code is self-documenting and a great view of the style is spread all along the file, it is easier to reuse. At the time of any extension, this documentation helps programmers to get an idea about where it needs to be updated and what to do.

R Markdown files are examples of literate programming documents that are used by researchers and data scientists to share their study results. To reproduce a study result, it is important to understand the process that converts the input data to the final result in successive steps. R Markdown files facilitate sharing reproducible study results since the code, result, and narratives that describe the computational steps (i.e., the process) and explanation of the result are located in the same place. Despite the popularity of R Markdown files, we know nothing about the executability of such files. This is important because R Markdown files are designed to share reproducible study results and executing the files is the first step toward reproducing the result. Understanding the challenges encountered while executing those files can help us plan for possible actions to address the issue.

This chapter presents a study to understand the executability status of R Markdown files. Toward this goal, we collected 191,966 R Markdown files from publicly available GitHub projects. To understand the executability status, we execute each Markdown file by separating their execution environment. We collect the error message if a file fails to execute.

We compared executable and non-executable Markdown files across different dimensions to understand the differences between the two groups of files. Next, we conduct a qualitative study to better understand the reasons that prevent the Markdown files from being executed. We first structured our study along with the following two research questions.

- **How executable is R Markdown files? Are there differences between the executable and non-executable groups?** Results from our study show that 64.95% of R Markdown files are not executable despite our best efforts. We investigate the differences between the two groups of files by considering different features. Results from our study show that the two groups of files are considerably different from each other in terms of their characteristics. For example, non-executable group of files depends on more libraries compared to that of the executable group.

- **What are the reasons that hinder the executability of R Markdown files?** Through a qualitative study, we identify three main reasons that hinder the executability of R Markdown files. These are a) Missing libraries (69.76%) b) Missing data (19.12%) c) Erroneous Code (11.12%).

We have found that a large number of R Markdown files suffer from issues that prevent the executability of those files and their exists differences between the two groups of files (i.e., executable vs non-executable). Users who wish to search and reuse online Markdown files may not be aware of these problems. A classifier can warn users if the selected document is highly likely to suffer from executability issues. Such a classifier can also warn the authors of those Markdown files, and suggest guidelines to ensure executability and reproducibility of those documents. Code search engines can also include such a classifier in their ranking to ensure that the top-ranked files are highly likely to be executable. Thus, we build classifiers considering features related to R Markdown files across seven different dimensions. **Evaluation with a large number of R Markdown files shows that such a classifier obtains a median AUC of 83.02**%.

3.1.1 Data Collection

To conduct our experiment, we downloaded the GHTorrent dataset, which was released on March 6, 2021. The dataset contains the metadata about the public repositories in GitHub. We focus on projects with a primary language of "R" because they are more likely to contain R Markdown files. After analyzing the data, we collected a total of 773,512 R projects. We remove those projects that were created through forking or are deleted from GitHub (i.e., they are no longer available) from our analysis. We were left with 383,452 projects after removing forked and deleted projects. This final list was used to clone the projects in our local machine.

We searched for R Markdown files after cloning those target projects locally. Surprisingly, only 16% of those cloned projects contain at least one R Markdown file. After removing those projects that do not contain any R Markdown files, we obtained 59,089 projects that contain 191,966 R Markdown files.

To provide some statistics of our studied dataset, the minimum number of R Markdown files per project is one and the maximum number is 556. Interestingly, a small number of projects in our studied dataset contain the majority of the R Markdown files. For example, we found that 14.89% of projects in our studied dataset contain five or more R Markdown files, which contain 58.07% (i.e., 111,496) R Markdown files.

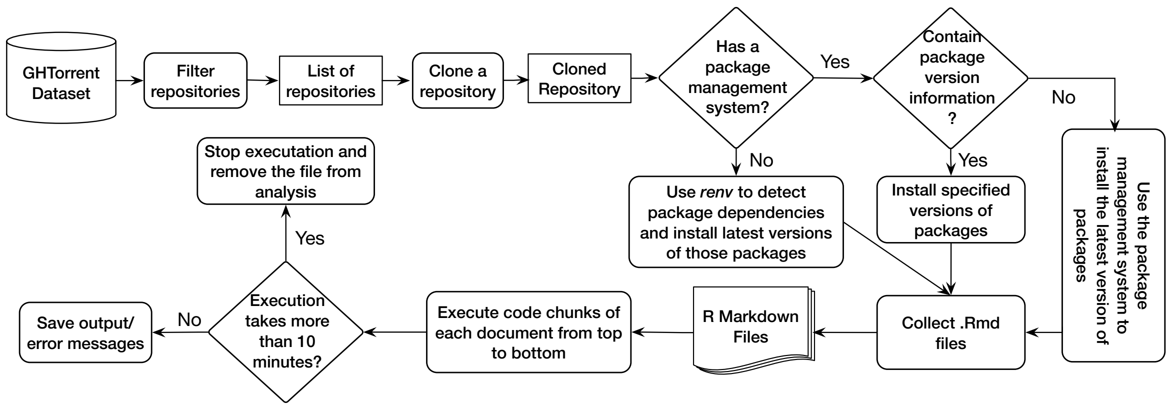

GHTorrent Dataset
Filter repositories
List of repositories
Clone a repository
Cloned Repository
Has a package management system?
Yes
No
Contain package version information ?
Yes
No
Use renv to detect package dependencies and install latest versions of those packages
Install specified versions of packages
Use the package management system to install the latest version of packages
Collect .Rmd files
R Markdown Files
Execute code chunks of each document from top to bottom
Execution takes more than 10 minutes?
Yes
No
Stop executation and remove the file from analysis
Save output/ error messages

Figure 3.1: Steps to execute R Markdown files.

3.1.2 Data Labelling

To conduct our experiment, we need to divide our studied dataset, containing 191,966 R Markdown files, into executable and non-executable groups. Thus, we need to execute each Markdown file and collect their execution status.

An R Markdown file can depend on packages that are external libraries and are not included with base R installation. Two different R Markdown files can depend on different versions of the same package. Thus, the execution of each document should be done in separation so that external packages do not conflict with each other. Thus, we follow the following process to execute R Markdown files.

Figure 3.1 shows the steps we followed to execute R Markdown files. In Appendix A, we provide a step-by-step guide for executing R Markdown files automatically. We created a virtual environment for each project that enables the dependencies of each project to be isolated and avoids conflicts between different versions of the same package. To create a virtual environment and install the necessary R packages, we entered commands (i.e., R functions or scripts) in the command line of a terminal. We used the *subprocess* module of Python to allow those commands to be executed from the Python scripts and collected the output. If a project uses a package manager (e.g., devtools [21], renv [7], packrat [20], checkpoint [22], automagic [27], or jetpack [28]), the necessary packages were installed using those package management tools. To further automate the process of installing dependencies, installation procedures were created for each package management tool, in case they were used. Furthermore, when package management tools were not used, we used the renv package manager to install the necessary dependencies. We used the *renv::init()* function which includes all the project's R packages in the newly created project library. To do this task, it crawls R files within the project to identify the dependencies using the *renv::dependencies()* function.

Once the necessary dependencies have been installed, all the R Markdown files in each R project directory are identified using the *os* and *re* modules in Python. After identifying the R Markdown files, we rendered them using *Rscript*, a command-line tool for running R scripts. Then we used the *subprocess* module in Python to execute the *Rscript* command for each R Markdown file. After completing the rendering process for the R Markdown file, we saved the output for future analysis. After execution of all Markdown files of a project, the virtual environment is deactivated and removed. This frees up system resources and avoids conflicts between different virtual environments.

Around a quarter (25.83%) of R Markdown files in our studied dataset failed to complete their execution within the execution time limit of 10 minutes. Upon investigation, we found that those documents were involved with processing large datasets (e.g., training machine learning models with large datasets) resulting in prolonged times to complete their execution. As a result, these files could not be executed within the specified time limit. Thus, we removed those files from our analysis, and the remaining files (142,717) were labeled into executable and non-executable groups.

3.1.3 Features Collection

In this subsection, we describe the features we collected to understand the differences between executable and non-executable groups of R Markdown files.

We hypothesize that R Markdown files located in active, popular, and collaborative projects are more likely to be executable since any issues in those projects are likely to be identified and fixed quickly. We have collected ten features at the project level as a proxy to capture the degree of activity, popularity, and collaboration of the project (e.g., number of stars and developers). Data collection of these features was facilitated through the use of the GitHub API [29].

At the file level, we have collected six features. For example, the total number of code cells indicates the computational tasks performed within an R Markdown file. Correspondingly, the total number of markdown cells reflects the degree of documentation present in the R Markdown file. During our manual study, we observed that many R Markdown files

Table 3.1: STUDIED FEATURES THAT ARE RELATED TO SEVEN DIFFERENT DIMENSIONS.

	Feature	**Explanation (data type)**	**Rationale**
Project	NumContrib	The number of developers (numeric)	Higher values of these features indicate that the project is well-maintained, popular and collaborative. R Markdown files located in those projects are more likely to be executable.
	NumStars	The number of stars (numeric)	
	NumForks	The number of forks (numeric)	
	NumWatchers	Total number of watchers (numeric)	
	NumDays	The duration of the project in days (numeric)	
	NumCommits	The number of commits (numeric)	
	NumClosedIssues	The number of closed issues (numeric)	
	NumOpenIssues	The number of open issues (numeric)	
	NumPullReqs	The number of pull requests (numeric)	
File	NumCommits	The number of commits (numeric)	Higher values of these features indicate a complex file which is more likely to be non-executable.
	NumContrib	The number of contributors (numeric)	
	NumCodeCells	The number of code cells (numeric)	
	NumMkdCells	The number of markdown cells (numeric)	
	NumPaths	The number of paths in the file (numeric)	
	NumCharFileName	The number of characters in the file name (numeric)	
Readability	FleschMkdCells	These metrics are calculated by the length of sentences and the number of syllables per word, that are used to quantify the readability of the text (numeric)	We are interested in learning whether the readability of an R Markdown file has any impact on its executability.
	FogMkdCells		
	LixMkdCells		
	KincaidMkdCells		
	AriMkdCells		
	SmogMkdCells		
Dependency	NumImpLib	The number of imported libraries (numeric)	An R markdown file with high dependency values is difficult to maintain. Thus, the file is more likely to be non-executable.
	OrdDepTree	The number of dependent libraries of those imported libraries (numeric)	
	PkgMang	Type of package manager (categorical)	
	NumImpRFile	The number of imported r files (numeric)	
	NumImpDataFile	The number of imported data files (numeric)	
	NumImpStmt	The number of import statements (numeric)	
	NumLang	The number of used languages (numeric)	
D. Exp.	DevContrib	The number of commit contributed by developer (numeric)	We are interested in learning whether developer experience has any relation with the executability of the file.
	DevContribRatio	The number of commits contributed by a developer/Total number of commits in that repository (numeric)	
Code Quality	NumBlkLin	The number of Blank lines (numeric)	These features represent the code quality of R Markdown files. We are interested in learning whether code quality features have any relation with the executability of R Markdown files.
	NumComLin	The number of Comment lines (numeric)	
	NumPhyLin	The number of Physical lines (numeric)	
	NumWarn	The number of warnings, errors and style suggestions obtained from the linter output for the file (numeric)	
	NumErrors		
	NumStyleSugges		
Content	MkdText	The words appear in Markdown cell (textual)	We are interested in learning whether certain words are more specific to executable or non-executable group of files.
	ProjDesc	The words appear in project description (textual)	
	LibList	The library names appear in the file (textual)	
	LinterOutput	The words appear in linter output (textual)	

are not executable because the imported files are missing or the absolute paths used in the documents are no longer valid. Thus, we consider the number of absolute paths and relative paths used in the document as the features. A prior study on Jupyter Notebook shows that a short file name may indicate quality issues present in that file. Thus, we consider the length of the file name as a feature [11].

We collect seven different features for the dependency dimension. A Markdown file can not be only dependent on external libraries but can also be dependent on external R source files, R Markdown files, and data files. First, we collect the total number of packages used in an R Markdown file by identifying the import statements used in that file. Second, we determine the order of dependency for each markdown file using the Library.io dataset [30]. This is done as follows. If an R Markdown file (F) imports three different packages (namely X, Y, and Z) using the import statement, we look into the Library.io dataset to identify the directly dependent packages of X, Y, and Z. Together, all these packages represent the order of dependency for F. R Markdown files can put code in R scripts and then run the code using source() or *sys.source()*. We consider the number of imported R scripts by identifying those statements. R Markdown often imports different data files (e.g., csv or text files) by calling different functions (e.g., *read.csv()*). Thus, we also consider the number of imported data files. Intuitively, the use of a package manager can help the project to better maintain external library dependencies. Thus, we consider the name of the package manager used in the project as a feature. An R Markdown file can contain code from other languages (e.g., Python, Julia) and depends on different language engines for the purpose of their execution. Thus, we also consider the number of languages used in a document as the feature.

The experience of a developer is determined using the developer contribution ratio following a prior work [31]. A developer X can have a lower contribution ratio than a developer Y in another project even if X contributed more than Y. Thus, we also consider the number of commits made by a developer as a proxy of his/her experience. An R Markdown file can be edited by one or more developers. The more developers are involved in developing a file, the more difficult it can be to maintain it. Thus, we take the median value when a document is edited by multiple developers.

For the code metric dimension, we have considered six features. We perform static analysis of R code using the *lintr* package [32] after merging all code chunks. We calculate the remaining code metrics using the *clock* package [33].

For the readability dimension, we have collected six readability features. These are *Kincaid* [34], *ARI* [35], *Flesch* [36], *Fog* [37], *Lix* [38], and *SMOG* [39]. We employ the Python package *readability* to calculate the above-mentioned features for the description of the project and the markdown text [40]. For the content dimension, we have considered four different features. For each R Markdown file, we merge the content of markdown cells, remove stop words, and calculate the frequency of words that appear in them using the scikit-learn library in Python [41]. We repeat the same process three more times for the remaining features where the words are coming from three different sources of information. These are a) library names used in the file b) the description of the project that contains the Markdown file c) output obtained from linting code cells written in R using the *lintr* package.

3.2 RQ1: How executable are R Markdown files? Are there differences between the executable and non-executable file groups?

Motivation: Understanding the prevalence of the non-executability of R Markdown files helps us to determine the severity of the problem. We are also interested in learning whether the executable and non-executable groups of markdown files differ from each other. Our investigation can provide a deeper understanding of the differences between the two groups of files, not only from the perspective of their content but also based on the contributing users.

Approach: We collect the output and error messages for those R Markdown files that either complete their execution or throw exceptions within the execution time limit of 10 minutes (see Section 3.1.2 for more details). We also categorize those files into executable and non-executable groups.

For each of the R Markdown file-related numeric features (see Section 3.1.3 for more details), we perform the Mann–Whitney U test [42] to determine whether or not the differences between executable and non-executable groups of Markdown files (along the collected numeric features) are statistically significant. We obtain a p-value by running the Mann-Whitney U test to compare two groups. We use a predetermined level of significance, which is 0.05, to compare the p-value. If the p-value is equal to or less than the significance level, we reject the null hypothesis and conclude that there is a significant difference between the two groups. We also use the Cohen's d test to measure the magnitude of the differences. We use the thresholds provided by Cohen [43] to determine the effect size, where $|d| < 0.2$ indicates that the effect size is negligible, $0.2 \leq |d| < 0.5$ indicates that the effect size is small, $0.5 \leq |d| < 0.8$ indicates that the effect size is medium, otherwise the effect size is large.

Result: A large number of R Markdown files failed to complete their execution. To be precise, 92,695 R Markdown files (64.95%) encountered errors out of the total processed files (142,717), while only 50,022 (35.05%) R Markdown files were executed successfully without any issues. Subsequently, we continued our investigation to determine

Table 3.2: P-value & Cohen 's $|d|$ for the studied numeric features.

		Mean/Median/Min/Max		
Dimension	**Feature**	**Executed**	**Non-executed**	**P-value & Cohen 's $\|d\|$**
Project	NumContrib	3.99/2.00/1.00/254.00	2.83/2.00/1.00/286.00	>=0.05 & 0.11 (N)
	NumStars	29.05/3.00/0.00/4453.00	22.23/3.00/0.00/5926.00	<0.05 & 0.04 (N)
	NumForks	9.05/2.00/0.00/2004.00	6.25/2.00/0.00/1956.00	<0.05 & 0.05 (N)
	NumWatchers	29.05/3.00/0.00/4453.00	22.23/3.00/0.00/5926.00	<0.05 & 0.04 (N)
	NumDays	822.85/1308.00/1.00/5005.00	869.05/1320.00/1.00/5479.00	<0.05 & 0.05 (N)
	NumCommits	353.32/155.00/1.00/21104.00	242.84/166.00/1.00/47517.00	<0.05 & 0.09 (N)
	NumClosedIssues	44.41/2.00/0.00/6744.00	28.82/3.00/0.00/6744.00	<0.05 & 0.07 (N)
	NumOpenIssues	5.60/1.00/0.00/816.00	3.45/1.00/0.00/630.00	>=0.05 & 0.09 (N)
	NumPullReqs	20.81/1.00/1.00/2018.00	12.34/1.00/1.00/2018.00	<0.05 & 0.09 (N)
File	NumCommits	3.76/3.00/1.00/335.00	5.15/5.00/1.00/527.00	<0.05 & 0.14 (N)
	NumContrib	1.22/1.00/1.00/45.00	1.26/1.00/1.00/23.00	<0.05 & 0.05 (N)
	NumCodeCells	4.99/6.00/0.00/287.00	13.76/16.00/0.00/622.00	<0.05 & 0.60 (M)
	NumMkdCells	5.44/7.00/0.00/2344.00	13.23/16.00/0.00/621.00	<0.05 & 0.47 (S)
	NumPaths	0.62/0.00/0.00/154.00	2.01/2.00/0.00/591.00	<0.05 & 0.26 (S)
	NumCharFileName	13.55/18.00/1.00/101.00	15.66/20.00/2.00/136.00	<0.05 & 0.24 (S)
Readability	FleschMkdCells	106.03/206.84/-2162.99/206.84	55.16/62.17/-2565.31/206.84	<0.05 & 0.71 (M)
	FogMkdCells	6.61/10.63/0.00/268.76	11.01/12.67/0.00/294.17	<0.05 & 0.57 (M)
	LixMkdCells	29.14/48.69/0.00/699.03	46.80/54.43/0.00/761.01	<0.05 & 0.72 (M)
	KincaidMkdCells	0.96/11.10/-15.70/315.20	10.03/13.20/-15.70/376.60	<0.05 & 0.72 (M)
	AriMkdCells	12.08/16.20/-16.30/1147.10	16.94/18.70/-16.30/1076.50	<0.05 & 0.26 (S)
	SmogMkdCells	3.87/13.75/-39.61/760.80	12.37/15.43/-39.61/1260.35	<0.05 & 0.46 (S)
Dependency	NumImpLib	1.21/1.00/0.00/528.00	4.01/5.00/0.00/372.00	<0.05 & 0.59 (M)
	OdrDepdTree	19.35/22.00/0.00/5350.00	60.85/86.00/0.00/5712.00	<0.05 & 0.50 (S)
	NumImpRFile	0.11/0.00/0.00/47.00	0.37/0.00/0.00/332.00	<0.05 & 0.12 (N)
	NumImpDataFile	0.49/0.00/0.00/136.00	1.57/1.00/0.00/588.00	<0.05 & 0.23 (S)
	NumImpStmt	1.21/1.00/0.00/528.00	4.01/5.00/0.00/372.00	<0.05 & 0.59 (M)
	NumLang	0.72/1.00/0.00/3.00	1.01/1.00/0.00/4.00	<0.05 & 0.99 (L)
Developer Experience	DevContrib	2.74/3.00/1.00/220.00	3.62/4.00/1.00/239.00	<0.05 & 0.16 (N)
	DevContribRatio	0.92/1.00/0.00/1.00	0.90/1.00/0.01/1.00	<0.05 & 0.07 (N)
Code Quality	NumBlkLin	33.35/39.00/0.00/16364.00	63.63/74.00/0.00/9031.00	<0.05 & 0.30 (S)
	NumComLin	74.92/81.00/0.00/17569.00	107.76/122.00/0.00/10744.00	<0.05 & 0.20 (S)
	NumPhyLin	28.67/28.00/0.00/4650.00	108.93/116.00/0.00/13756.00	<0.05 & 0.44 (S)
	NumWarn	0.86/0.00/0.00/695.00	3.88/0.00/0.00/1687.00	<0.05 & 0.14 (N)
	NumErrors	0.03/0.00/0.00/4.00	0.03/0.00/0.00/10.00	<0.05 & 0.05 (N)
	NumStyleSugges	33.63/19.00/0.00/9892.00	138.91/123.00/0.00/34802.00	<0.05 & 0.29 (S)

whether these R Markdown files shared similarities or exhibited notable distinctions. This investigation covered six key dimensions: project, file, readability, dependency, developer experience, and code quality.

All the features except the number of open issues and the number of contributors in a project, are statistically significantly different between executable and non-executable groups of files. For example, for the readability dimension, the non-executable group of files obtains a lower *Flesch* readability score compared to that of the executable group of files. The *Flesch* readability scores are statistically significantly different between the two groups of files with a medium effect size. Table 3.2 shows the comparison of studied numeric features between two groups of files, including the mean, median, minimum and maximum values of each feature.

Non-executable files have significantly more dependencies compared to executable files. For example, non-executable R markdown files have an average of 0.37 dependent R files, 1.57 dependent data paths, 4.01 imported libraries, and 60.85 order of dependency, respectively. In contrast, for executable files, these values were lower at 0.11, 0.49, 1.21, and 19.35, respectively. This is reasonable, as a file with more dependencies is more likely to have broken dependencies. While both executable and non-executable R Markdown files had a minimum number of zero, the maximum number consistently surpassed that of executable R Markdown files for those features, except for the number of libraries used in that file. For example, considering executable files, the maximum number of libraries used in a single file was 528 whereas it was 372 for non-executable files. The maximum value for the order of dependency for R Markdown files is 5,350 for an executable file, in contrast, the value for the non-executable file is higher, specifically 5,712.

Non-executable R Markdown files exhibited consistently larger feature values in the realm of code quality compared to executable R Markdown files. The averages for non-executable R Markdown files, considering the number of lines, blank lines, comment lines, and physical or code lines were 280.33, 63.63, 107.76, and 108.93, respectively. In contrast, executable files showcased lower mean values of 136.96, 33.35, 74.92, and 28.67 for the same categories. For executable R Markdown files, the average number of messages in linter output is lower, roughly 34.51, than the non-executable group of files for which the value was 142.82. Though both non-executable and executable R Markdown files have a minimum value of zero, the maximum values for executable files surpassed all features except the number of physical lines representing code. The disparities in code quality-related features between non-executable and executable files were statistically significant ($p - value < 0.05$) with a small effect size (such as for comment and blank lines the values were 0.20 and 0.30, respectively).

In case of the file dimension, non-executable files have higher feature values (e.g., higher commits, contributors, paths used in the file) compared to that of executable files. Non-executable R Markdown files, on the whole, showcased greater values in terms of contributions compared to executable R Markdown files. For example, non-executable files have an average commit of 5.15 whereas the value is 3.76 for the executable files. Here, the effect size of commits, and contributors are negligible whereas the effect size is medium for the number of code cells between executable and non-executable groups of files. The mean values for non-executable files, regarding the total contributors and commits, were 1.26 and 5.15, respectively. In contrast, for executable files, these values were slightly lower at 1.22 and 3.76. Despite both non-executable and executable files have a minimum of one contributor, the maximum number of contributors and commits consistently surpassed that of executable R Markdown files. The disparities in contribution-related features between executable and non-executable files were found statistically significant ($p - value < 0.05$), accompanied by negligible effect size, 0.05 and 0.14 for the number of contributors and commits, respectively.

3.3 RQ2: What are the reasons that hinder the executability of R Markdown files?

Motivation: Understanding why R Markdown files are not executed can benefit us in two ways. First, R Markdown users can understand the problem better and make necessary changes to execute those documents. Second, tool developers can develop automated techniques to support the execution of such documents.

Approach: We conducted a qualitative study to determine the underlying reasons that prevent the execution of Markdown files. It is not feasible to manually analyze all non-executable Markdown files for this qualitative study. Thus, we randomly sampled 387 Markdown files to obtain a 95% confidence level with a margin of error of 5%. We perform the random selection of files using the pandas.DataFrame.sample function [44] to conduct the manual study. We performed an open-coding-like process on the sampled non-executable Markdown files to identify the reasons that prevent the execution of those files. The process is summarized as follows.

- Two participants checked the error messages obtained while executing the Markdown files and manually investigated the files without any specific reasons in mind. The error message we obtained may not represent the underlying reasons that affect the executability of the markdown files. For example, while executing a markdown file we received the following error message: "Error in library (AlphaSimR): there is no package called "AlphaSimR". However, it is not clear what prevents the library from being installed without further investigation. For the purpose of this study, participants review prior research papers, online documentation, and GitHub issues to identify the underlying reasons.

- A set of reasons was derived from the manual study. The sampled Markdown files are then categorized based on these derived reasons. If there was a disagreement during the manual study, the participants discussed it with each other to make the final decision.

- We computed Cohen's kappa [45] to measure to inter-rater agreement before the disagreement was resolved. The higher the kappa value, the better the agreement

Table 3.3: List of reasons that hinder the executability of R Markdown files.

Reasons	**Number of files**	**Percentage (%)**
Dependency not installed	171	44.18
Library not installed	91	23.51
File missing	56	14.47
Semantic error	40	10.34
Local directory missing	18	4.65
Deleted library	5	1.29
Outdated library	3	0.78
Syntax error	3	0.78

between the observers. We obtained a kappa value of 0.81 which indicates a high degree of agreement.

The result from our manual study is summarized as follows.

Results: Our qualitative analysis identified the following reasons that hinder the execution of R Markdown files (see Table 3.3 for the distribution of manually analyzed R Markdown files).

3.3.1 Missing libraries

The most common problem that hinders the executability of R Markdown files is missing libraries. Around 270 out of 387 files that we manually analyzed were unable to execute due to missing libraries for various reasons. We describe those reasons in the following sub-sections.

3.3.1.1 Dependency not installed

This is the most common issue we observed. 171 out of 387 files encountered this issue. Each library depends on other libraries to complete its tasks, and if these dependencies are not correctly installed, errors can occur. To illustrate this, let's take the example of a library named *leaflet*, which is used to create interactive web maps within an R Markdown file. When *renv* was used to manage the dependency, it identified the library correctly. However, *leaflet* has dependency on several libraries, such as *raster* and *sf*. The *raster* library depends on another library, called *terra*, which has specific system requirements (e.g., C++17, GEOS (= 3.4.0), GDAL (= 2.2.3), PROJ (= 4.9.3), sqlite3). When we tried to install *terra*, it required those specific system requirements to be installed. If those requirements are not found on the system that will fail the installation of the *terra* library. Failure to install *terra* impacts the installation of *raster* which ultimately causes the installation of *leaflet* library to fail.

Another example is *rJava*, which is a low-level R to Java interface. We found that *rJava* also has system requirements (Java JDK 1.2 or higher (for JRI/REngine JDK 1.4 or higher) and GNU make). If JDK is not available on the system, the installation of *rJava* will fail.

3.3.1.2 Library not installed

23.51% (91 out of 387) of the R Markdown files were not executed due to a library not being correctly installed (triggering errors, such as "Error in library"). Upon closer examination, we found that *renv* primarily depends on the *CRAN* for installing libraries. The package management tool *renv* offers some parameters to be configured, such as *repos* and *bioconductor* for downloading libraries from alternative sources (such as GitHub repositories and Bioconductor). However, it struggles to install libraries that are hosted in places other than *CRAN* and requires manual installation of libraries for such cases.

For instance, let's consider a library named *ggtree* that is used for visualizing tree and annotation data. When we used *renv* to automatically install imported libraries in an R Markdown file, *renv* correctly identified the library but failed to install it because the library is hosted in *Bioconductor*.

3.3.1.3 Outdated library

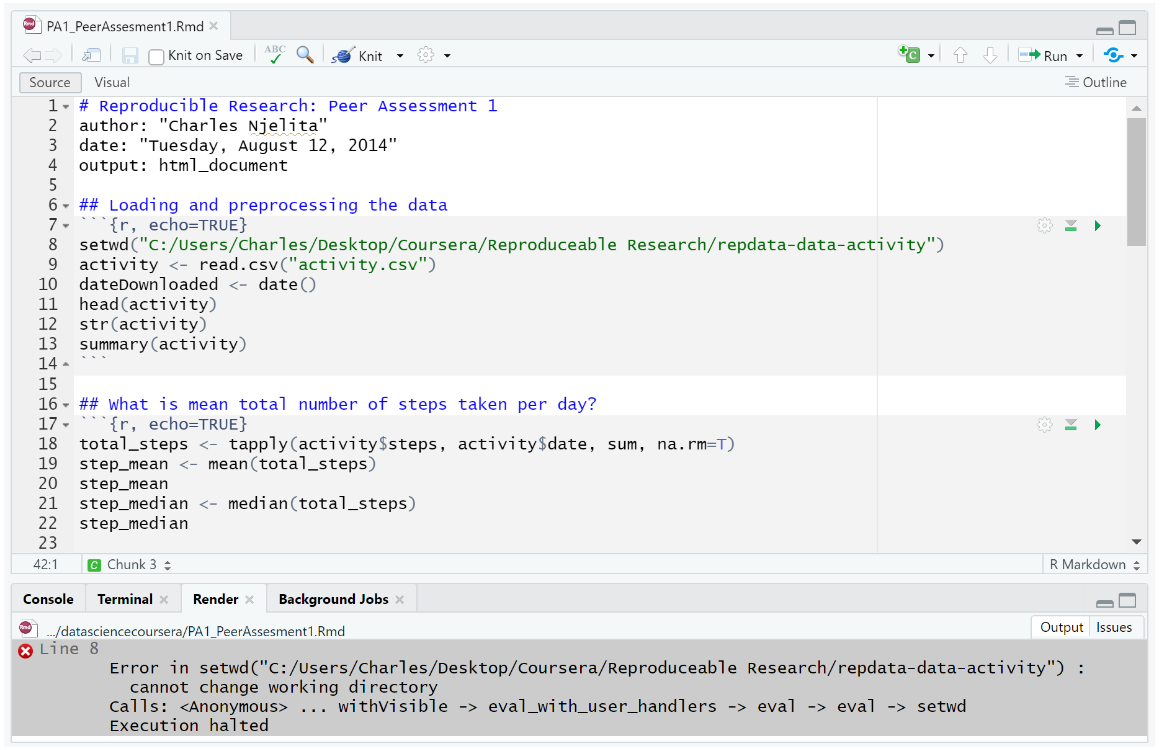
We also found 3 (out of 387) cases in which outdated libraries hinder the execution of R Markdown files. For example, in an R Markdown file, *renv* tried to install and load a library named *EBImage* that is located in *Bioconductor*. The R Markdown file used the *biocLite* function to install the package pragmatically from *Bioconductor*. However, the function is now outdated and *BiocManager* is now used to install any libraries from *Bioconductor* for R version 3.5 or greater. Here, the developer used the outdated function to install the library. Thus, we failed to install the required library that prevented the execution of the R Markdown file.

3.3.1.4 Deleted Library

We also found some libraries that were used in R Markdown files but are no longer available now as those libraries were removed from *CRAN*. In total, 5 out of 387 files had this issue. Thus, we were unable to execute those files. For example, *HydeNet* was used in an R Markdown file but is now removed from *CRAN*. When *renv* tried to install the dependency, it produced an error because the target library is no longer available.

3.3.2 Missing data

Developers occasionally forget to modify local paths in their code before committing it or forget to upload required file(s), which can prevent R Markdown files from being executed because those paths or files are no longer available. The following section describe each of those cases.



~~~
# Reproducible Research: Peer Assessment 1
author: "Charles Njelita"
date: "Tuesday, August 12, 2014"
output: html_document

## Loading and preprocessing the data
```{r, echo=TRUE}
setwd("C:/Users/Charles/Desktop/Coursera/Reproduceable Research/repdata-data-activity")
activity <- read.csv("activity.csv")
dateDownloaded <- date()
head(activity)
str(activity)
summary(activity)
```

## What is mean total number of steps taken per day?
```{r, echo=TRUE}
total_steps <- tapply(activity$steps, activity$date, sum, na.rm=T)
step_mean <- mean(total_steps)
step_mean
step_median <- median(total_steps)
step_median

~~~

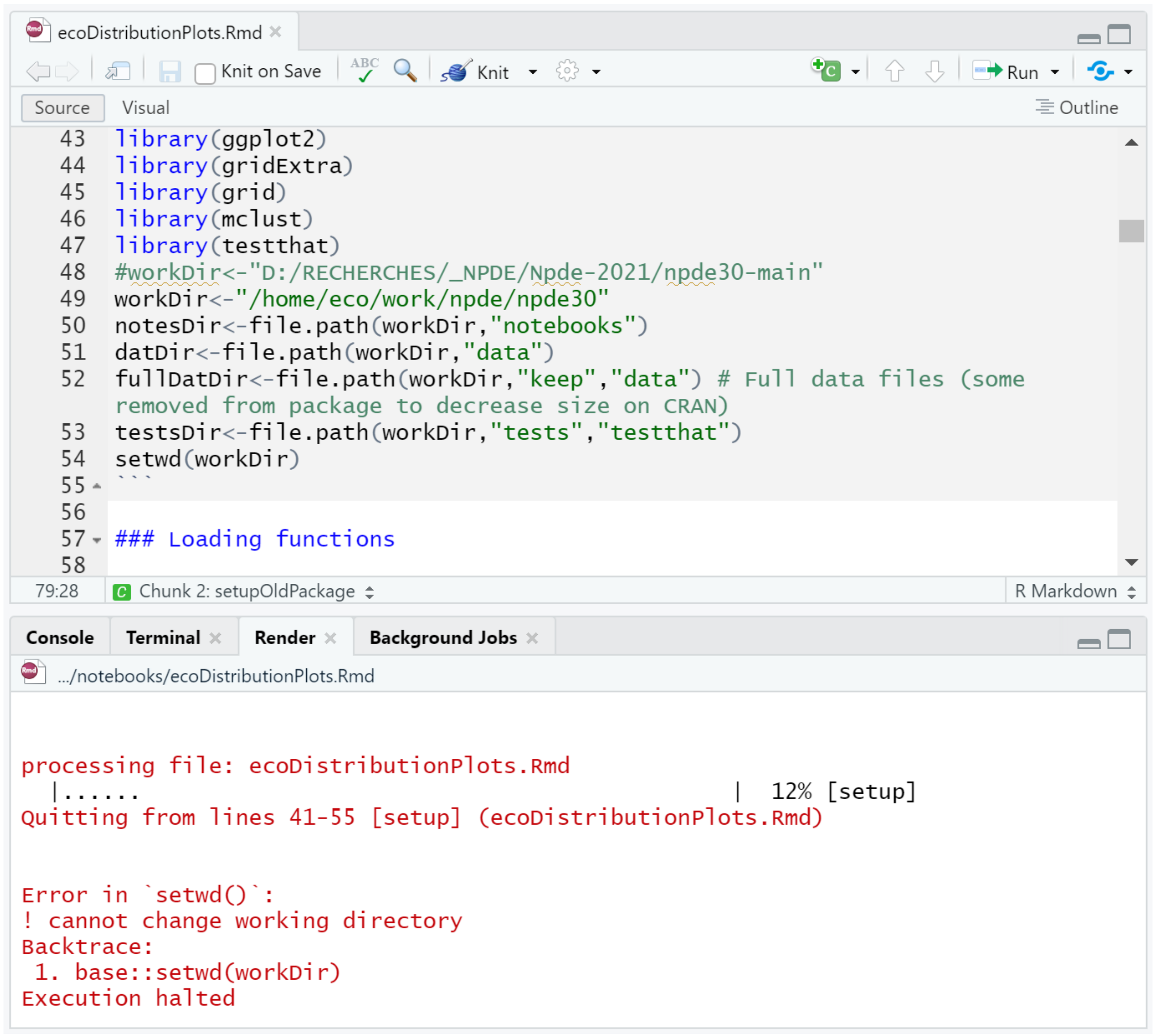
```
.../datasciencecoursera/PA1_PeerAssesment1.Rmd
Line 8
        Error in setwd("C:/Users/Charles/Desktop/Coursera/Reproduceable Research/repdata-data-activity") :
          cannot change working directory
        Calls: <Anonymous> ... withVisible -> eval_with_user_handlers -> eval -> eval -> setwd
        Execution halted
```

Figure 3.2: An example of missing a local directory.

3.3.2.1 Local directory missing

18 out of 387 files were not executed because of missing directories. This error occurs when a developer uses a local directory of their system in an R Markdown file, but forgets to remove it from the R Markdown file before committing it . When someone downloads the R Markdown file and tries to execute it, the file fails to execute because the path to the directory is no longer available. For example, a developer used "setwd("C:/Users/Charles/Desktop/Coursera/Reproduceable Research/repdata-data-activity")" to change the working directory (see figure 3.2). When we cloned the project and tried to execute the file we received the following error: "Error in setwd". This is because the location to the directory is invalid in our system. For better understanding, we can consider another example where a user assigned the following path to the *workDir* variable: "/home/noble_mannu/Documents/PhD/First/STAT_2131_Applied_Statistical_Methods_I". After assigning the variable the user changed his working directory using *setwd(workDir)* command. When we tried to execute the file we received an error message. This is because the path does not exist in our system. For better understanding, we can look for another example in figure 3.3 where the developer tried to set

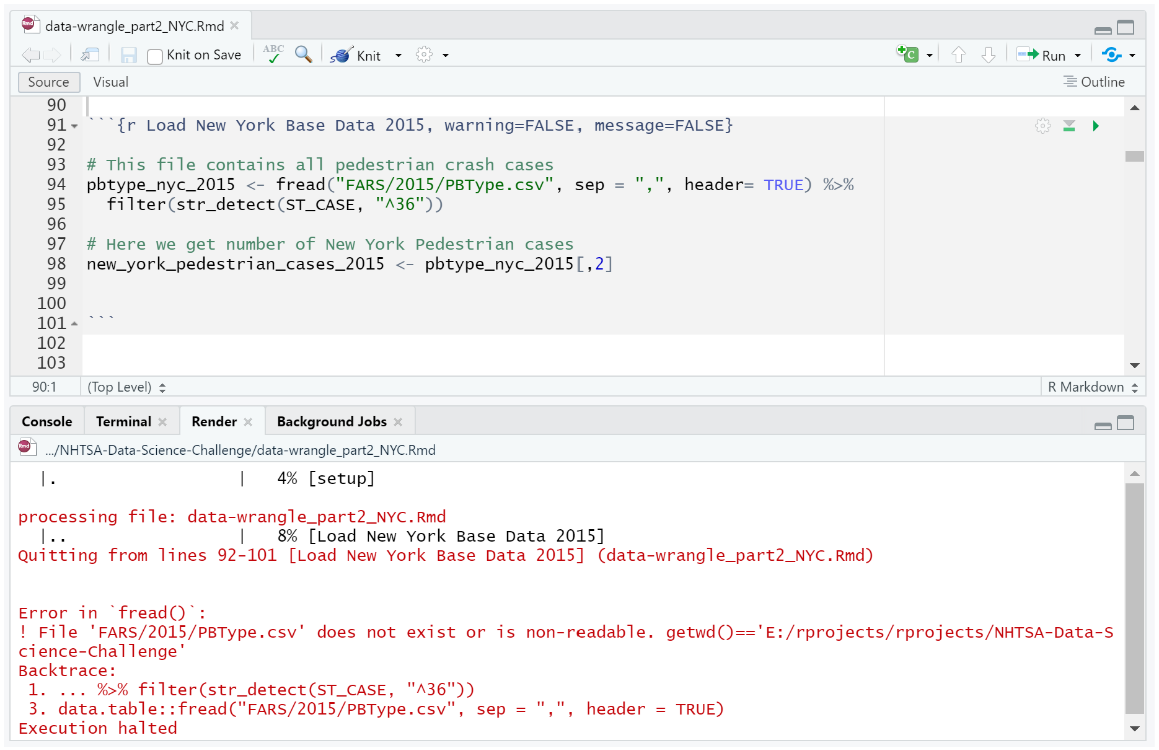
the working directory to "/home/eco/work/npde/npde30". When we tried to run that file, *Error in 'setwd()':* hinder the execution. This occurred because the path only exists in the creator's system, not ours.



```
library(ggplot2)
library(gridExtra)
library(grid)
library(mclust)
library(testthat)
#workDir<-"D:/RECHERCHES/_NPDE/Npde-2021/npde30-main"
workDir<-"/home/eco/work/npde/npde30"
notesDir<-file.path(workDir,"notebooks")
datDir<-file.path(workDir,"data")
fullDatDir<-file.path(workDir,"keep","data") # Full data files (some
removed from package to decrease size on CRAN)
testsDir<-file.path(workDir,"tests","testthat")
setwd(workDir)
```

Loading functions
```

```
processing file: ecoDistributionPlots.Rmd
  |......                                          |  12% [setup]
Quitting from lines 41-55 [setup] (ecoDistributionPlots.Rmd)

Error in `setwd()`:
! cannot change working directory
Backtrace:
 1. base::setwd(workDir)
Execution halted
```

Figure 3.3: Another example of missing a local directory.

3.3.2.2 File missing

We also found a noticeable amount (i.e., 56 out of 387 files or 14.5%) of R Markdown files that were not executed because of missing files. It is found that developers accessed files in R Markdown files to complete certain tasks. When they committed those Markdown files in GitHub, they forgot to upload those required files. For example, when we downloaded an R Markdown file and tried to execute the file, we found an error that stopped the execution of that file. When we further analyzed the error, we found that the Markdown file used
```


data-wrangle_part2_NYC.Rmd
Knit on Save
Knit
Run
Source
Visual
Outline
Console
Terminal
Render
Background Jobs
.../NHTSA-Data-Science-Challenge/data-wrangle_part2_NYC.Rmd

~~~
```{r Load New York Base Data 2015, warning=FALSE, message=FALSE}

# This file contains all pedestrian crash cases
pbtype_nyc_2015 <- fread("FARS/2015/PBType.csv", sep = ",", header= TRUE) %>%
  filter(str_detect(ST_CASE, "^36"))

# Here we get number of New York Pedestrian cases
new_york_pedestrian_cases_2015 <- pbtype_nyc_2015[,2]


```


~~~

```
90:1    (Top Level)    R Markdown
```

```
  |.                   |   4% [setup]

processing file: data-wrangle_part2_NYC.Rmd
  |..                  |   8% [Load New York Base Data 2015]
Quitting from lines 92-101 [Load New York Base Data 2015] (data-wrangle_part2_NYC.Rmd)


Error in `fread()`:
! File 'FARS/2015/PBType.csv' does not exist or is non-readable. getwd()=='E:/rprojects/rprojects/NHTSA-Data-S
cience-Challenge'
Backtrace:
 1. ... %>% filter(str_detect(ST_CASE, "^36"))
 3. data.table::fread("FARS/2015/PBType.csv", sep = ",", header = TRUE)
Execution halted
```

Figure 3.4: An example of missing a file.

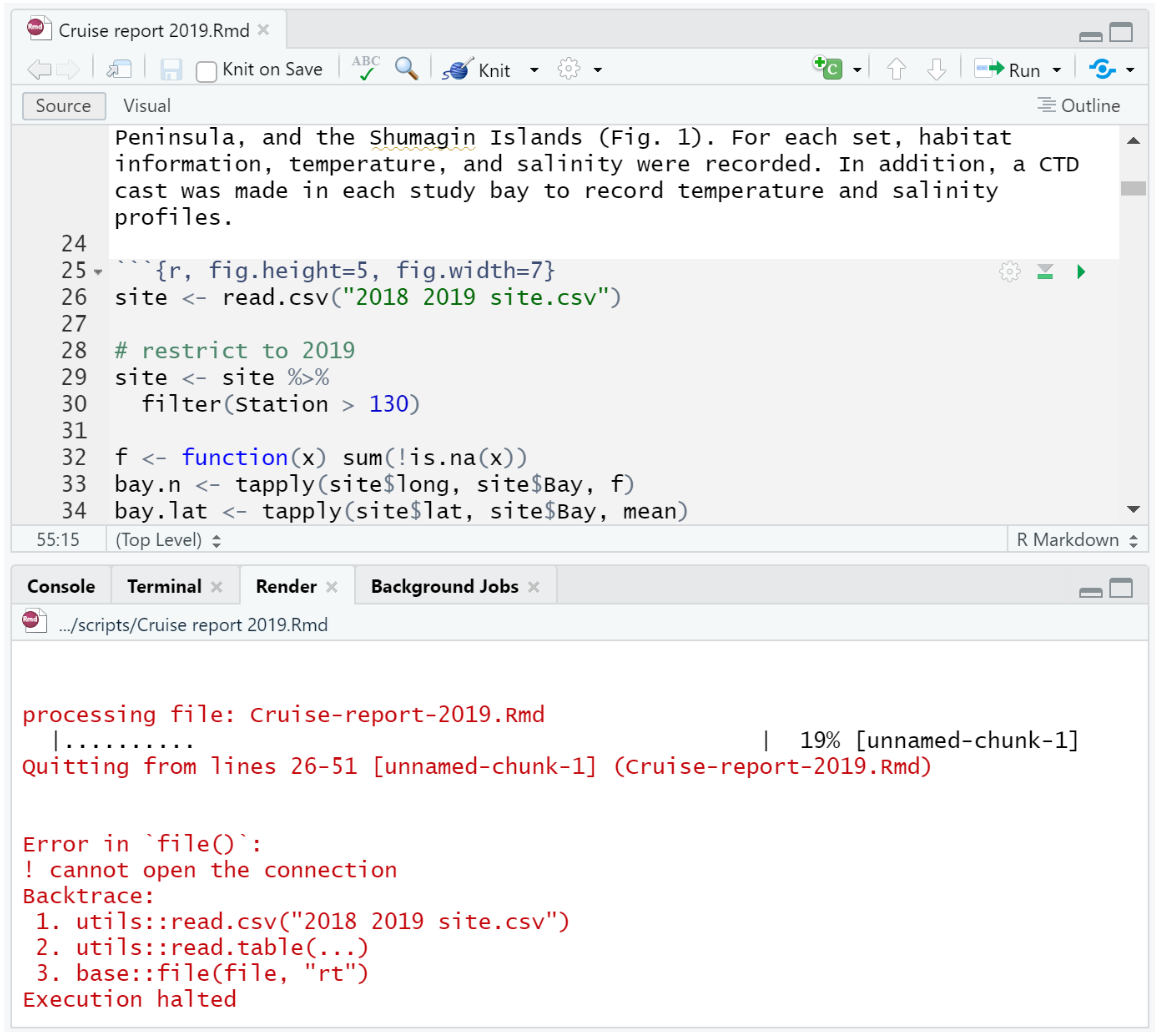
"fread("FARS/2015/PBType.csv", sep = ",", header= TRUE)" to read a CSV file but CSV file was not available in the project (see Figure 3.4). The missing file prevented the execution of the R Markdown file. Another example is in figure 3.5), where the developer created an R Markdown file to create a report. At the beginning of the file, he read a CSV file "2018 2019 site.csv" to create that report. When we downloaded that file and tried to execute it, it stopped its execution because of the unavailability of the CSV file.

3.3.3 Erroneous Code

R Markdown files failed to execute because of the presence of erroneous code in those files. We observed two different types of errors (i.e., syntax error and semantic error) caused the issue.

3.3.3.1 Syntax error

3 files failed to execute due syntax errors. For example, we encountered an R Markdown file that was not seamlessly merged. The file failed to execute because of the presence of surplus syntax in it (see Figure 3.6).



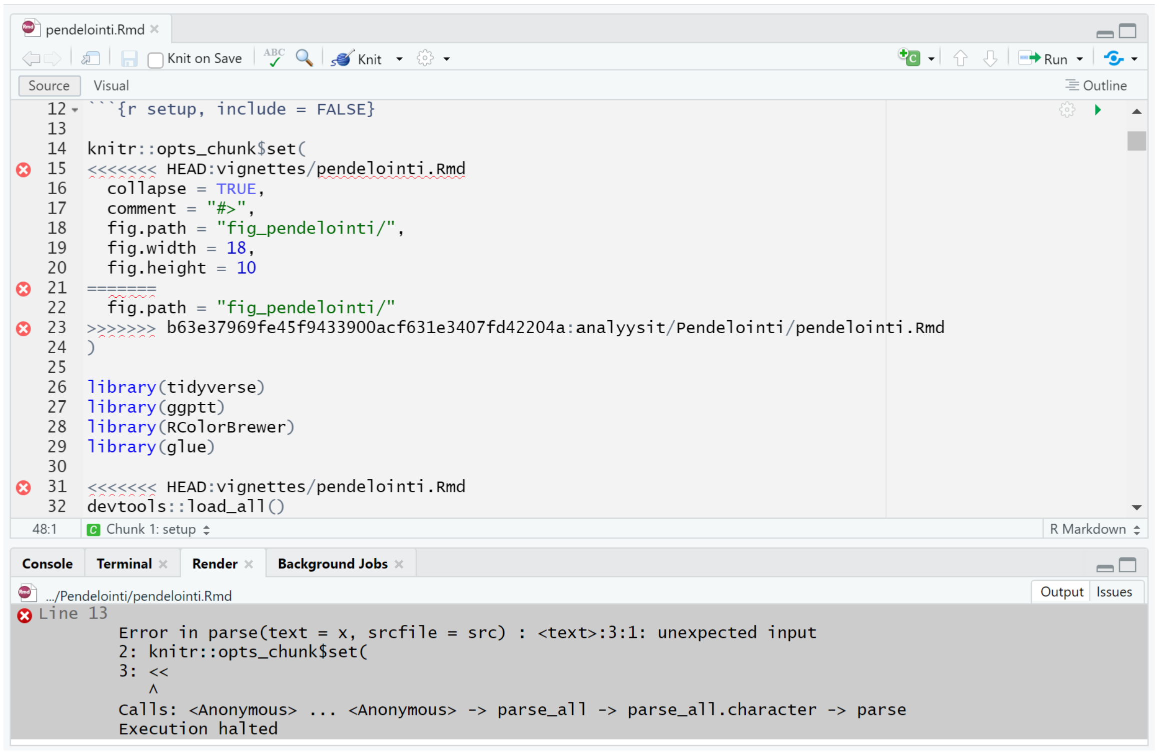
~~~
Peninsula, and the Shumagin Islands (Fig. 1). For each set, habitat
information, temperature, and salinity were recorded. In addition, a CTD
cast was made in each study bay to record temperature and salinity
profiles.

```{r, fig.height=5, fig.width=7}
site <- read.csv("2018 2019 site.csv")

# restrict to 2019
site <- site %>%
  filter(Station > 130)

f <- function(x) sum(!is.na(x))
bay.n <- tapply(site$long, site$Bay, f)
bay.lat <- tapply(site$lat, site$Bay, mean)
~~~

```
processing file: Cruise-report-2019.Rmd
  |..........                                               |  19% [unnamed-chunk-1]
Quitting from lines 26-51 [unnamed-chunk-1] (Cruise-report-2019.Rmd)


Error in `file()`:
! cannot open the connection
Backtrace:
 1. utils::read.csv("2018 2019 site.csv")
 2. utils::read.table(...)
 3. base::file(file, "rt")
Execution halted
```

Figure 3.5: Another example of missing a file.

3.3.3.2 Semantic error

Semantic errors, also known as logical errors, occur when R Markdown files are syntactically correct but there are some logical errors in those files. Around 10.5% (40 out of 387) of files were not executed due to such error. For example, we found a file where a variable *db* is assigned a value based on certain conditions. If the condition does not satisfy, *db* will be set to null because there is no default value for this variable. When we tried to execute the file, it produced errors because of the null value. If the developer could manage the null value carefully, it would not produce such errors. For better understanding, we can consider another example where a *dataframe* is created with fixed column names: Maths, Geography, English, Physics, Chemistry, and Sex. When a developer added a new row containing values for all the columns except Chemistry, it produced an error as the number



~~~
```{r setup, include = FALSE}

knitr::opts_chunk$set(
<<<<<<< HEAD:vignettes/pendelointi.Rmd
  collapse = TRUE,
  comment = "#>",
  fig.path = "fig_pendelointi/",
  fig.width = 18,
  fig.height = 10
=======
  fig.path = "fig_pendelointi/"
>>>>>>> b63e37969fe45f9433900acf631e3407fd42204a:analyysit/Pendelointi/pendelointi.Rmd
)

library(tidyverse)
library(ggptt)
library(RColorBrewer)
library(glue)

<<<<<<< HEAD:vignettes/pendelointi.Rmd
devtools::load_all()
~~~

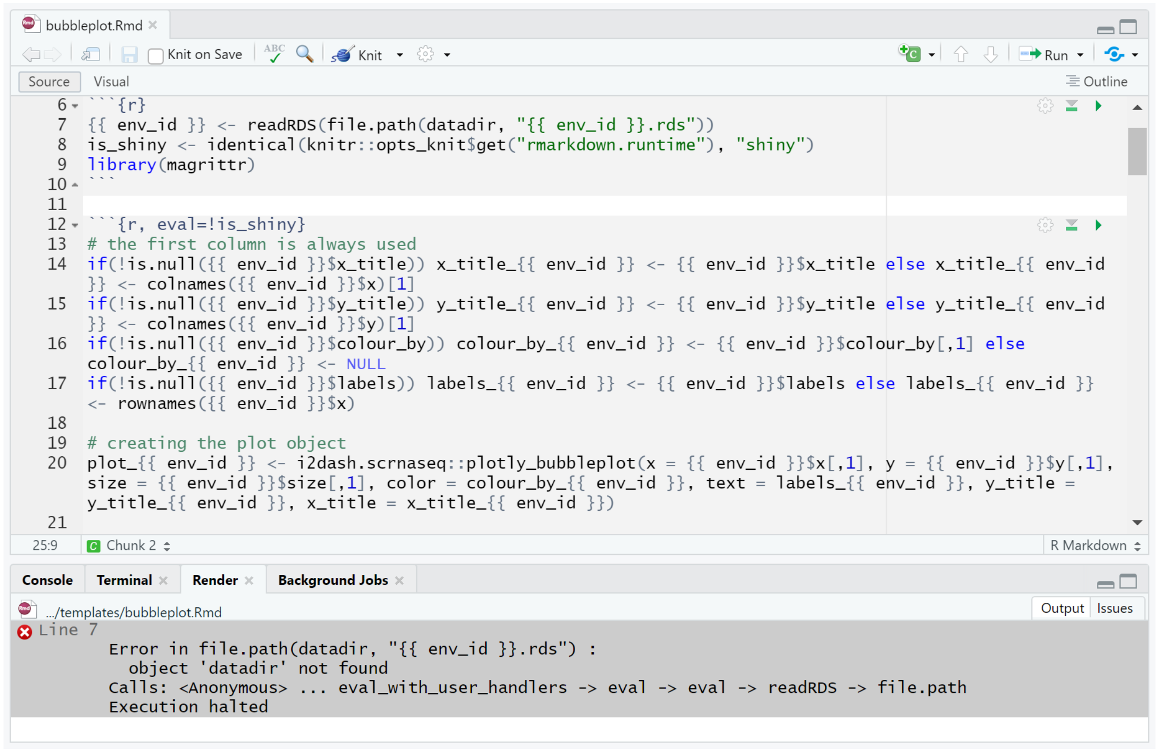
```
Line 13
        Error in parse(text = x, srcfile = src) : <text>:3:1: unexpected input
        2: knitr::opts_chunk$set(
        3: <<
           ^
        Calls: <Anonymous> ... <Anonymous> -> parse_all -> parse_all.character -> parse
        Execution halted
```

Figure 3.6: An example of a syntax error.

of column arguments did not match. Another example, within the context of the figure 3.7, the file path is assigned using the variable *datadir*. However, upon close inspection, we found that there was no initialization or assignment above it. Consequently, we failed to execute the file because the variable was not defined previously.

3.4 RQ3: Can we predict whether an R Markdown file is not executable?

Motivation: We have found that 64.95% of R Markdown files are not executable, even though they are meant for sharing and supporting reproducible results. A non-executable R Markdown file can cause even more confusion for novice users when they try to reproduce the study result. This is because they may not be familiar with the underlying reasons that hinder the execution of the file. In our study, 25.83% of all R Markdown files failed to complete their execution within 10 minutes, the actual time to complete their execution can be even more. Such a long waiting time makes it difficult to test the executability of R Markdown files through execution, particularly when the goal is to search for reusable documents to complete a development task. Thus, we are interested in learning whether we can build a classifier to predict the executability of R Markdown files. The classifier can



~~~
```{r}
{{ env_id }} <- readRDS(file.path(datadir, "{{ env_id }}.rds"))
is_shiny <- identical(knitr::opts_knit$get("rmarkdown.runtime"), "shiny")
library(magrittr)
~~~

```{r, eval=!is_shiny}
# the first column is always used
if(!is.null({{ env_id }}$x_title)) x_title_{{ env_id }} <- {{ env_id }}$x_title else x_title_{{ env_id }} <- colnames({{ env_id }}$x)[1]
if(!is.null({{ env_id }}$y_title)) y_title_{{ env_id }} <- {{ env_id }}$y_title else y_title_{{ env_id }} <- colnames({{ env_id }}$y)[1]
if(!is.null({{ env_id }}$colour_by)) colour_by_{{ env_id }} <- {{ env_id }}$colour_by[,1] else colour_by_{{ env_id }} <- NULL
if(!is.null({{ env_id }}$labels)) labels_{{ env_id }} <- {{ env_id }}$labels else labels_{{ env_id }} <- rownames({{ env_id }}$x)

# creating the plot object
plot_{{ env_id }} <- i2dash.scrnaseq::plotly_bubbleplot(x = {{ env_id }}$x[,1], y = {{ env_id }}$y[,1], size = {{ env_id }}$size[,1], color = colour_by_{{ env_id }}, text = labels_{{ env_id }}, y_title = y_title_{{ env_id }}, x_title = x_title_{{ env_id }})
```

```
Line 7
        Error in file.path(datadir, "{{ env_id }}.rds") :
          object 'datadir' not found
        Calls: <Anonymous> ... eval_with_user_handlers -> eval -> eval -> readRDS -> file.path
        Execution halted
```

Figure 3.7: An example of a semantic error.

alert users about executability issues when they select an online Markdown file to use. Such a classifier can also be used by search engines to automatically filter files that are highly likely to be non-executable.

Approach: We consider the problem as a binary classification task. It is important to identify viable factors for the success of the classification. We use all the factors across seven different dimensions that were discussed in Table 3.1. In the following subsections, we describe the methodology of building classifier(s), evaluation metrics, and analysis of the result.

Classifier Construction: We consider five different classifiers in this study. These are Gaussian Naive Bayes, Logistic Regression, Decision Tree, Random Forest, and Extreme Gradient Boosting (XGBoost). Prior studies in the software engineering area also used these classifiers. Gaussian Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem that assumes the features follow a Gaussian distribution [46]. The reason why the algorithm is called 'naive' is because it assumes that all features are independent of each other. This assumption is often regarded as too simplistic, but it can still produce satisfactory results in real-world scenarios. Logistic Regression uses a logistic function to convert feature values to binary outcomes [47]. Decision Trees are supervised learning algorithms that predict the target variable by learning decision rules from input features [48].

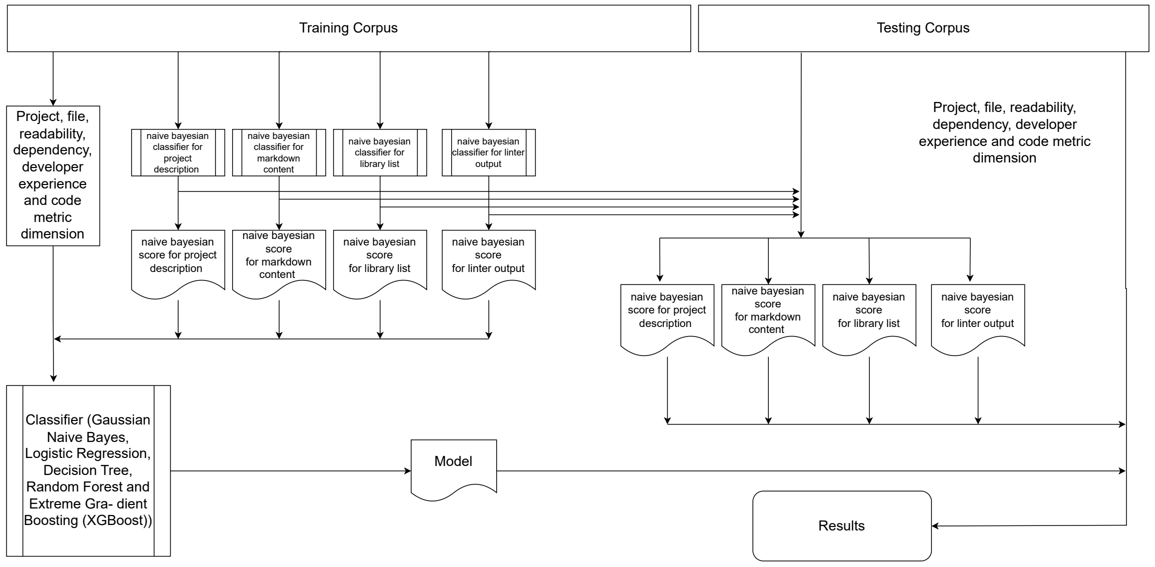

Training Corpus
Testing Corpus
Project, file, readability, dependency, developer experience and code metric dimension
naive bayesian classifier for project description
naive bayesian classifier for markdown content
naive bayesian classifier for library list
naive bayesian classifier for linter output
Project, file, readability, dependency, developer experience and code metric dimension
naive bayesian score for project description
naive bayesian score for markdown content
naive bayesian score for library list
naive bayesian score for linter output
Classifier (Gaussian Naive Bayes, Logistic Regression, Decision Tree, Random Forest and Extreme Gra- dient Boosting (XGBoost))
Model
Results

Figure 3.8: Classifier creation for R Markdown files.

Random Forest is a robust ensemble learning method that combines the knowledge of several decision trees [49]. XGBoost is a high-performance gradient-boosting algorithm that shows high predictive power and exceptional speed [50].

Since the number of non-executable R Markdown files is significantly higher than the number of executable files (i.e., the data is imbalanced. Around 65% of them are non-executable and 35% of them executable.), we apply an oversampling algorithm, called SMOTE [51]), to the training dataset to handle the data imbalance issue. When constructing the classifiers, we use the composite model architecture, similar to the prior work of Ibrahim et al. [31], which consists of two steps (Figure 3.8). First, we construct a classifier using the content to determine the likelihood of a file to be executable. For each of the R Markdown files, the classifier provides the probability that the document belongs to the executable category based on the content of the document. Please note that the content can come from four different sources of information (these are project description, markdown content, library names, and linter output) and we construct the first-level classifier for each source of information. Second, we combine the probability scores obtained from the four first-level classifiers with the remaining features to determine whether the file will be executable or not.

Performance evaluation: We evaluate the performance of classifiers using AUC (Area under the ROC Curve) as the primary evaluation metric. The value of AUC ranges between 0 and 1. An AUC of 0.5 indicates random guessing. While an AUC of 0 indicates com-

Table 3.4: The performance of all five classifiers considered in our study. Here, the terms DT, GNB, KNN, LR, and XGBoost refer to Decision Tree, Gaussian Naive Bayes, K-Nearest Neighbors, Logistic Regression, and Extreme Gradient Boosting, respectively.

Classifier	**AUC**		**Accuracy**	**Precision**	**Recall**	**F1 score**
DT	83.02	E	85.52	78.64	77.10	77.86
		NE	85.42	87.41	88.36	87.88
GNB	74.36	E	77.93	75.07	60.61	67.05
		NE	76.23	73.66	84.59	78.73
KNN	79.81	E	83.83	70.03	78.40	73.98
		NE	84.38	89.59	84.72	87.09
LR	74.40	E	80.41	58.42	76.61	66.29
		NE	81.83	90.38	80.13	84.95
XGBoost	82.68	E	86.33	72.66	84.29	78.05
		NE	86.93	92.70	86.28	89.37

plete failure, an AUC of 1 indicates a perfect classifier. In addition, we also determine the precision, recall, and F-measure. For all experiments, we use the 10-fold cross-validation to measure the performance of a classifier. In the case of 10-fold cross-validation, the dataset is randomly divided into 10 equal-size folds. We use one of the folds for testing and the remaining nine folds for testing. The process is repeated 10 times and we record the average value for each metric.

Results: Table 3.4 shows the performance of all five classifiers considered in our study. The DT classifier shows a median AUC of 83.02%, the highest AUC value among all five classifiers considered for this study. The result is closely followed by XGBoost, with a median AUC of 82.68%. The remaining three classifiers (i.e., KNN, LR and GNB) also perform well, obtaining a median AUC of 79.81%, 74.40% and 74.36%. If our goal is to identify R Markdown files that are most likely to be non-executable (i.e., we do not want to miss any non-executable file), XGBoost performs the best, obtaining 86.93% accuracy for the non-executable file category. This is followed by LR (81.83%), KNN (84.38%), DT (85.42%) and GNB (76.23%). Overall, all the classifiers perform well in our study. Thus, the selected factors are effective in distinguishing non-executable R Markdown files from those that are executable.

3.5 Discussion

3.5.1 Implications of our findings

Future research is encouraged to develop tools to resolve non-executable R Markdown files (e.g., resolving dependencies/libraries issues). In RQ3, we developed classifiers to predict the executability of an R Markdown file and achieve good AUC, which showcases the potential of building classifiers to detect non-executable R Markdown files. However, how to resolve non-executable files requires future research. In RQ2, we observe the majority of the issues that lead to the failures of execution for R Markdown files are related to missing and broken dependencies of libraries to be installed. For instance, One research direction is to first analyze the specific system requirements for installing a library (e.g., C++17, GEOS=(3.4.0), GDAL= (2.2.3), PROJ (=4.9.3,sqlite3)), then use a large language model to generate the specific commands to resolve the system requirements iteratively.

R Markdown files are usually written for data analysis tasks, and data itself plays an important role as the R Markdown files. However, in RQ2, we observe that a remarkable portion of the files failed to execute due to missing data. We strongly encourage the authors of R Markdown files to make sure the data files are uploaded if possible. Authors should also carefully document any dependencies and system requirements to ensure that the files can be executed without problems.

3.5.2 Threats to Validity

Threats to external validity refer to the generalizability of our findings. In this study, we collected markdown files from GitHub repositories whose primary language is "R". To collect the list of GitHub repositories we used the GHTorrent dataset released on March 6, 2021. Thus, we may miss repositories containing markdown files that are not tracked by the GHTorrent dataset. Thus, the results from our study may not be generalized to other repository hosting services or other markdown files. However, we want to highlight the fact that GitHub is a popular hosting platform for Git repositories and we consider a large number of markdown files in this study. The repositories can be changed due to the evolution of those repositories (i.e., files can be added, deleted, or modified). However, we considered the latest version of the repository at the time of conducting the study. Thus, our results should largely carry forward.

Threats to internal validity refer to the bias and errors in conducting our studies. Due to the large number of markdown files, it is not feasible to manually analyze markdown files that are non-executable. Thus, we conducted the manual study on a sample rather than the entire population. One can argue that our manual study may not be representative. However, we would like to point to the fact that we consider a statistically representative sample for our manual study that ensures a 95% confidence level with a 5% confidence interval.

We conducted a manual study to understand the reasons that prevent us from executing those markdown files. To avoid any biases, two participants investigated the markdown files and labeled them separately. The high inter-rater agreement of our manual study indicates a high degree of reliability of the study. The manual study was conducted by two participants and it can be argued that this could lead to a biased result. However, both of the participants have extensive experience in programming and software development - more than five years in programming and three years specifically in R programming. They performed the manual study objectively, without any preconceived assumptions, and they made their decisions solely based on the collected data.

During our study, we did not perform any hyper-parameter tuning and considered the default hyper-parameter values. We cannot guarantee that this will not affect the classifier's performance, and therefore, further research is required in this area. This remains as a future work.

3.6 Conclusion

In this study, we investigate the executability of R Markdown files collected from open-source GitHub repositories. We investigate the relationship between features related to seven different dimensions and the executability status of those files. We find that 64.95% of Markdown files failed to execute. We observe that all the studied features, except the number of contributors and the number of open issues, are statistically significantly different between executable and non-executable groups of files. Results from our manual study show that library-related issues are the dominant factor that prevents the execution of Markdown files. We also show that the selected features can be used to build a classifier that can identify non-executable files with an accuracy of 92.69%. Such a classifier can warn users as they search online for R Markdown files to reuse. Authors should carefully document any dependencies and assumptions (e.g., version of R, package managers used, data, and session requirements) to facilitate the reuse. Tool developers should focus on developing tools and techniques to automatically correct runtime requirements of R Markdown files and R code. While several studies have been done in the Python ecosystem [10] [18], such support is considerably missing in the R ecosystem.

Chapter 4

RFT: A Smart Tool To Fix R Markdown Issues To Restore Executability

4.1 Introduction

R Markdown files act as solid examples of literate programming because they incorporate R code, explanations, and results in a way that makes it effortless to execute and reproduce study findings in documents. R Markdown files are meant to be reproducible but their executability is still a little-studied topic that concerns users. In the previous chapter, we investigated the executability of a large number of R Markdown files collected from GitHub and found that a significant percentage (64.95%) of those files were not executable, even with our best efforts. We investigated the errors that occurred during the execution of R Markdown files to better understand the complexities surrounding this problem. Our results emphasize the need for a more sophisticated understanding of the problems that prevent the execution of R Markdown files. As a result, we offer a new classifier that is intended to predict whether R Markdown files will be executable or not, without executing it. This classifier has potential applications in search engine optimization, improving the ranking of literate programming documents. It will help developers to find executable R Markdown files, which will help to save resources, time and cost by ignoring the non-executable files.

Through manual investigation, we identify reasons that prevent the execution of R Markdown files. These can be divided into three groups- a) Missing libraries b) Missing data c) Erroneous Code. Dependency issue is the most common issue, impacting a significant 44.18% of the files that we examined. If the libraries with their dependencies are not installed carefully, the complex network of dependencies between libraries can cause errors and make it difficult to execute R Markdown files. We also discovered cases where R Markdown files used libraries that were either deprecated, removed from CRAN, or rendered obsolete, which resulted in errors during execution. Apart from that, around 19% of files can not be executed due to missing directories or files. This problem frequently occurs when developers forget to change the local paths in their code or upload necessary files, making the referenced paths or files unavailable for correct execution. Finally, a subset of R Markdown files with execution problems caused by incorrect code was discovered by our analysis. Our results highlight the complex range of obstacles that hinder the execution of R Markdown files, including problems related to dependency handling, library accessibility, path setups, and code correctness. These various obstacles must be addressed to guarantee that R Markdown files run smoothly in different development environments.

Regarding dependencies, renv, Packrat, and devtools are the most popular tools in the R package management community for managing dependencies in R Markdown projects. Each tool offers a unique feature set and set of restrictions suited to various stages of the development process. Devtools is particularly useful for package development tasks because of its wide range of development tools and integration with GitHub. It is less appropriate for guaranteeing project reproducibility, though, since it is deficient in features for isolating project environments and in comprehensive dependency management. Packrat's ability to capture and manage dependencies unique to a project makes it highly reproducible. However, there is a higher learning curve and the packrat.lock file needs to be actively maintained by users, which could be a burden. Renv, on the other hand, offers stringent version control and thorough environment management while striking a compromise between development and reproducibility. To integrate with other tools, users can run into compatibility problems or need to make additional configurations. The particular use case, project size, and desired ratio of development flexibility to reproducibility assurance will determine which of these tools is best. Users must carefully consider these factors and choose the tool that best fits their workflow and project needs.

During our investigation, we found a flaw in renv's capacity to track R versions associated with packages. Although renv does a good job of tracking package versions, it is not as good at managing the R version, which leaves room for errors. Renv normally restores dependencies using the system's current R version rather than the version that was specified, which could lead to compatibility problems. Moreover, renv introduces a possible source of errors by not being able to determine the version compatibility between two packages. When renv is utilized in a project without a package manager, there is still another disadvantage. Under these circumstances, renv downloads the most recent package versions without confirming their compatibility, endangering the stability of the project. Developers must be made aware of any potential problems with the file for them to address and fix any compatibility or version-related problems, which will help to mitigate these problems. This proactive approach can make a big difference in how smoothly projects are developed and implemented.

Following a thorough investigation, it was found that a substantial large number of the files had problems, most of which were caused by library-related issues that prevented them from being executed. A more thorough investigation showed that these library errors extended beyond version-specific problems to include compatibility problems across various R versions. The problem surfaced when trying to install a library inside the confines of a fixed R version because the package manager, renv, only functioned within those restrictions. For example, if the system version was 4.3.1 and renv stored R version 4.0.4, it would try to restore dependencies in 4.3.1 rather than the necessary 4.0.4, which could lead to errors.

Additionally, our investigation produced instances of certain libraries that were incompatible with other libraries and could not be installed. Consider the conflict between the libraries scales (version 1.1.0) and ggplot2 (version 3.4.4), for example (Figure 4.1). The dependency error that appeared during the installation process was caused by the fact that scales version 1.2.0 or a later version is required for ggplot2 version 3.4.4. As a result, an error occurred because the scales version 1.1.0 was incompatible. The difficulties encountered highlighted the significance of carefully examining and managing dependency data in order to improve the system's overall effectiveness.

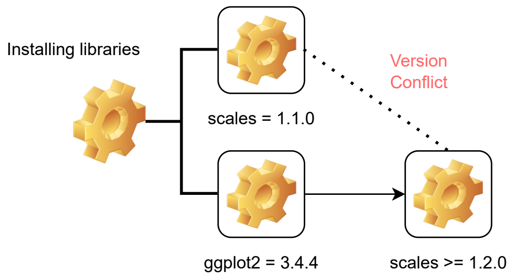


Figure 4.1: An example of version conflict.

In order to improve the publishing process for R Markdown files, we created a special tool named RFT (R Markdown Fixing Tool) that methodically evaluates each file and helps users to fix any issues before publishing. To determine compatibility between versions in dependency files, we use a knowledge base and the Z3 algorithm with SMT dependency constraints. We can find and fix compatibility problems with this methodical analysis, which sets the stage for a successful implementation. The use of Conda to install the appropriate R version before moving on to dependencies is a crucial component of our methodology. We've had a great deal of success with this approach, fixing dependency problems in a substantial portion of files that were previously not executed because of compatibility problems. We also provide support for system requirement identification, deprecated and deleted library detection, and file and directory path validation. These tools are intended to address and resolve potential issues that might occur. This creative solution guarantees a more seamless and effective workflow when working with R Markdown files by expediting the publishing process and enabling users to anticipate and address potential roadblocks.

4.2 Approach

We downloaded and attempted to run the R Markdown file in the previous section. We discovered a few common problems that prevent the R Markdown file from being executed. In this section, we develop a tool RFT (R Markdown Fixing Tool) (Figure 4.2) that, given an R Markdown file as input, will identify which files may not be reproducible and which parts require inspections to ensure their executability.

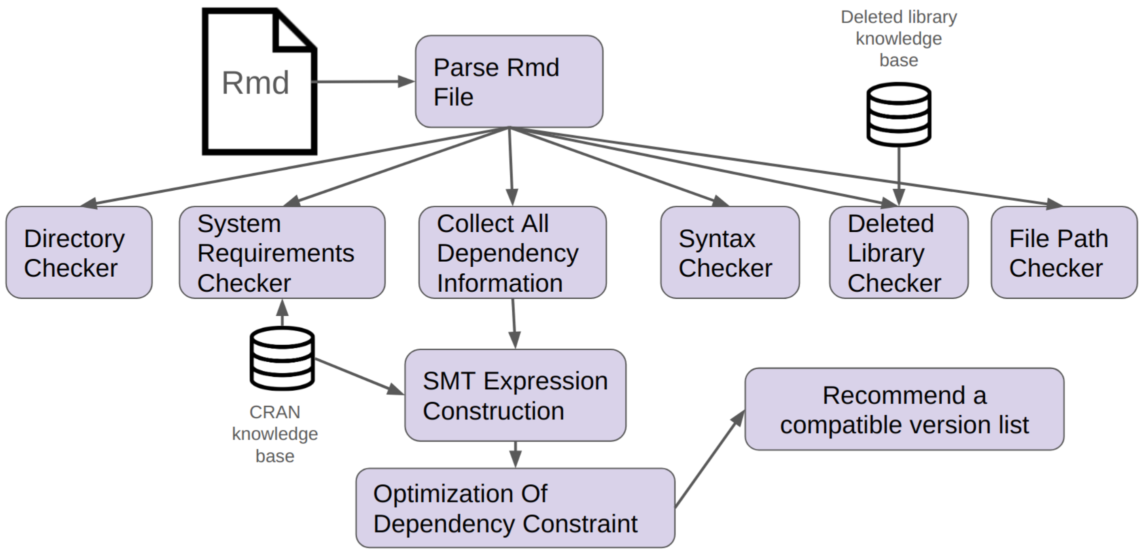
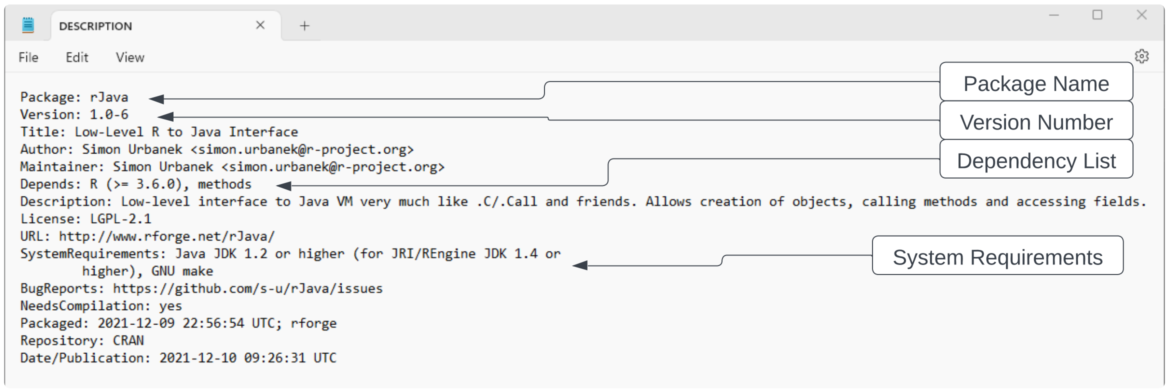

Rmd
Parse Rmd File
Deleted library knowledge base
Directory Checker
System Requirements Checker
Collect All Dependency Information
Syntax Checker
Deleted Library Checker
File Path Checker
CRAN knowledge base
SMT Expression Construction
Recommend a compatible version list
Optimization Of Dependency Constraint

Figure 4.2: The working process of RFT.

4.2.1 Knowledge Base Creation

Building a thorough knowledge base is essential in the early phases of our process to gain a deep comprehension of the complex dependencies linked to every software package. In order to achieve this, the Comprehensive R Archive Network (CRAN) is our chosen repository because it is the most popular repository to host R packages along with their versions.

Additionally, CRAN is a versatile platform that consists of web servers and FTP networks that are used to host and maintain a wide variety of R packages.



```
Package: rJava
Version: 1.0-6
Title: Low-Level R to Java Interface
Author: Simon Urbanek <simon.urbanek@r-project.org>
Maintainer: Simon Urbanek <simon.urbanek@r-project.org>
Depends: R (>= 3.6.0), methods
Description: Low-level interface to Java VM very much like .C/.Call and friends. Allows creation of objects, calling methods and accessing fields.
License: LGPL-2.1
URL: http://www.rforge.net/rJava/
SystemRequirements: Java JDK 1.2 or higher (for JRI/REngine JDK 1.4 or
        higher), GNU make
BugReports: https://github.com/s-u/rJava/issues
NeedsCompilation: yes
Packaged: 2021-12-09 22:56:54 UTC; rforge
Repository: CRAN
Date/Publication: 2021-12-10 09:26:31 UTC
```

Figure 4.3: Example of dependency package information.

Our process involves carefully extracting the most recent versions of the code for every library, which are easily accessible on CRAN. Additionally, in order to guarantee a comprehensive analysis, we broaden our scope by downloading earlier versions of these packages from a specified URL. Then, for each package version, a methodical examination is carried out, carefully examining the DESCRIPTION file to extract and record each dependency.

Figure 4.3 illustrates an example DESCRIPTION file containing comprehensive dependencies details of a package. The information is arranged logically in the file: “packages” contains the names of the packages, “versions” the version number of the packages, “depends” the list of libraries that the packages depend on, and “system requirements” the list of requirements needed to install before installing the packages. Our approach entails a methodical study of each package’s dependency file to collect these important facts systematically.

The gathered information is then combined with the corresponding version numbers to keep everything precise and clear as it is arranged and structured inside a dataframe. This knowledge base functions as a fundamental tool, providing an understanding of the complex web of dependencies supporting every package. Using CRAN as our main source allows us to have access to the most recent versions while also providing a complete historical view by incorporating earlier versions.

Upon closer inspection, CRAN (Comprehensive R Archive Network) contains a repository that houses 20,325 packages, each of which has multiple versions, making up a sizable dataset with 158,564 unique versions. We methodically downloaded all of the packages and recorded each one with its matching versions. We also examined dependencies in detail like figure 4.3, resulting in the creation of a centralized store of dependencies for 158,564 unique versions of those libraries.

To enhance the compatibility of SMT solvers with version constraints on dependency packages, we apply a methodical way to transform each constraint (e.g., *rJava==1.**) into an SMT expression. The first step is to query the knowledge base to get an ordered list of all the versions of the library in concern that were created when the knowledge base was constructed from CRAN. We then use this version information to identify the versions that satisfy the given constraint, and we use their index range to create an SMT expression. In the case of *rJava== 1.** for example, we can determine the versions *1.0-4, 1.0-5, 1.0-6,* and *1.0-10* by sorting the ordered list of 56 *rJava* versions. We transform *rJava== 1.** into the SMT expression *53 rJava 56* given that their index range is 53 to 56.

4.2.2 Collect Dependency Information

In the initial stage of our procedure, we start by carefully going over each file and methodically gathering data about the libraries that are used in them. A file usually depends on the libraries that have been imported into it. Packages are typically imported into R using methods like *require(package˙name)*, *library(package˙name)*, and *install.packages("package˙name")*. We extract and create an extensive list of the imported packages by using these patterns. This collected list is then utilized for compatibility testing, guaranteeing that these libraries install without any problems.

4.2.3 Solving Dependency Constraint

This section primarily explains how to gather all necessary library version constraints for a given R project. Additionally, we describe how to optimize version constraints.

4.2.3.1 SMT Expression Construction

We collect detailed dependency information while analyzing an R project using the procedure described in the "Collect Dependency Information" section. Then we gather all the transitively dependent libraries along with their version constraints by utilizing the local dependency knowledge base based on the directly dependent libraries. We construct a systematic approach to build the SMT expression which is inspired by a previous work [18]. The description of the process is given below.

First, the R Markdown files in the projects are parsed, and the import statements are looked at to determine direct dependencies. Then, TRUE is assigned as an initial value to the SMT expression for version constraints, denoted as Expr. After that, the program gathers the SMT expression, ExprLi, for each directly dependent library Li along with the dependent libraries for Li. ExprLi and the overall expression Expr are subsequently merged at each iteration using the conjunction operation.

Subsequently, the function LibraryAnalysis is utilized for this task, accepting three inputs: a collection of library names S, the version constraint VCi for Li, and the dependent library name Li. When a library Li and its necessary dependent libraries are provided, this function is in charge of producing the SMT expression of the version constraint. To avoid calculating library SMT expressions that have already been collected twice, the input set s is used. In order to prevent unnecessary computations, the function first determines whether the library Li is already in the set S before adding it. After analyzing the version constraint VCi using a knowledge base, it produces a list of versions, V. And after that ExprV, which is the SMT expression for the version constraint, is initially set to FALSE. The function retrieves the set of direct libraries D that are version-consistent with each version Vi in the list V. Then it initializes the SMT expression for direct libraries, ExprD, to TRUE. The function recursively calls itself on the direct library Di with its version constraint VCDi and the set Si for each tuple in D. Between ExprD and the recursive result ExprDi, the conjunction operation is executed to merge the data. For every version Vi, the disjunction of ExprV and ExprD is updated in the SMT expression ExprV. Lastly, the function returns the conjunction of ExprV and the original version constraint VCi after removing the library Li from the set S. The function returns its version constraint VCi directly if the library Li has already been resolved.

To summarise, the algorithm uses the LibraryAnalysis function to control the creation of SMT expressions for specific libraries and their dependencies, iteratively examines direct dependencies and the constraints they are linked to, and performs conjunction operations to systematically construct SMT expressions for version constraints.

4.2.3.2 Optimization of Dependency Constraint

Our method uses the most widely used open-source constraint-solving tool, Z3-solver, to solve the version constraints for the required libraries after gathering SMT expressions for them [52] [3]. Our effort is motivated by a prior study [18] in which they optimized the version constraints using Z3.

We use a methodical approach wherein Expr, an SMT expression, is the first input. The aim is to produce a list L containing the names of libraries with version restrictions (e.g., “rJava== 1.0-10”). It starts by extracting all library names from Expr and loops through each library in the resulting list, represented by LNi. Using the "AND" operation, it makes a copy of the original expression (Exprcopy) for each version Vi in the retrieved versions and incorporates the version constraint (LNi == Vi) into the expression. We will use the constraint LNi == Vi to preserve the version Vi if it is satisfiable.

4.2.4 Handling the Installation of Libraries

A workable solution for the compatible versions of the necessary libraries is provided by the step that solves dependency constraints. Then, our method turns this workable solution into a dependency configuration file that can be used to successfully install the necessary libraries. For example, the appropriate library versions for the project client are Lib-X 1.0.1, Lib-Y 2.0.1, and Lib-Z 3.0.1. Our method transforms these library versions into the dependency configuration file, "RVersionRequirements.lock", containing the equations Lib-X = 1.0.1, Lib-Y = 2.0.1, and Lib-Z = 3.0.1. Afterward, it creates a compatible installation script for the libraries to be downloaded.

4.2.5 Identifying Erroneous Code (SyntaxChecker)

After a R Markdown file is examined, our procedure entails determining whether any incorrect code is present. Finding semantic errors is difficult because every R Markdown file is created with a specific purpose in mind, making it challenging to recognize logical problems without having an in-depth knowledge of the reasoning behind them. Our main goal is to detect syntactical errors within the file.

To overcome this issue, we parse the R Markdown file using Antlr [53]. When parsing succeeds, it means there are no problems with the file, which is a sign of its integrity. On the other hand, if an error occurs during the parsing process, our tool notifies users with detailed information regarding the type of error that occurred, including the precise location of error. Equipped with this useful data, users can quickly locate the appropriate line and handle any problems that may arise. This meticulous process guarantees that the R Markdown file is corrected accurately and is prepared for flawless execution.

4.2.6 Checking Missing Data (FilePathChecker & DirectoryChecker)

Around 19 percent of files have issues with non-execution, which are mainly caused by missing data. This includes situations in which execution fails because referenced directories or files are not in the path that is supplied. The analysis finds gaps, like missing directories that are expected to change the set directory but might only be present on the system of the creator. It is crucial to resolve these differences in order to make sure smooth file execution and proper system operation.

To address the first issue, we use a comprehensive approach for collecting file path information via file parsing. Then, our program performs a methodical check to verify that these files actually exist. It will provide no issues when files are found in their specified paths. When the program fails to locate a file, it offers comprehensive information indicating the exact location of the absent file along with the line that corresponds to it in the code. Programmers sometimes use absolute file paths rather than relative ones, which makes the file easier to identify locally but makes it irreproducible on other systems. Our tool finds instances in which developers employed full paths rather than relative paths in order to address this. After that, developers are quickly alerted to change the path based on our tool's instructions, guaranteeing smooth file reproducibility. This painstaking procedure not only guarantees that no files are missing but also encourages developers to use relative paths, which makes the codebase more widely reproducible.

To overcome the second challenge, we collected and inspected every path provided in the file. To confirm that each path existed, verification was done programmatically. As long as a folder path was confirmed as existing, there was no cause for concern during this assessment. But if a folder was missing from the system, a notification was sent out that identified the exact line that needed to be changed in the path. Similar to the method used in the first problem, this comprehensive examination included the verification of actual paths. Users were alerted immediately by our tool if absolute paths were found, suggesting that they needed to replace these with relative paths to improve file executability. This comprehensive path verification procedure is implemented by the system to guarantee not only the integrity of file references but also to assist users in resolving possible discrepancies, thus enabling a more streamlined and dependable file reproduction process.

4.2.7 Identifying Deleted and Outdated Libraries (DeprecatedLibraryChecker & DeletedLibraryChecker)

Identifying obsolete and deleted libraries in an R Markdown file involves creating an extensive database with details about those libraries. To do this, CRAN's library data has been reviewed closely, and information about libraries that have their documentation removed or out-of-date is only collected. The task also includes the incorporation of data from the Library.io dataset, containing packages from multiple package managers. The dataset helps gather data on deleted libraries while focusing specifically on packages written in R. The result of these efforts is the development of a single database that incorporates obsolete and deleted libraries. This combined database is a crucial resource for later tasks, especially when it involves finding out which libraries are out-of-date in R Markdown files. Using this painstaking curation procedure, the system guarantees a strong base for detecting and dealing with out-of-date libraries, supporting maintenance and enhancement of R Markdown files.

Our tool parses library data extracted from package import statements to process R Markdown files. After that, it compares this data with the database of obsolete and deleted libraries. The tool alerts the file's creator if a match has been found, indicating unavailability. Because the specified libraries are either removed or out-of-date, this proactive approach intends to prevent users from encountering execution issues when they attempt to download and execute the file. We help make the R Markdown files more reproducible by encouraging the use of up-to-date and available dependencies by warning authors about problematic packages. This system ensures that users can reliably replicate the file without running into issues with outdated or missing libraries, which enhances the code's overall effectiveness and dependability.

4.2.8 Identifying System Requirements (SystemRequirementsChecker)

Based on our investigation, 44.18% of R Markdown files have irreproducibility problems because the system requirements of the dependent libraries are not met. One prominent instance is the usage of rJava, a low-level R to Java interface, that requires certain system requirements like GNU make and Java JDK 1.2 or higher (for JRI/REngine JDK 1.4 or higher). If the above requirements are not satisfied, rJava cannot be installed, making the entire R Markdown file non-executable. It is important for developers to be aware of dependency requirements in order to improve the executability of their projects. By addressing these problems, the community can collaborate more easily and consistently while building a more solid and dependable ecosystem for R Markdown files.

As part of our strategy, we extracted dependency information and associated it with the system requirements in a methodical manner to tackle the current challenge. Our approach is based on an in-depth web crawling of package pages from CRAN in order to create an extensive database of relevant data. Notably, each page has a particular "System-Requirements:" section that contains the essential system requirements information. We ensure a thorough coverage with our comprehensive data collection process, which has been specifically designed to fit the packages that are available on CRAN. The resulting System-Requirements database is carefully archived for later use. This well-thought-out compilation is a useful tool that makes it easier to find and use crucial system-related data. In addition to resolving the immediate issue, our strategy lays the groundwork for expedient access to critical information, highlighting the importance of a clean, easily accessible database in negotiating the complex web of interdependencies between systems.

When working with an R Markdown file, the first step is to parse the file and create a list of packages that are used. The next step is to investigate the SystemRequirements library to find out which particular system requirements are needed for every package that has been found. All of this data is then combined into a single file called, SystemRequirements.txt. The principal aim of this process is to provide users with an open understanding of the necessary system requirements. This information makes it easy for those who are trying to download and run the file to determine if their system is compatible with the requirements. This technique basically acts as a helpful manual, allowing users to proactively confirm that the related R Markdown file operates as intended by making sure all required system requirements are met.

4.3 Experimental Setup

We evaluate the effectiveness of our approach in resolving dependency conflict (DC) problems on publicly available R projects which was collected previously in the dataset creation section. The following research questions are the main focus of our evaluation.

4.3.1 Can our process resolve the DC issues successfully?

This research question evaluates the effectiveness of our approach in resolving data conflict (DC) issues through an analysis of dependency library installations on our dataset. We compare our approach with a baseline that installs all dependency libraries sequentially using scripts. We also compare our procedure with the renv project manager, which is a dependency management tool used by the chosen project. The renv.lock files facilitate the restoration of dependencies. In order to shed light on potential improvements for DC issue resolution, we hope to ascertain the effectiveness and benefits of our approach relative to the renv project manager and the baseline method through this investigation.

One of the most important parts of our assessment of the “renv” project is the examination of the R Markdown file, which contains the dependencies for the project. Using this file as input, the process first extracts the necessary dependencies from the import statements. Moreover, we collect these libraries’ direct dependencies with great care, understanding that they are directly and indirectly essential to the project’s functionality.

For every library, we create version constraints in order to simplify the complex web of dependencies. This is a crucial step in making sure the project is cohesive and compatible.

Subsequently, these transitive and direct dependencies are transformed into SMT expressions, thereby laying the groundwork for an organized resolution of dependency constraints. In this optimization process, the Z3 solver shows up as a crucial tool because of its ability to resolve possible conflicts between various packages.

Utilizing the output of the Z3 solver, we create a script after the dependencies have been harmonized. Providing a clean, streamlined base for the "renv" project, this script contains the package names and their matching version numbers. The reliability and stability of the project are largely attributed to this methodology, which guarantees an organized and effective management of dependencies.

Using Conda, we installed the required R version in the project folder. We then install the necessary library versions from within that folder using a pre-generated script. Afterward, we methodically render each R Markdown file, automating the procedure and compiling all of the results. This methodical implementation is an important checkpoint in assessing the effectiveness of our strategy, offering a comprehensive view of its operation.

By using the baseline approach to evaluate our algorithm in our study, we improve the methodology's robustness. To identify imported libraries, the baseline method parses R Markdown files. Afterward, we install these libraries one by one and render each R Markdown file, documenting the results. This set of findings is our starting point, making it easier to compare our suggested algorithm comparatively. Establishing this baseline provides a point of comparison for performance evaluation and gives us important insights into the efficacy and efficiency of our strategy. The thorough analysis of our algorithm's improvements over traditional techniques is ensured by this all-encompassing evaluation strategy, which strengthens the validity of our research findings.

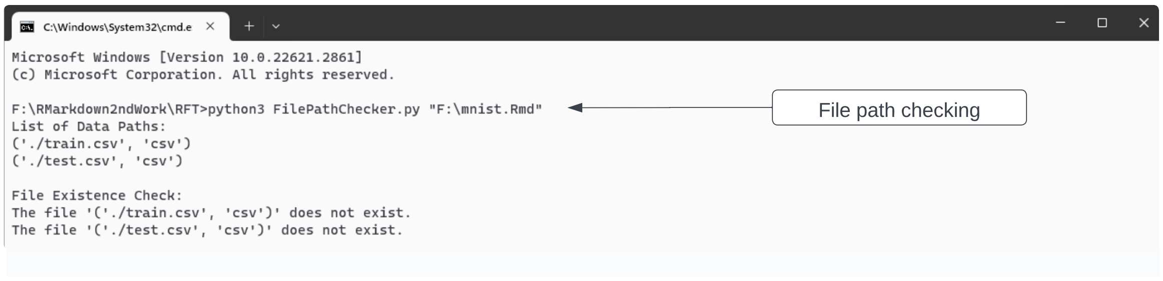
We thoroughly compared our methodology in our study with the well-known renv project manager, a common tool of the projects that we chose to evaluate. First, we used renv to restore the project dependencies. Then, we rendered all R Markdown files and collected the results with care. Notably, we organized the results for later review and analysis, which strengthened the comparison between the two methods. This strategic investigation emphasizes our method's efficacy and places it in the perspective of current practices.

Our tool consists of six components: a directory checker, system requirement checker, syntax checker, deleted library checker, file path checker, and version compatibility checker. With the first five components, we can easily identify issues and notify users to modify the necessary lines. We identified all the issues related to these checkers and with a simple effort, we were able to execute all the files that couldn't be executed due to these issues. To evaluate the last checker, we randomly chose 20 R Markdown files that couldn't be executed due to library compatibility issues. After applying our tool, we collected compatible versions of libraries for each file and applied those compatible lists. As a result, we were able to re-execute 13 files, indicating that our technique is working and we can now execute files that previously couldn't be executed due to library issues.

4.4 Usage Scenario

Bob is eager to become an expert in the area of machine learning and data analysis. He looks for real-world code examples to help him learn, and he has collected 20 R markdown files that show his go-to method in action. It takes time to go through each of these files, and not all of them might be executable. Bob can choose to use our tool in this case to effectively re-execute the needed files.

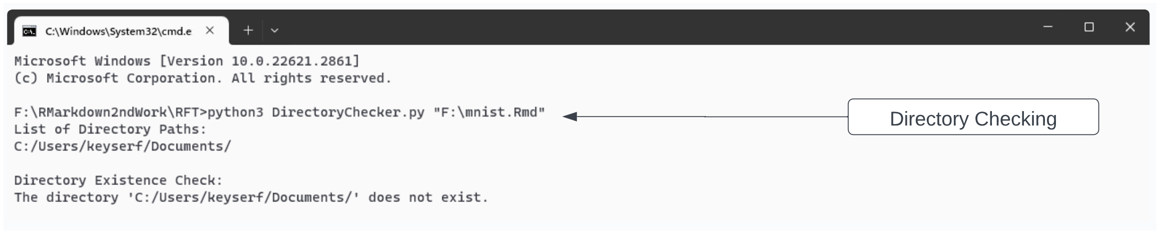
Bob starts the tool by giving it an R Markdown file to work with, carefully going over each output to make sure the file will execute in the future. The tool looks over every project file in the first stage. The program alerts Bob to change the directory for successful execution if any files reference paths that are not available on his machine. As an example in figure 4.4, Bob has chosen an R Markdown file that consists of two files: "./train.csv" and "./test.csv." By capturing these files and alerting Bob about their absence, the program helps Bob change the paths so that the results are reproducible.



```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 FilePathChecker.py "F:\mnist.Rmd"
List of Data Paths:
('./train.csv', 'csv')
('./test.csv', 'csv')

File Existence Check:
The file '('./train.csv', 'csv')' does not exist.
The file '('./test.csv', 'csv')' does not exist.
```

Figure 4.4: Checking File Paths.

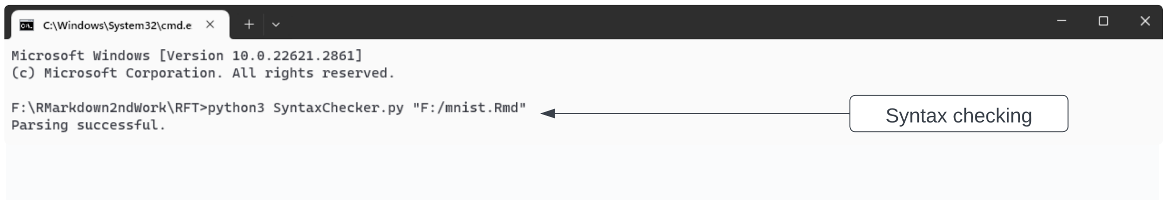
The tool searches files for folder paths that are necessary to modify the current working directory. When these files are missing, Bob gets notified and has to change the directory. To change the directory for a certain task, for example in figure 4.5, certain paths like "C:/Users/keyserf/Documents/" and "C:/Users/keyserf/Desktop/asset˙allocation/" are used. Interestingly, these routes only exist within the file system of the developer, which presents a problem with universality. Bob receives the notice as a preventive action to make sure the files are flexible and rerun. By locating and resolving directory anomalies, this automated procedure strengthens the system's resilience and highlights how crucial path alignment is to ensure smooth operation across several file systems.



```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 DirectoryChecker.py "F:\mnist.Rmd"
List of Directory Paths:
C:/Users/keyserf/Documents/

Directory Existence Check:
The directory 'C:/Users/keyserf/Documents/' does not exist.
```

Figure 4.5: Checking Directory Paths.

Parsing a file is also attempted by our tool using pypandoc library [54], and if there are no parsing problems, the procedure is considered successful (figure 4.6). The tool will alert Bob and provide the line number if an error arises while parsing. It will also highlight the exact line where the problem is found. This makes it possible for Bob to find and fix the file's issue quickly. This result will help Bob to make this file executable again.

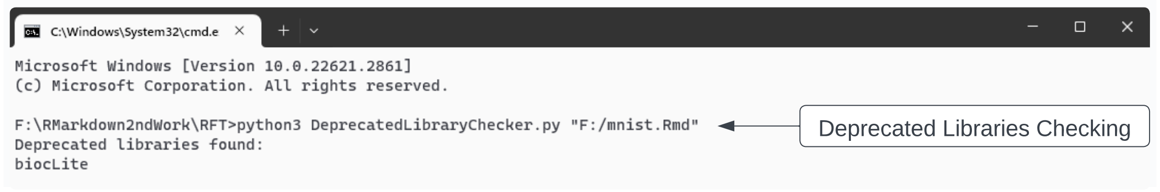


```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 SyntaxChecker.py "F:/mnist.Rmd"
Parsing successful.
```

Figure 4.6: Checking Parsing the file.

The next procedure entails parsing a file, locating and gathering imported packages, and then determining which of these packages are deprecated. To do this, a pre-existing deprecated library list is compared to the list of imported packages. Bob receives a notification if a deprecated library is found, encouraging him to replace the deprecated library with a suitable one.

For example in figure 4.7, Bob chose an R Markdown file that used the biocLite function to install the package pragmatically from Bioconductor. However, the function is now outdated and BiocManager is now used to install any libraries from Bioconductor for R version 3.5 or greater. Here, the developer used the outdated function to install the library. Our tool will identify the library and show the list of deprecated libraries so that he can replace the library and make the file executable again.

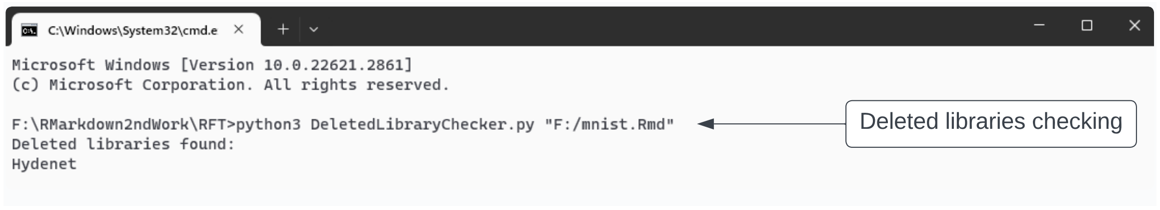


```
C:\Windows\System32\cmd.e

Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 DeprecatedLibraryChecker.py "F:/mnist.Rmd"
Deprecated libraries found:
biocLite
```

Figure 4.7: Checking deprecated library.

The procedure also involves deciding whether libraries are used in the source code and locating any library deletions by contrasting the library with a list that has already been prepared based on deletion. Bob receives information if a library is discovered to be removed, and he is then prompted to change the code by replacing the missing library to keep the file executable.

In illustration (Figure 4.8), HydeNet was used in an R Markdown file which is selected by Bob but is now removed from CRAN. When Bob tries to install the dependency, it will produce an error because the target library is no longer available. By using our tool Bob can identify the deleted library easily so that he can replace the library with a suitable one to make the file executable again.

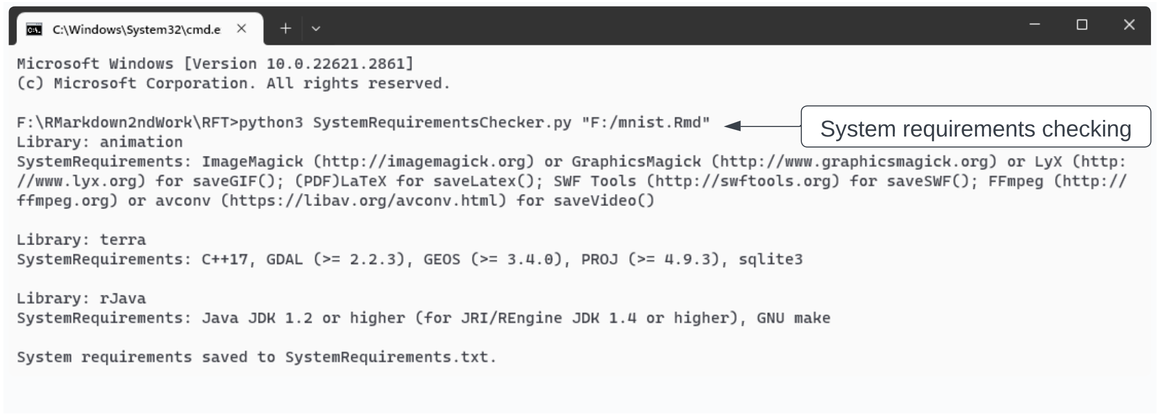


```
C:\Windows\System32\cmd.e

Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 DeletedLibraryChecker.py "F:/mnist.Rmd"
Deleted libraries found:
Hydenet
```

Figure 4.8: Checking deleted library.

The tool methodically compiles the libraries indicated in the R Markdown file that Bob supplied. It then gathers needs for the system from an already existing library database. Then the tool generates a file called **systemRequirements.txt** that contains all of the system dependency information of those libraries.

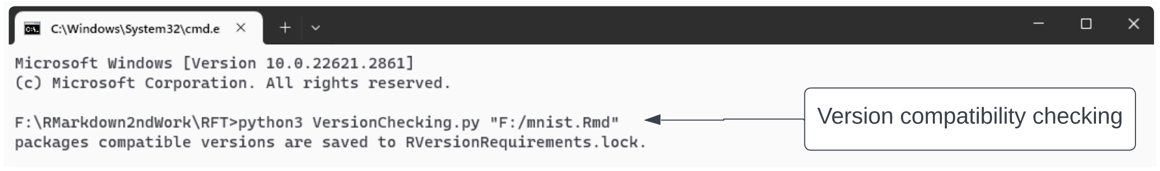
Bob has supplied an R Markdown file in this scenario that makes use of the animation, terra, and rJava packages. Before installation, certain system prerequisites for each of these libraries must be satisfied. To simplify this procedure, our tool creates a systemRequirements.txt file that includes all of these libraries' dependencies (Figure 4.9). After reviewing and meeting these system prerequisites, Bob will be able to successfully install the libraries and rerun the files as necessary. This method streamlines the setup and makes it easier for the R Markdown file to run without a hitch.



```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 SystemRequirementsChecker.py "F:/mnist.Rmd"
Library: animation
SystemRequirements: ImageMagick (http://imagemagick.org) or GraphicsMagick (http://www.graphicsmagick.org) or LyX (http:
//www.lyx.org) for saveGIF(); (PDF)LaTeX for saveLatex(); SWF Tools (http://swftools.org) for saveSWF(); FFmpeg (http://
ffmpeg.org) or avconv (https://libav.org/avconv.html) for saveVideo()

Library: terra
SystemRequirements: C++17, GDAL (>= 2.2.3), GEOS (>= 3.4.0), PROJ (>= 4.9.3), sqlite3

Library: rJava
SystemRequirements: Java JDK 1.2 or higher (for JRI/REngine JDK 1.4 or higher), GNU make

System requirements saved to SystemRequirements.txt.
```

Figure 4.9: Checking System Requirements.

The next procedure involves retrieving library information from designated R Markdown files, extracting all package versions, transforming the extracted information into version constraints, and using the Z3 solver to find a set of compatible versions. The “RVersionRequirements.lock” file contains the saved results (Figure 4.10). Bob then installs the libraries using this lock file, checks that they are compatible, and runs the files again.



```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

F:\RMarkdown2ndWork\RFT>python3 VersionChecking.py "F:/mnist.Rmd"
packages compatible versions are saved to RVersionRequirements.lock.
```

Figure 4.10: Checking versions.

4.5 Threats to Validity

Threats to external validity refer to the generalizability of our findings. We built our knowledge base by using packages from CRAN, which is a popular repository for R package publications. On the other hand, we noticed that R packages are also available on GitHub [55] and Bioconductor [56]. Thus, R packages located in those locations are not present in our database. If an R Markdown file depends on those packages, we will not be able to determine the required versions of those packages. This remains as a future work.

The evaluation of the dependency compatibility checker was done on 20 files. Therefore, it doesn’t represent the whole scenario, and the results may vary on a large dataset. Our future plan is to evaluate our technique on a large dataset.

4.6 Conclusion

In this study, we developed a tool called RFT (R Markdown Fixing Tool) to enable users to address issues with R Markdown files before publishing, to streamline the publishing process for these files. We employ a knowledge base and the Z3 algorithm with SMT dependency constraints to find compatible package versions in dependency files. With this thorough study, compatibility issues can be found and fixed, paving the way for a successful deployment. An important part of our process is to install the appropriate R version using Conda before going on to dependencies. This strategy has helped us solve a significant number of files' dependency issues that were previously impeding execution due to compatibility issues. In addition, we offer additional parsers for file and directory path validation, deprecated and removed library detection, and system requirement identification. These are the kinds of tools that are meant to deal with and fix possible problems. By accelerating the publication process and empowering users to anticipate and address possible obstacles, this innovative approach ensures smoother and more efficient executability while working with R Markdown files.

Chapter 5

Conclusion

In this chapter, the thesis comes to a close with a summary of its contributions in Section 5.1. A summary of the significance of this thesis and a roadmap for future research are provided in Section 5.2.

5.1 Summary

In this study, we investigate the executability of R Markdown files collected from open-source GitHub repositories. We investigate the relationship between features related to seven different dimensions and the executability status of those files. We find that 64.95% of R Markdown files failed to execute. We observe that all the studied features, except the number of contributors and the number of open issues, are statistically significantly different between executable and non-executable groups of files. Results from our manual study show that library-related issues are the dominant factor that prevents the execution of Markdown files. We also show that the selected features can be used to build a classifier that can identify non-executable files with an accuracy of 92.69%. Such a classifier can warn users as they search online for R Markdown files to reuse. Authors should carefully document any dependencies and assumptions (e.g., version of R, package managers used, data, and session requirements) to facilitate the reuse. Tool developers should focus on developing tools and techniques to automatically correct runtime requirements of R Markdown files and R code. While several studies have been done in the Python ecosystem [10] [18], such support is considerably missing in the R ecosystem.

To help developers reuse R Markdown files available online, we offer a command-line tool that is intended to identify possible problems in R Markdown files. The program takes an R Markdown file as an input and produces an easy-to-use list of outputs that help developers identify areas that need their attention before publishing their work online to improve the executability of the R Markdown file. The tool utilizes static analysis to identify missing data and coding issues (i.e., syntax errors). The tool depends on a knowledge base of package dependencies and their version constraints created by mining open-source R packages available online. It leverages an SMT solver (i.e., Z3) to identify compatible versions of used packages.

5.2 Future Work

Prospective directions for further improvement and refinement in this field of study are promising for future research. Based on our present discoveries and the created instrument, multiple avenues for subsequent research can be investigated:

- **Extending the Knowledge Base:** We used CRAN packages as the foundation for our knowledge base. The knowledge base includes package dependencies and system requirements, which our tool uses to identify compatible versions of used packages. R packages are also stored on two more locations, GitHub [55] and Bioconductor [56]. Future work should focus on extending the knowledge base to include packages from diverse repositories like GitHub [55], and Bioconductor [56]. This extension will enhance the tool's capacity to handle a greater variety of scenarios and offer a more thorough understanding of dependency management.

- **Analyzing and Addressing System Requirements:** Our investigation showed that a large number of R Markdown files are not being executed because of the system requirements imposed by used packages. For example, "terra" which provides methods for spatial data analysis with vector and grid data, has specific system requirements (C++17, GEOS=(3.4.0), GDAL= (2.2.3), PROJ (=4.9.3,sqlite3)). If the system requirements are not satisfied before installing the library, it will produce an error. Future research can look more closely at the system requirements. By utilizing large language models, we can produce targeted scripts that can automatically fulfill these system prerequisites, guaranteeing more efficient code execution.

- **Integrating with GitHub:** One exciting direction for future work is to create add-ons for our tool that integrate with GitHub seamlessly, which will improve user experience and expedite the development process. The ability to instantly check uploaded R Markdown files for possible problems can be made possible through this integration. Such an approach can offer a proactive solution by automatically identifying and resolving dependency issues.

- **Resolving Issues on Dependency Management:** Future work can concentrate on improving the tool's ability to automatically resolve dependency-related issues discussed in issue repositories, building on the automated checks carried out through GitHub integration. To lessen the amount of manual user intervention necessary, the tool may do this by updating packages or by installing any missing dependencies.

- **Supporting Automatic Graders for Programming Classes:** While the primary focus of this thesis is on R Markdown files, the results obtained and the techniques developed can be applied in other areas. An example of this is the grading of student assignments where they submitted code examples to solve programming challenges. This can be accomplished through an automatic grading mechanism that requires the execution of the submitted code examples. Even if those code examples are written in different languages, we anticipate a high degree of similarity of reasons that prevent their executability. Thus, the solutions developed in this thesis can lay the foundation to build a more robust tool.

Appendix A

Automatically Execute R Markdown Files

In this section, we provide a detailed guide on how to automatically execute R Markdown files. We explain each of the data files, which contain the necessary data, and the scripts that outline the process of executing those files. Furthermore, we have also included the necessary commands to setup the system before we run the execution of R Markdown files.

Along with R Markdown files that execute in less than 10 minutes, the file **AutomaticallyExecuteR_MarkdownFiles/ProjectList.csv** includes a carefully curated list of projects that have been filtered from the GHTorrent Dataset. Cloning the dataset and following the instructions will allow to replicate the results. An illustrative example is given below-

Table A.1: Important file list to automatically execute R Markdown files.

File Name	**Description**
ProjectList.csv	Contains the git paths of the projects
cloneProjects.py	Clone all the projects using ProjectList.csv
AutomaticallyExecuteR_MarkdownFiles.py	Render all the R Markdown files from those projects

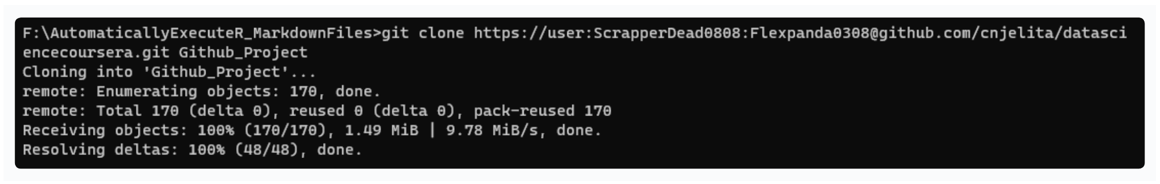
For illustration, consider the repository **https://github.com/cnjelita/datasciencecoursera** where multiple R Markdown files are located. Our goal is to execute those files automatically and store the output.

Installing the required dependencies and cloning the project are the first steps in starting the procedure. Cloning requires that Git be installed beforehand, which you can do with the following command.

```
# Install Git
sudo apt-get update
sudo apt-get install git
```

After that, the terminal command which is given below can be used to clone the repository. Put your Git login credentials in place of **UserName** and **Password**. The **Github_Project** folder contains a cloned version of the project. Adjust the path if you have a preference for a different directory.

```
git clone https://user:UserName:Password@github.com/cnjelita/datascien-
cecoursera.git ''Github_Project''
```



```
F:\AutomaticallyExecuteR_MarkdownFiles>git clone https://user:ScrapperDead0808:Flexpanda0308@github.com/cnjelita/datasci
encecoursera.git Github_Project
Cloning into 'Github_Project'...
remote: Enumerating objects: 170, done.
remote: Total 170 (delta 0), reused 0 (delta 0), pack-reused 170
Receiving objects: 100% (170/170), 1.49 MiB | 9.78 MiB/s, done.
Resolving deltas: 100% (48/48), done.
```

Figure A.1: Cloning repositories.

In order to automate operations, the process involves systematically installing the required libraries and running R Markdown scripts, all while maintaining an efficient workflow. First things first, installing the necessary libraries is a must to enable this automated process.

Make sure Python is installed before attempting to automate the process. Visit the official Python website to obtain the most recent version: **https://www.python.org/downloads/**. Alternatively, download Python 3.11—the version that is used for our particular procedure—by running the following command.

- **Update the package list:**

  ```
  sudo apt update
  ```

- **Prerequisites Install:**

  ```
  sudo apt install -y build-essential zlib1g-dev libncurses5-dev
  libgdbm-dev libnss3-dev libssl-dev libreadline-dev libffi-dev
  libsqlite3-dev libbz2-dev liblzma-dev tk-dev libdb-dev
  ```

- **Python source code Download:**

```
mkdir ~/python-3.11  # Create a directory for the Python source code
cd ~/python-3.11
wget https://www.python.org/ftp/python/3.11.0/Python-3.11.0.tgz
```

- **Extract the file:**

```
tar -xf Python-3.11.0.tgz
```

- **Configure and build:**

```
cd Python-3.11.0
./configure --enable-optimizations
make -j$(nproc)
```

- **Python install**

```
sudo make install
```

- **Installation verify**

```
python3.11 --version
```

After that, we need to install pip and upgrade the setuptools so that we can install the required libraries using pip.

- **Install pip**

```
sudo apt install python3-pip
```

- **Upgrade setuptools using pip**

```
pip install --upgrade setuptools
```

- **Install the required build tools**

```
sudo apt install build-essential
```

- **Install rpy2**

```
pip install rpy2==3.4.5
```

- **Install pandas**

```
pip install pandas
```

- **Install tqdm**

  ```
  pip install tqdm
  ```

You must have R installed on your computer in order to render R Markdown files. Get it from the official R Project website (**https://www.r-project.org/**) or use the supplied command to install version 4.1.2.

- **The CRAN repository add**

  ```
  sudo apt update
  sudo apt install -y software-properties-common
  sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux-
  /ubuntu focal-cran40/'
  ```

- **Add the CRAN GPG key**

  ```
  sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
  E298A3A825C0D65DFD57CBB651716619E084DAB9
  ```

- **Install R**

  ```
  sudo apt install r-base=4.1.2-1.2004.0
  ```

- **Verify the installation**

  ```
  R --version
  ```

- Alternatively, you can use the below command to install R. After executing the below code, it will install R version 4.1.2 in your system. As no version is mentioned there, it will automatically install the latest stable version and it will vary from time to time.

  ```
  sudo apt install r-base-core
  ```

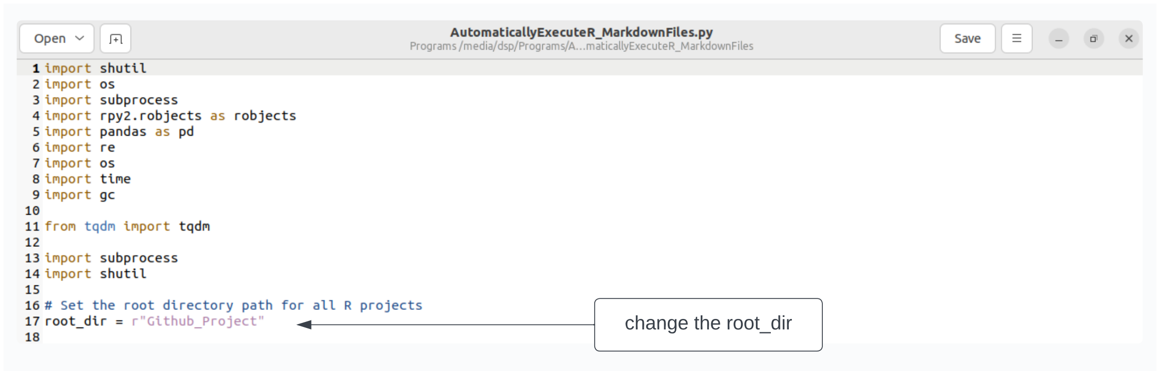
- The script uses various R libraries. To install them, open a terminal and run the following commands:

  ```
  Rscript -e ''install.packages('renv')''
  Rscript -e ''install.packages('devtools')''
  Rscript -e ''install.packages('rmarkdown')''
  ```

Installing Pandoc on your machine is necessary in order to get ready to render the R Markdown file. Pandoc is a flexible tool for converting documents between different markup and document formats. When it comes to R Markdown, Pandoc is frequently utilised as the underlying engine to transform R Markdown files into a variety of output formats, including Word, HTML, and PDF. Use the code supplied below to ensure a smooth installation.

```
sudo apt-get update
sudo apt-get install pandoc
```

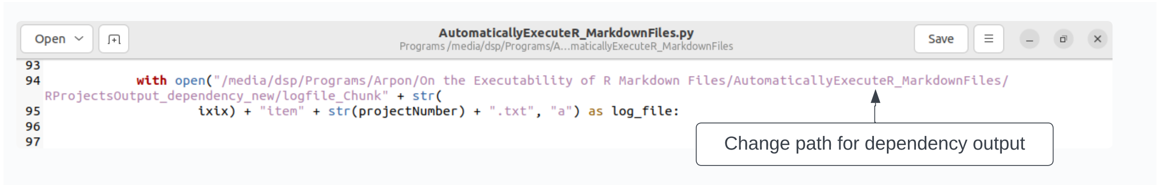
Then Run the **AutomaticallyExecuteR_MarkdownFiles/AutomaticallyExecuteR-_MarkdownFiles.py** script to execute the R Markdown files automatically. To run the script you need to change the **root_dir** to the folder where the projects are cloned previously.


Open
AutomaticallyExecuteR_MarkdownFiles.py
Programs /media/dsp/Programs/A...maticallyExecuteR_MarkdownFiles
Save
change the root_dir

```
1 import shutil
2 import os
3 import subprocess
4 import rpy2.robjects as robjects
5 import pandas as pd
6 import re
7 import os
8 import time
9 import gc
10
11 from tqdm import tqdm
12
13 import subprocess
14 import shutil
15
16 # Set the root directory path for all R projects
17 root_dir = r"Github_Project"
18
```

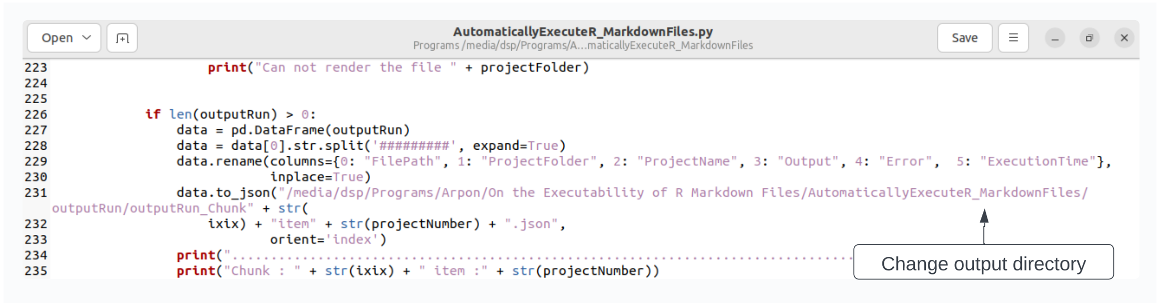
Figure A.2: Changing the root directory in line number 17.

The next step is to change the dependency output destination directory to the location where you want the dependency installation results to be kept. Additionally, you must specify where you wish to save the rendered results of each markdown file in the output directory.


Open
AutomaticallyExecuteR_MarkdownFiles.py
Programs /media/dsp/Programs/A...maticallyExecuteR_MarkdownFiles
Save
Change path for dependency output

```
93
94            with open("/media/dsp/Programs/Arpon/On the Executability of R Markdown Files/AutomaticallyExecuteR_MarkdownFiles/
RProjectsOutput_dependency_new/logfile_Chunk" + str(
95                    ixix) + "item" + str(projectNumber) + ".txt", "a") as log_file:
96
97
```

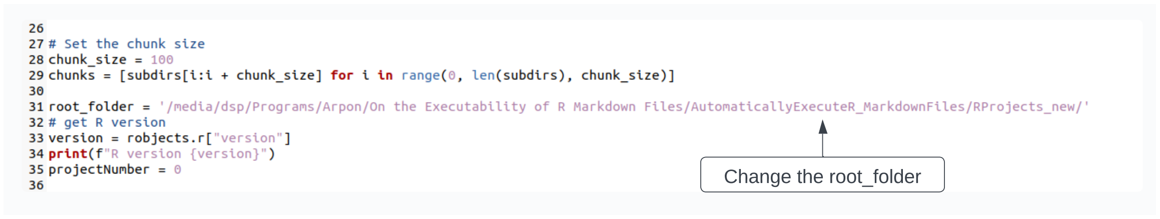
Figure A.3: Changing the dependency output destination in line number 94.


Open
AutomaticallyExecuteR_MarkdownFiles.py
Programs /media/dsp/Programs/A...maticallyExecuteR_MarkdownFiles
Save
Change output directory

```
223                     print("Can not render the file " + projectFolder)
224
225
226             if len(outputRun) > 0:
227                 data = pd.DataFrame(outputRun)
228                 data = data[0].str.split('#########', expand=True)
229                 data.rename(columns={0: "FilePath", 1: "ProjectFolder", 2: "ProjectName", 3: "Output", 4: "Error",  5: "ExecutionTime"},
230                             inplace=True)
231                 data.to_json("/media/dsp/Programs/Arpon/On the Executability of R Markdown Files/AutomaticallyExecuteR_MarkdownFiles/
outputRun/outputRun_Chunk" + str(
232                     ixix) + "item" + str(projectNumber) + ".json",
233                             orient='index')
234                 print("..................................................................................
235                 print("Chunk : " + str(ixix) + " item :" + str(projectNumber))
```

Figure A.4: Changing the output directory in line number 231.

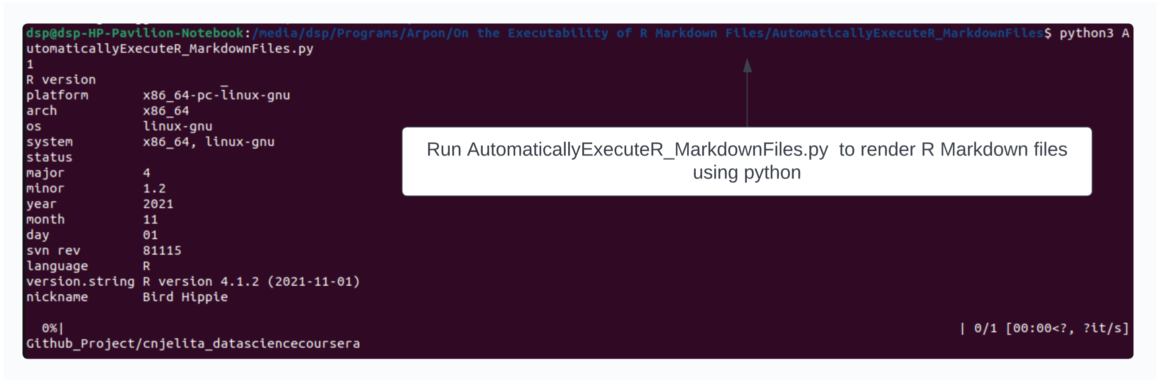
Lastly, it is essential to edit the **root_folder** to set the installation and rendering locations for every Markdown file. The purpose of this directory change is to preserve the original directory, from which the files are cloned, for potential future study. To ensure a smooth workflow, the project will be copied to the specified directory before it is executed.



```
26
27 # Set the chunk size
28 chunk_size = 100
29 chunks = [subdirs[i:i + chunk_size] for i in range(0, len(subdirs), chunk_size)]
30
31 root_folder = '/media/dsp/Programs/Arpon/On the Executability of R Markdown Files/AutomaticallyExecuteR_MarkdownFiles/RProjects_new/'
32 # get R version
33 version = robjects.r["version"]
34 print(f"R version {version}")
35 projectNumber = 0
36
```

Figure A.5: Changing the directory in line number 31.

The next action is to execute the “AutomaticallyExecuteR_MarkdownFiles.py” script, which installs dependencies by utilizing the relevant package management and renders each markdown file before saving the result.

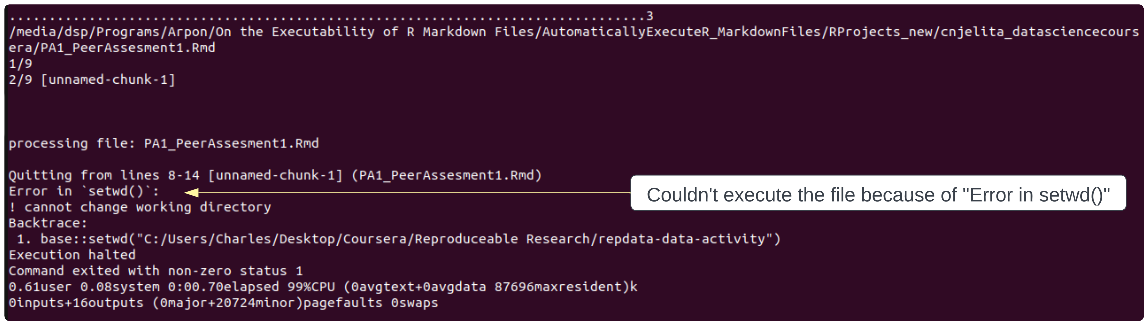


```
dsp@dsp-HP-Pavilion-Notebook:/media/dsp/Programs/Arpon/On the Executability of R Markdown Files/AutomaticallyExecuteR_MarkdownFiles$ python3 A
utomaticallyExecuteR_MarkdownFiles.py
1
R version
platform       x86_64-pc-linux-gnu
arch           x86_64
os             linux-gnu
system         x86_64, linux-gnu
status
major          4
minor          1.2
year           2021
month          11
day            01
svn rev        81115
language       R
version.string R version 4.1.2 (2021-11-01)
nickname       Bird Hippie

  0%|                                                                                              | 0/1 [00:00<?, ?it/s]
Github_Project/cnjelita_datasciencecoursera
```

Run AutomaticallyExecuteR_MarkdownFiles.py to render R Markdown files using python

Figure A.6: An example of executing the script to process R Markdown files.

The example which is given below illustrates the output generated during the rendering of the R Markdown file. In this instance, the rendering process was terminated due to the inability to specify the working directory. This occurred because the selected path in the code is only accessible on the file developer’s computer, not on our machine.



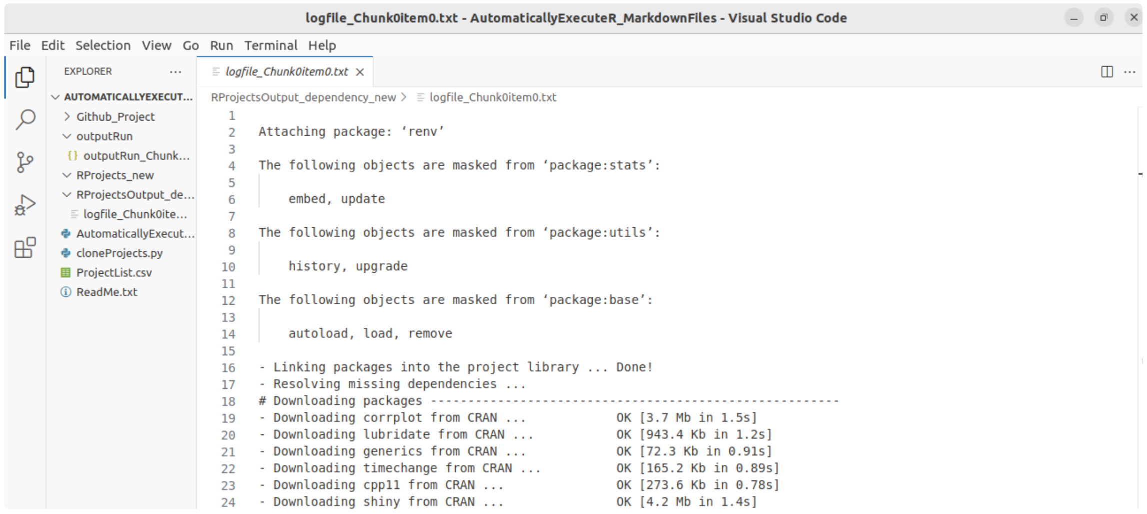
```
................................................................................3
/media/dsp/Programs/Arpon/On the Executability of R Markdown Files/AutomaticallyExecuteR_MarkdownFiles/RProjects_new/cnjelita_datasciencecours
era/PA1_PeerAssesment1.Rmd
1/9
2/9 [unnamed-chunk-1]



processing file: PA1_PeerAssesment1.Rmd

Quitting from lines 8-14 [unnamed-chunk-1] (PA1_PeerAssesment1.Rmd)
Error in `setwd()`:
! cannot change working directory
Backtrace:
 1. base::setwd("C:/Users/Charles/Desktop/Coursera/Reproduceable Research/repdata-data-activity")
Execution halted
Command exited with non-zero status 1
0.61user 0.08system 0:00.70elapsed 99%CPU (0avgtext+0avgdata 87696maxresident)k
0inputs+16outputs (0major+20724minor)pagefaults 0swaps
```

Figure A.7: An example of rendering output while automatically executing an R Markdown file.

The output of the dependency installation is kept in the "RProjectsOutput_dependency_new" directory and is arranged in the following file structure. It's crucial to emphasize that our method processes projects in chunks, with the output files labeled with the project number and the chunk number for easy identification.

Table A.2: File list generated from automatic execution

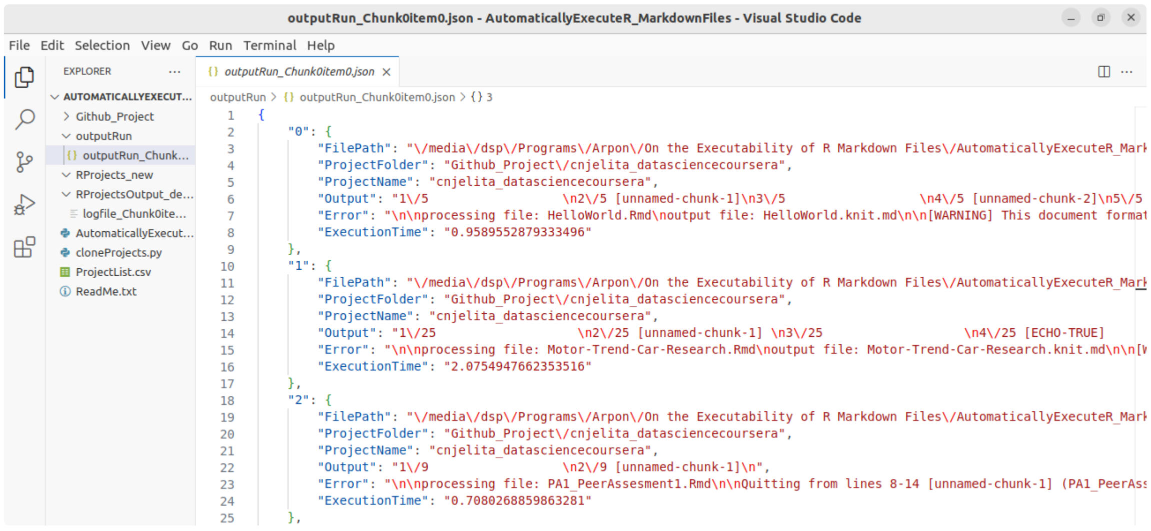
Name	**File name with extension**	**Description**
Dependency output	"logfile_Chunk"+ chunkNumber+ "item"+ projectNumber +".txt"	This specified filename is where our script will save the unique dependency installation results for every project.
Render output	"outputRun_Chunk"+ chunkNumber+ "item"+ projectNumber +".json"	This specified filename is where our script will save the render results of R Markdown files for every project.



```

Attaching package: ‘renv’

The following objects are masked from ‘package:stats’:

    embed, update

The following objects are masked from ‘package:utils’:

    history, upgrade

The following objects are masked from ‘package:base’:

    autoload, load, remove

- Linking packages into the project library ... Done!
- Resolving missing dependencies ...
# Downloading packages -------------------------------------------------------
- Downloading corrplot from CRAN ...            OK [3.7 Mb in 1.5s]
- Downloading lubridate from CRAN ...           OK [943.4 Kb in 1.2s]
- Downloading generics from CRAN ...            OK [72.3 Kb in 0.91s]
- Downloading timechange from CRAN ...          OK [165.2 Kb in 0.89s]
- Downloading cpp11 from CRAN ...               OK [273.6 Kb in 0.78s]
- Downloading shiny from CRAN ...               OK [4.2 Mb in 1.4s]
```

Figure A.8: An example of installing dependencies.

When the program processes each R Markdown file, it will collect data like “FilePath”, “ProjectFolder”, “ProjectName”, “Output”, “Error”, and “ExecutionTime”. After that, this information will be kept in the **outputRun** folder. The following file structure is offered as an illustration.


outputRun_Chunk0item0.json - AutomaticallyExecuteR_MarkdownFiles - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
AUTOMATICALLYEXECUT...
Github_Project
outputRun
outputRun_Chunk...
RProjects_new
RProjectsOutput_de...
logfile_Chunk0ite...
AutomaticallyExecut...
cloneProjects.py
ProjectList.csv
ReadMe.txt
outputRun_Chunk0item0.json
outputRun > outputRun_Chunk0item0.json > {} 3

```
{
    "0": {
        "FilePath": "\/media\/dsp\/Programs\/Arpon\/On the Executability of R Markdown Files\/AutomaticallyExecuteR_Mark
        "ProjectFolder": "Github_Project\/cnjelita_datasciencecoursera",
        "ProjectName": "cnjelita_datasciencecoursera",
        "Output": "1\/5                    \n2\/5 [unnamed-chunk-1]\n3\/5                    \n4\/5 [unnamed-chunk-2]\n5\/5
        "Error": "\n\nprocessing file: HelloWorld.Rmd\noutput file: HelloWorld.knit.md\n\n[WARNING] This document format
        "ExecutionTime": "0.9589552879333496"
    },
    "1": {
        "FilePath": "\/media\/dsp\/Programs\/Arpon\/On the Executability of R Markdown Files\/AutomaticallyExecuteR_Mark
        "ProjectFolder": "Github_Project\/cnjelita_datasciencecoursera",
        "ProjectName": "cnjelita_datasciencecoursera",
        "Output": "1\/25                    \n2\/25 [unnamed-chunk-1] \n3\/25                    \n4\/25 [ECHO-TRUE]
        "Error": "\n\nprocessing file: Motor-Trend-Car-Research.Rmd\noutput file: Motor-Trend-Car-Research.knit.md\n\n[W
        "ExecutionTime": "2.0754947662353516"
    },
    "2": {
        "FilePath": "\/media\/dsp\/Programs\/Arpon\/On the Executability of R Markdown Files\/AutomaticallyExecuteR_Mark
        "ProjectFolder": "Github_Project\/cnjelita_datasciencecoursera",
        "ProjectName": "cnjelita_datasciencecoursera",
        "Output": "1\/9                    \n2\/9 [unnamed-chunk-1]\n",
        "Error": "\n\nprocessing file: PA1_PeerAssesment1.Rmd\n\nQuitting from lines 8-14 [unnamed-chunk-1] (PA1_PeerAss
        "ExecutionTime": "0.7080268859863281"
    },
```

Figure A.9: An example of output obtained by executing an R Markdown file.

Then, we merge all the files and check for errors. Any output that contains the words “Error in” implies that something went wrong during execution. With the use of this data, we generate a dataset that divides R Markdown files into executable and non-executable categories. When “error in” is absent in the output of the files, it means that the files have been successfully re-executed.

Execution Overview:

- Obtain every project listing located in the root directory.
- Divide the list into digestible sections for faster processing.
- Carry out tasks on every project segment one after the other.
- The first step is to copy the project into a new directory called RProjects_new.
- Locate and gather all of the R Markdown files in each project, keeping out package management folders (packrat, renv, and .checkpoint, for example).
- Check to see if the project has a package management system.
- Restore dependencies appropriately if a package management system is found and version information is provided.
- If you use a package management system but version information is missing, get the most recent version.
- Use renv to install the dependencies if no package management is being used.
- Process-wise, render every R Markdown file.
- The execution will end if processing the file takes more than ten minutes.
- Archive all R Markdown files automatically, including errors and output.
- If you have any further projects to process, delete the processed project and move on to step 5.

After that, we combine the files and carefully review them to look for any errors. Any output that contains the words "Error in" indicates that something went wrong during execution. We use this data to construct a dataset that divides R Markdown files into executable and non-executable categories. If there is no "Error in" message in a file's output, re-execution was successful.

Bibliography

[1] D. E. Knuth, "Literate programming," in *Proceedings of the The Computer Journal*, 1984, pp. 97–111.

[2] "Knitr," 2023. [Online]. Available: https://cran.r-project.org/web/packages/knitr/index.html

[3] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, March 2008, pp. 337–340.

[4] "Cran," 2023. [Online]. Available: https://cran.r-project.org/web/packages/available_packages_by_name.html

[5] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: an analysis of stack overflow code snippets," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 391–402.

[6] E. Horton and C. Parnin, "Gistable: Evaluating the executability of python code snippets on github," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 217–227.

[7] "Renv," 2023. [Online]. Available: https://cran.r-project.org/web/packages/renv/index.html

[8] S. Mirhosseini and C. Parnin, "Docable: Evaluating the executability of software tutorials," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 375–385.

[9] M. M. Hossain, N. Mahmoudi, C. Lin, H. Khazaei, and A. Hindle, "Executability of python snippets in stack overflow," *arXiv preprint arXiv:1907.04908*, 2019.

[10] E. Horton and C. Parnin, “Dockerizeme: Automatic inference of environment dependencies for python code snippets,” in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 328–338.

[11] J. Pimentel, L. Murta, V. Braganholo, and J. Freire, “A large-scale study about quality and reproducibility of jupyter notebooks,” in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 507–517.

[12] R. Mayer and A. Rauber, “A quantitative study on the re-executability of publicly shared scientific workflows,” in *Proceedings of the IEEE 11th International Conference on e-Science*, 2015, pp. 312–321.

[13] C. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li, and D. De Roure, “myexperiment: a repository and social network for the sharing of bioinformatics workflows,” *Nucleic Acids Research*, pp. W677–W682, 2010.

[14] S. Mondal, M. M. Rahman, and C. K. Roy, “Can issues reported at Stack Overflow questions be reproduced? an exploratory study,” in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 479–489.

[15] A. large-scale study on research code quality and execution, “Trisovic, aleksandar and lau, michael k. and pasquier, thomas and crosas, mercè,” in *Proceedings of the Scientific Data*, 2022, p. 60.

[16] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, “A model to understand the building and running inter-dependencies of software,” in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 140–149.

[17] E. Horton and C. Parnin, “V2: Fast detection of configuration drift in python,” in *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 477–488.

[18] C. Wang, R. Wu, H. Song, J. Shu, and G. Li, “smartpip: A smart approach to resolving python dependency conflict issues,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[19] M. Lungu, R. Robbes, and M. Lanza, “Recovering inter-project dependencies in software ecosystems,” in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 309–312.

[20] “Packrat,” 2023. [Online]. Available: https://rstudio.github.io/packrat/

[21] “Devtools,” 2023. [Online]. Available: https://devtools.r-lib.org/

[22] "Checkpoint," 2023. [Online]. Available: https://cran.r-project.org/web/packages/checkpoint

[23] "Packman," 2023. [Online]. Available: https://cran.r-project.org/web/packages/pacman/index.html

[24] J. Wang, T. Kuo, L. Li, and A. Zeller, "Assessing and restoring reproducibility of jupyter notebooks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 138–149.

[25] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of jupyter notebooks," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1622–1633.

[26] C. Zhu, R. Saha, M. Prasad, and S. Khurshid, "Restoring the executability of jupyter notebooks by automatic upgrade of deprecated apis," in *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 240–252.

[27] "Automagic," 2023. [Online]. Available: https://github.com/cole-brokamp/automagic

[28] "Jetpack," 2023. [Online]. Available: https://github.com/ankane/jetpack

[29] "Github rest api documentation," 2023. [Online]. Available: https://docs.github.com/en/rest?apiVersion=2022-11-28

[30] "Libraries.io," 2023. [Online]. Available: https://libraries.io/data

[31] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan, "Should i contribute to this discussion?" in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 181–190.

[32] "Lintr," 2023. [Online]. Available: https://lintr.r-lib.org/

[33] "Cloc," 2023. [Online]. Available: https://github.com/hrbrmstr/cloc

[34] J. P. Kincaid, R. P. F. Jr, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel," US Naval Air Station, Tech. Rep. Research Branch 8–75, 1975.

[35] R. J. Senter and E. A. Smith, "Automated readability index," 1967.

[36] R. Flesch, "A new readability yardstick," in *Journal of Applied Psychology*, 1948, p. 221.

[37] R. Gunning, *The Technique of Clear Writing.* New York: McGraw-Hill, 1952.

[38] J. Anderson, "Lix and rix: Variations on a little-known readability index," in *Journal of Reading*, 1983, pp. 490–496.

[39] G. H. McLaughlin, "Smog grading—a new readability formula," in *Proceedings of the J. Reading*, 1969, pp. 639–646.

[40] "Readability," 2023. [Online]. Available: https://pypi.org/project/readability/

[41] "Scikit-learn," 2023. [Online]. Available: https://scikit-learn.org/

[42] H. Mann and D. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," in *Annals of Mathematical Statistics*, 1947, pp. 50–60.

[43] K. Kelley and K. J. Preacher, "On effect size," in *Psychological Methods*, 2012, p. 137.

[44] "pandas.dataframe," 2023. [Online]. Available: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html

[45] J. Cohen, "A coefficient of agreement for nominal scales," in *Proceedings of the Educational and Psychological Measurement*, 1960, p. 37.

[46] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification.* Hoboken: Wiley, 2000.

[47] D. W. H. Jr., S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression.* John Wiley & Sons, 2013, vol. 398.

[48] J. R. Quinlan, "Induction of decision trees," in *Machine Learning*, 1986, pp. 81–106.

[49] L. Breiman, "Random forests," in *Proceedings of the Machine Learning*, 2001, pp. 5–32.

[50] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[51] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," in *Journal of Artificial Intelligence Research*, 2002, pp. 321–357.

[52] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva, "Iterative and core-guided maxsat solving: A survey and assessment," 2013, pp. 478–534.

[53] "antlr," 2023. [Online]. Available: https://www.antlr.org/download.html

[54] “pypandoc,” 2023. [Online]. Available: https://pypi.org/project/pypandoc/

[55] “Github,” 2023. [Online]. Available: https://github.com/

[56] “Bioconductor,” 2023. [Online]. Available: https://www.bioconductor.org/