Code Duplication and Reuse in Jupyter Notebooks

by

Andreas Peter Koenzen
B.Sc., Catholic University of Asunción, 2017

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Code Duplication and Reuse in Jupyter Notebooks

by

Andreas Peter Koenzen
B.Sc., Catholic University of Asunción, 2017

Supervisory Committee

Dr. Neil A. Ernst, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne D. Storey, Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Neil A. Ernst, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne D. Storey, Supervisor
(Department of Computer Science)

ABSTRACT

Reusing code can expedite software creation, analysis and exploration of data. Expediency can be particularly valuable for users of computational notebooks, where duplication allows them to quickly test hypotheses and iterate over data, without creating code from scratch. In this thesis, I'll explore the topic of code duplication and the behaviour of code reuse for Jupyter notebooks; quantifying and describing snippets of code and explore potential barriers for reuse. As part of this thesis I conducted two studies into Jupyter notebooks use. In my first study, I mined GitHub repositories, quantifying and describing code duplicates contained within repositories that contained at least one Jupyter notebook. For my second study, I conducted an observational user study using a contextual inquiry, where my participants solved specific tasks using notebooks, while I observed and took notes. The work in this thesis can be categorized as exploratory, since both my studies were aimed at generating hypotheses for which further studies can build upon. My contributions with this thesis is two-fold: a thorough description of code duplicates contained within GitHub repositories and an exploration of the behaviour behind code reuse in Jupyter notebooks. It is my desire that others can build upon this work to provide new tools, addressing some of the issues outlined in this thesis.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My supervisors **Dr. Neil A. Ernst** and **Dr. Margaret-Anne D. Storey** for wisely and patiently teaching me analytical thinking and for guiding me into becoming a researcher. It was not an easy path, but I was very fortunate to have such great supervisors to guide me through.

My fellow colleagues at the **CHISEL Lab**, for their help, the good laughs and interesting conversations during lunch time. A special thanks to my good friend **Omar Elazhary** for brainstorming with me possible paths for my research.

The **participants** of my study, for dedicating an hour of their time to my questions and prying. I am most grateful for their contribution to my work.

The **staff of the Computer Science department** at the University of Victoria for their support and help. They made the bureaucratic journey much smoother.

The **University of Victoria**, my *alma mater*, for the imparted knowledge.

My family.

Whatever you can do, or dream you can, begin it. Boldness has genius, power and magic in it.

—W.H. Murray, *The Scottish Himalayan Expedition*

DEDICATION

To my daughter Emma and my wife Alicia. To my mother Ana María and my father Jürgen Peter ($\star$ 1943, † 1985).

Chapter 1

Introduction

Computational notebooks have become the preferred tool for users exploring and analyzing data. Their power, versatility and ease of use have made this new medium of computation the de facto standard for data exploration [1]. During intensive data exploration sessions, users tend to generate great numbers of artifacts (e.g., graphs, scripts, notebooks, database files, etc.) [2]. By reusing these artifacts — in the form of Jupyter code cells — users can expedite experimentation and test hypotheses faster [3, 4]. Despite the fact that software engineering best practices include avoiding code duplication whenever possible [5, 6], it is common behaviour with Jupyter notebooks as it is especially easy to duplicate cells, make minor modifications, and then execute them [7, 8].

Through appropriate tools, this form of code reuse expedites data exploration, but creates notebooks that are hard to read, maintain and debug. The recommended way to reuse code is to create modules, which are standalone code files (e.g., Python or R scripts) that can be imported locally into a notebook [8]. Unfortunately, it is reported that only about 10% of notebooks contain such local imports (those imported from the repository directory) [9]. Hence, there is a great amount of code in notebooks for which there is no provenance, and understanding where code in notebooks originates and how it is reused is important if we want to create new tools for this environment.

Previous work in the area of computational notebooks describes developers' motivations for reuse and duplication but does not show how much reuse occurs or which

barriers users face reusing code. To address this gap, I first analyzed GitHub repositories for code duplicates contained in Jupyter notebooks, and then conducted an observational user study where participants solved specific tasks using notebooks. In my first study, I focused explicitly on code duplicates.

My definition of code duplicates is that of Roy and Cordy: "*snippets of code copied and pasted with or without modifications, intentionally reused in order save time and effort*" [5], although there is still some debate as to what exactly a clone is [10].

Given the often transient nature of notebooks, combined with the fast-paced nature of data exploration, I hypothesized that code duplication happens often in Jupyter notebooks and that it might even be useful for reducing time between ideas and results while exploring data. While understanding the usefulness of duplicates is beyond the scope of this thesis and may well be a worthy subject of research in future studies, I did manage to show with my studies that this activity does happen with considerable frequency in Jupyter notebooks.

We know from software engineering research that "*Cloning can be a good strategy if you have the right tools in place. Let programmers copy and adjust, and then let tools factor out the differences with appropriate mechanisms.*" [10], I argue that code duplication can be beneficial for Jupyter notebooks with the support of the "right tools".

Code duplicates — also known as code clones — have been studied extensively in software engineering, and research shows that a significant number of software systems contain code clones[1] [5, 11]. No such study exists for computational notebooks.

I differentiate between *code duplication* (artifact) and *code reuse* (behaviour). I analyzed code duplication inside repositories and not across them. Hence, in this thesis I use the term *code duplicate* to signal code that is contained and replicated in a single project. Although notebooks support cells of multiple types (including code and markdown text), I focused my study on code cells.

[1] In this thesis I will use the terms *clone* and *duplicate* interchangeably.

1.1 Research Questions

The overarching goal of this thesis is to discuss and describe the topic of code reuse within the realm of Jupyter notebooks from two different perspectives: a quantitative one and a qualitative one. For that purpose, I began my studies with three main exploratory research questions.

RQ1: How much cell code duplication occurs in Jupyter notebooks? And what is the main programming goal of these duplicates? To answer this first question, I opted for a quantitative study, where I mined GitHub repositories containing at least one Jupyter notebook. The goal of this study was to quantify code duplicates and near-duplicates (this concept will be explained later on). I scoped my search of duplicates to a randomly sampled data set of 1,000 GitHub repositories containing at least one Jupyter notebook. Proceeding the detection of duplicates, I categorized these duplicates using the inductive coding technique, by which I was able to assign a main programming goal to each snippet.

RQ2: How does cell code reuse happen in Jupyter notebooks? The goal of this research question was to understand the preferred method of reuse in Jupyter notebooks, e.g., copy and paste, copy by typing, duplicating other notebooks, etc. Understanding how code gets inside Jupyter notebooks is very important.

RQ3: What are the preferred sources for code reuse in Jupyter notebooks? I believe that answering this research question correctly is paramount for the development of new tools that augment development using notebooks. Knowing from where a particular snippet of code came from is essential. If we manage to understand snippet's sources, then we could build better plugins or extensions to speed up reuse.

To answer the last two research questions I used and designed an observational lab study ($n = 8$), where I observed participants while they solved a particular set of tasks, recording audio/video feed and taking detailed notes of their behaviour. This

study was complemented with an opening questionnaire and a closing short interview.

1.2 Contributions

The contribution of this thesis is two-fold: first I managed to quantify and analyze code duplication in Jupyter notebooks within an acceptable recall, and second, I managed to observe reuse behaviour in Jupyter notebooks.

RQ1: How much cell code duplication occurs in Jupyter notebooks? And what is the main programming goal of these duplicates? My first study shows that, approximately one in thirteen code cells in Jupyter notebooks are duplicates, and that the main programming goal of these duplicated snippets varies between 4 main categories: visualization (21%), machine learning (15%), the definition of functions (12%) and data science (9%).

RQ2: How does cell code reuse happen in Jupyter notebooks? Reuse in Jupyter notebooks happens through various methods, users reused programming code by copying and pasting it or by typing it from memory. The most common method of reuse is copying and pasting, followed by copy by typing, and the least used method is duplicating a notebook.

RQ3: What are the preferred sources for code reuse in Jupyter notebooks? The preferred source of code reuse is browsing online for examples. The sites that were visited the most are: tutorial sites (35%), API documentation (32%) and Stack Overflow (14%). There was some reuse as well coming from other notebooks previously completed. The source with the least reuse is version control systems. Some participants hinted, that there is a correlation between the complexity of the code being reused and the source from where it is being reused. Simpler tasks can be reused easily from web sites, but more complex routines, especially long and advanced functions, which belong to one's own codebase, could merit reuse from other sources as well, like other notebooks and version control systems.

1.3 Structure

This thesis is structured as follows:

Chapter 2: In chapter 2 I briefly introduce the reader to the concept of computational notebooks and EDA (Exploratory Data Analysis), to conclude the chapter with an in-depth discussion of the state of the art in *code duplication and reuse* for Jupyter notebooks.

Chapter 3: In chapter 3 I explain the details of my first study (GitHub Mining), the results I obtained, the limitations of this study, and a brief conclusion.

Chapter 4: In chapter 4 I explain the details of my second study (Observational Study), the results I obtained, the limitations of this study, and a brief conclusion.

Chapter 5: In chapter 5 I discuss the results and general limitations of both studies, followed by a discussion about the impact these results have on practice and future research.

Chapter 6: In chapter 6 I conclude this thesis outlining the work done, a brief overview of my results, and a conclusion of my work.

In the appendix I include additional documents, charts and ancillary data regarding both studies. These ancillary documents are not part of the reproducibility package. This thesis' reproducibility package can be found at https://doi.org/10.5281/zenodo.3836691.

Chapter 2

Background & Related Work

Computational notebooks are a relatively new interactive computational paradigm that allows users to interleave code and text via a web interface. Programming code is introduced and segmented into *code cells* that are executed in a *kernel* (Python, R, Julia, C++, other) with computation output/results returned to the web interface for display. This new way of computation makes sharing and coding easy for programming newcomers, as users do not need to compile code or deal with low-level configurations. Several services currently offer computational notebooks: Google Colab [12] & Cloud AI Platform [13], Azure Notebooks [14], Databricks [15], nteract [16], Apache Zeppelin [17], to name a few. These services provide even more abstraction by taking take care of *kernel* configurations and just providing one for the user to select and use.

In this section I will try to synthesize the literature regarding the intersection of code reuse and computational notebooks. It is worth mentioning that not much research has been done in this specific area of research. There have been a few studies on Jupyter notebooks where code reuse was mentioned, but none has been dedicated exclusively to this topic. In the next sections I will talk briefly about what computational notebooks are, using them for data exploration, how developers search for information when coding, and code duplication and reuse.

2.1 Computational Notebooks

Computational notebooks or a notebook interface is a virtual notebook for literate programming [18]. The first computational notebook was Wolfram Mathematica 1.0 dating back as far as 1988. Notebooks extended work done by Iverson [19], where a user using a simple interface could introduce mathematical and logical expression which were computed by an interpreter and the output returned to user. All this was done interactively, allowing the user to try out different expressions easily. This is known today as REPL (read-eval-print loop). In 2007, Fernando Perez and Brian Granger released IPython [20], which was a Python REPL system for scientists, with support not only for complex code expressions, but also to display rich text and images. That project evolved into Project Jupyter, altering the interface to a web-based one, thus introducing support for more complex interactions and display.

Notebooks are designed to offer an easy to use and comfortable interface into the workflow of scientific computing, from interactive exploration to publishing a detailed record of computation. Notebooks are organized into cells, chunks of code and markdown which can be individually modified and run. Output from cells appears directly below it and it is stored as part of the document itself. Direct output in most interactive shells can only be text, notebooks can include rich output such as plots, animations, formatted mathematical equations, audio, video and even interactive controls and graphics. Prose text can be interleaved with the code and output in a notebook to explain and highlight specific parts, forming a rich computational narrative [21].

2.1.1 Uses of Computational Notebooks

Computational notebooks have become the preferred tool for users exploring and analyzing data. Their power, versatility and ease of use have made this new medium of computation the de facto standard for data exploration [1]. During intensive data exploration sessions, users tend to generate great numbers of artifacts [2]. By reusing these artifacts — in the form of Jupyter code cells — users can expedite experimentation and test hypotheses faster [3, 4]. Despite the fact that software engineering best

Figure 2.1: Example of a Jupyter notebook rendered using the latest version of JupyterLab.

practices include avoiding code duplication whenever possible [5, 6], it is common behaviour with Jupyter notebooks as it is especially easy to duplicate cells, make minor modifications, and then execute them [7, 8].

This form of code reuse expedites data exploration, but creates notebooks that are hard to read, maintain and debug. The recommended way to reuse code is to create modules, which are standalone code files (e.g., Python or R scripts) that can be imported locally into a notebook [8]. Unfortunately, it is reported that only about 10% of notebooks contain such local imports (those imported from the repository directory) [9].

During a data exploration phase, an analyst looks for patterns in data by trying

out different alternatives [22,23] until a satisfactory result is found or new hypotheses arise, which in turn gives way to new exploration phases. This process is necessary to achieve satisfactory results, since there is no single path known beforehand that will lead them to relevant insights, but rather each unfruitful path may well provide the basis for new ones. This acts in contrast to "professional programming", where a programmer is ruled by a set of requirements which were established beforehand and by which he must abide.

2.1.2 Types of Users and Programming Paradigms

There have been studies outlining code and artifacts' reuse behaviour before, like Brandt *et al.* in [4], where they studied what they called *opportunistic programming*, which is the paradigm where programmers work opportunistically, emphasizing speed and ease of development over code robustness and maintainability. This paradigm is particularly useful for designing, prototyping and understanding very rapidly in the development process what the right solution is.

Other studies, like the one by Sandberg *et al.* [24], have proposed terms like *exploratory programming* to refer to "programmers exploring and trying out multiple alternatives". This term and definition was coined for occasions where software developers tried variations in their own code and ran those variations to see if the outcome improved. This exploratory behaviour also aligns with the role of data analysts, trying different approaches on data before they can discover meaningful patterns [25].

The term *research programmers* was also defined by Guo [26] in his Ph.D. thesis to refer to developers writing code only to extract insights from data. Another relevant term is *end-user developer* which was coined by Ko *et al.* [27] and it is defined as: "programming to achieve the result of a program primarily for personal, rather [than] public use". Data analysts can be classified as end-user developers, given that they use and extend programming code solely to analyze data.

The activity of exploring data is tightly correlated to reusing previous artifacts, due to the fact that most code developed using notebooks is not meant for production, but rather to extract insights as fast as possible [23], hence analysts pay little to no

attention to software engineering practices like *maintainability* [4].

2.2 Code Duplication and Reuse in Computational Notebooks

Code cloning or duplication is considered to be a bad practice or bad smell in software engineering as described by Fowler [6], as it is believed to cause maintainability issues [5, 28]. However, other studies that analyzed the impact and damage of code clones have provided evidence that the problem might be less severe than what was originally estimated [29]. It is always preferable, and in fact it is highly recommended as good practice, to create modules with functions that can be accessed through interface implementations. However, resorting to duplicates can sometimes simplify the development effort, especially if the goal of the code is to be used as playground or testing, as is the case with Jupyter notebooks [7].

Previous studies in computational notebooks have analyzed how people use them, and reports shows that, when it comes to modularity only about 10% of Jupyter notebooks contain imports from local libraries [9]. This flexibility in the design of Jupyter notebooks might be due to the fact that their users are not concerned with coding best practices [30] but with ease of use. Or due to the fact that users of Jupyter notebooks prioritize finding a solution over writing high quality code, as reported in a study by Kery *et al.* [23].

Although coding best practices are not paramount for users of Jupyter notebooks, there are projects that try to shift that attitude into one more oriented towards reusability and modularity. One of these projects is *Papermill* [31], an *nteract* [16] library for passing parameters to Jupyter notebooks. It lets users reuse a notebook by passing specific parameters at run-time, allowing one to try multiple approaches without needing to create extra cells. This form of reuse is particular necessary for computational notebooks since they allow users to execute notebooks from the command line just like a regular script, and to collect computation results using different mediums (local files, S3, and others).

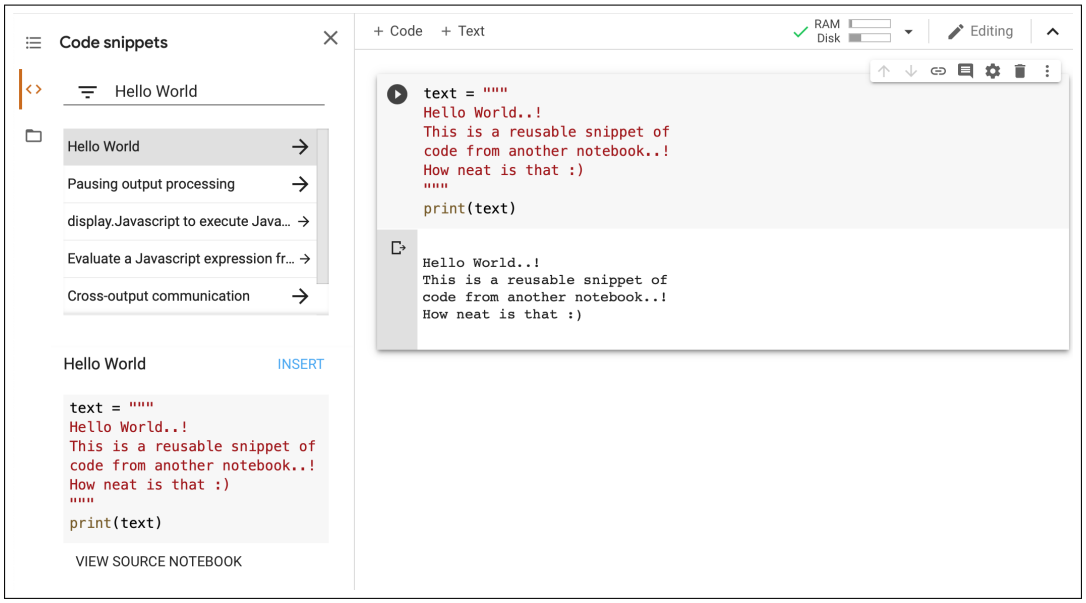


Figure 2.2: Google Colab function to search and reuse snippets of code from other notebooks. Snippets can be reused with one click of the mouse.

Another form of reuse that is widely used is the practice of adding snippets of code to notebooks with the click of a mouse. This form of reuse entails a local library of snippets, from which the user could read and write snippets. Google Colab [12] offers a function for users to specify a notebook where reusable snippets of code reside and from where users can reuse with a simple click (See Figure 2.2). This form of quick duplication and reuse has been defined by users of notebooks on Stack Overflow and other internet forums as a "*super needed feature*" and as a "*useful way to insert small, reusable code chunks into a notebook with a single click*".

Other forms of reuse have been studied before. Kery and Myers [3] reported that developers relied extensively on copying versions of their files to support their data exploration sessions. Others have suggested new tools that expedite exploration by enabling better access to previous artifacts and exploration history [23, 32]. These tools have focused on internal *in-notebook* code duplication and reuse, using past cells and a notebook's history as a source of reuse.

Head *et al.* [33] examined how data scientists manage notebook artifacts at Microsoft, and proposed a tool for cleaning the notebook history by interleaving cells and pruning unnecessary code, leaving only the code necessary to recreate the desired

output. This solution provided a way for developers to clean their notebooks before reusing and sharing them with others.

Chattopadhyay *et al.* [34] surveyed Microsoft data scientists about notebook pain points. One of the reported pain points is the difficulty of exploring and analysing code, which results in continual copy and paste cycles. Their participants also ranked activities based on importance, and *Reuse Existing* code was labeled as at least important 94% of the time.

It is also worth mentioning that reuse is not limited to any specific source. It can come from either web pages, other notebooks, or from version control system (VCS) repositories (e.g., *git*, SVN and others). Other studies have investigated version control systems supporting analysts' exploration of data, like studies conducted by Kery *et al.*, where participants reported not relying on VCS for their exploration sessions despite using them often for other tasks [35].

As it is with VCSes, reusing code from other Jupyter notebooks presents some issues as well. Studies have reported difficulties choosing easily identifiable names for files and folders [23], which generate confusion when trying to find relevant snippets of code. Imagine a data analyst creating a different notebook for each analysis path they decide to take, e.g., they may well end up with many different notebooks named *hypothesis_1.ipynb*, *hypothesis_2.ipynb* and so on [22, 34]. This type of versioning presents many problems when it comes to finding useful snippets of code, including how to distinguish one exploration path from the other, and how to quickly know which one contains the snippet we are looking for. One way to solve these problems would be to provide for longer names that could describe more in depth what a notebook contains or is about, but that introduces new problems in itself, namely, longer and more convoluted names. Another solution would be to allow users to traverse previous notebooks more easily, maybe by indexing them and offering a search interface, akin to Google Cloud Source Repositories' code search function [36].

2.3 Chapter Summary

In this chapter I went over previous literature regarding computational notebooks and code reuse. I have also explained what data exploration using notebooks looks like, different programming paradigms for data exploration and the relevance of duplication and reuse. I must admit that the literature surrounding this area of research is limited, which I consider to be a magnificent opportunity to enhance the underlying knowledge of this new medium of computation, which has proven to be tremendously popular among users outside of computer science. In the next chapters I will further explore this topic by outlining two studies I conducted in order to better understand the necessities of these users: one quantitative, quantifying and describing clones in Jupyter notebooks, and the other through a qualitative lens, understanding information seeking behaviour and methods of reuse, by using an observational study.

Chapter 3

Quantifying and Describing Jupyter Code Cell Duplicates on GitHub

Exploratory data analysis is detective work.

—John Tukey, *Exploratory Data Analysis*

Herzig and Zeller describe *mining software archives* as a process to "obtain lots of initial evidence" by extracting data from software repositories [37].

In order to better understand code duplication in computational notebooks, I decided to mine GitHub repositories using the data set created by Rule *et al.* in [22]. I decided to use this data set for the reason that the methods used for its creation were scientifically sounded and proved effective by Rule *et al.* in [22]. As for the repository mining approach, it is a highly regarded method of understanding programmer's behaviour, and has been used effectively in other clone detection studies [38]. Rule's study retrieved 1.25 million notebooks from GitHub, which they estimated as 95% of the notebooks available in 2017. I used a random sample of 1,000 repositories provided with this data set. This random sample of 1,000 repositories contained a total of 6,515 Jupyter notebooks. Jupyter notebooks are just self-contained JSON files segmented into cells, along with base64 encoded output of these cells and associated metadata. It is important to outline the property of notebooks of being *self-contained*, because

it means that all data necessary to reproduce a particular notebook is contained within the JSON file, including all output of cells. This property permits notebooks to grow to a significant size. For example, notebooks with videos or animations can easily span several megabytes in size. Cells within notebooks can be of various types: markdown cells are cells that contain documentation or text in Markdown format[1], source code cells contain programming code in any number of different languages[2], output data (e.g., images, audio files, videos, animations, etc., which are encoded as base64 data [39]), and raw data. This study focuses on source code cells — the ones with snippets of programming code. For the remainder of this document I will refer to *code cell* as *cell*.

The goal of this first study was to answer **RQ1**: **How much cell code duplication occurs in Jupyter notebooks? And what is the main programming goal of these duplicates?** using a quantitative method of analysis (software repository mining) and a qualitative lens (inductive coding).

3.1 Code Duplicates

> **Code Clone** Snippets of code copied and pasted with or without modifications, intentionally reused in order save time and effort [5].

According to Roy and Cordy [5] clones can be introduced in a software system by:

a) by copy and paste,

b) by forking, and

c) by design, functionality and logic reuse.

and they categorize clones into four types:

Type-1: An exact copy of a code snippet except for white spaces and comments in the source code.

[1] https://daringfireball.net/projects/markdown/

[2] https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

Type-2: A syntactically identical copy where only user-defined identifiers such as variable names are changed.

Type-3: A modified copy of a Type-2 clone where statements are added, removed or modified. Also, a Type-3 clone can be viewed as a Type-2 clone with gaps in-between.

Type-4: Two or more code snippets that perform the same computation but are implemented through different syntactic variants.

In this study, I focus on the first three types of duplicates. Detecting Type-4 duplicates is complex and I believe the first three types are sufficient to answer **RQ1**. In this document I sometimes use the word near-duplicate to refer to Type-2 and Type-3 clones, and sometimes the word duplicate refers to Type-1 clones, but I make this distinction explicit. Code clone and code duplicate will be used interchangeably throughout this document.

3.2 Analyzed Jupyter Notebooks Data Set

The data set I used for this study is the **Sample notebook data** provided and used by Rule *et al.* in [22] and which can be found at this link[3]. It is composed of 1,000 randomly sampled repositories, which contained exactly 6,515 Jupyter notebooks. Before jumping into the analysis of code duplicates within this particular data set, I ensured that this data set was representative of *personal* behaviour and not *collective* behaviour. Since notebooks can be shared and edited by many developers as part of the same repository or project. I considered that multi-developer code duplication was possible, like in the following example: imagine a notebook edited by two developers, where developer A could create a notebook, fill it with code cells that perform some task, and then developer B within the same project could easily reuse what developer A did on another notebook. Analyzing collective duplication and reuse was not part of this thesis and henceforth when referring to duplication and reuse, it will imply

[3] https://library.ucsd.edu/dc/object/bb2733859v

personal behaviour. As outlined by Kalliamvakou *et al.* in [40], there are perils to mining GitHub for information. *Peril V* of their study states: "Two thirds of projects (71.6% of repositories) are personal." This peril comes as an advantage for my study, since personal repositories are the ones I'm interested in. In fact, my own analysis shows that at least 75% (Third Quartile = Q_3) of the 6,515 notebooks in this data set were edited by one committer. This fact provides sufficient evidence to support the argument that this data set reflects personal behaviour and not a collective one. Also, forked repositories were not considered and were excluded from the data set.

3.3 Code and Function to Detect Duplicates

To compute duplicates (Type-1) and near-duplicates (Type-2 and Type-3) for this study, I implemented my own detection code using Python. For the detection function — the function that actually computes if two snippets are closed enough to be catalog as clones — I created a "Duplicate Ratio Function", which is listed below. My Python code computes duplicates and near-duplicates in a conservative manner, in which possible permutations of snippets are computed only once, according to a triangulation of the SHA256 hash of the cell, the file name of the notebook for which the cell belongs to and the cell number (the position in the notebook). This conservative approach was necessary to avoid counting cells more than once, and it proved to be the best approach. The limitations of this design will be covered extensively in the limitations section of this chapter (See Section 3.8). Using this triangulation of hash, file name and cell number, I was able to almost uniquely identify a code cell inside a repository, with some minor collisions.

Duplicate Ratio Function 1 Function for computing the *duplicate ratio* (DR) between two code cells.

Input: 2 Non-Empty Code Cells (Code Block C_1 & C_2)

Output: $[0, +\infty)$

 # Levenshtein distance between blocks.

1: $ld : int \leftarrow LD(C_1, C_2)$

 # Compute number of characters of both blocks.

2: $lg_1 : int \leftarrow len(C_1)$

3: $lg_2 : int \leftarrow len(C_2)$

 # Compute number of lines of code of both blocks.

4: $lc_1 : int \leftarrow loc(C_1)$

5: $lc_2 : int \leftarrow loc(C_2)$

 # Lambdas assign how much weight we want to give to each feature.

6: $\lambda_1 : int = 6$ {Penalizes short blocks of code.}

7: $\lambda_2 : int = 8$ {Penalizes few lines of code.}

 # $\mathsf{avg}()$ is a vanilla function to compute averages.

8: **return** $ld \div ((\log \mathsf{avg}(lg_1, lg_2))^{\lambda_1} + (\log \mathsf{avg}(lc_1, lc_2))^{\lambda_2})$

Duplicate Ratio Function 1 returns the *duplicate ratio* (DR) between two cells. It returns a real number in the interval $[0, +\infty)$. $LD(C_1, C_2)$ corresponds to the Levenshtein distance [41] between cells C_1 and C_2. $\mathsf{avglen}(C_1, C_2)$ corresponds to the average *number of characters* in cell C_1 and C_2, and $\mathsf{avgloc}(C_1, C_2)$ is the average *number of lines of code* in cells C_1 and C_2. Parameters λ_1 and λ_2 are constants which act as weights. λ_1 weights the number of characters, and λ_2 weights cell lines of code. Setting these parameters allowed me to deemphasize short, quick print statements (few lines of code) or long blocks of text with few lines of code. I experimented with λ settings, heuristically determining the optimal setting to be $\lambda_1 = 6$, $\lambda_2 = 8$, such that lines of code carry more weight than the number of characters (See Section 3.5.2). In Duplicate Ratio Function 3.1 one can observe my clone detection function in a more concise mathematical form.

$$DR(C_1, C_2) = \frac{LD(C_1, C_2)}{(\log \mathsf{avglen}(C_1, C_2))^{\lambda_1} + (\log \mathsf{avgloc}(C_1, C_2))^{\lambda_2}} \tag{3.1}$$

I measured the quality of my detection function in terms of recall. Recall is an absolute metric used in Information Retrieval for assessing how many of all relevant results were retrieved. It is also used in Computer Vision object detection as a metric to assess how many of the ground-truth labels were detected by the network in a detection layer.

$$Recall = \frac{TP}{TP + FN} \tag{3.2}$$

The goal of my detection code was to minimize as much as possible the *false negatives* (FN) and maximize the *true positives* (TP).

Precision is another metric also used in Information Retrieval and Computer Vision. It is used to measure how many of the results retrieved are actually relevant. It is a relative metric and it is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

Another goal of my detection code was to minimize as much as possible the *false positives* (FP), plus focus my analysis on detecting only snippets that were actually clones of another one.

Duplicates with a DR of 0 are identical (Type-1 duplicates), and the bigger the DR value is, the less similar the two blocks are. I only considered code to be duplicates if it had a DR of 0.3 or lower. I came up with that cut-off value by heuristics, experimenting with a smaller random sample, empirically assessing snippets detected as duplicates. I detected duplicates with different thresholds for the cut-off value, e.g., 0.0-0.1, 0.1-0.2, ..., 0.9-1.0, and I was able to verify that at threshold 0.8-0.9, the recall began to decrease drastically (See Appendix A for some examples of detected clones at different thresholds).

I opted for a text-based/string-based method of detecting clones because it has been used effectively in other studies [42]. I also required cross-language support because Jupyter notebooks support multiple programming languages and kernels. The Levenshtein distance is the minimum number of operations (insertions, deletions or substitutions) required for a string to be equal to another one. This method for detecting code duplicates proved to be effective for detecting Type-1, Type-2 and Type-3 duplicates (see below), but with a highly inefficient running time of $O((n*m)!)$, where n and m are the lengths of C_1 and C_2 in characters. I implemented my own function (Duplicate Ratio Function 1) in order to have more control in the detection of snippets. I also removed comments and leading/trailing white space from lines of code.

3.4 Computational Constraints

The size of this data set is 1.46GB, and presented a computational challenge to analyze with the conventional machines I had at my disposal. My analysis yielded that the median (Q_2) of Jupyter code cells in a repository is around 28 non-empty code cells. If I take a conservative number of 25 cells per repository I would have roughly $\binom{25}{2} = 300$ comparisons per repository, with extreme cases in notebooks with more than a 1,000 cells, yielding $\binom{1000}{2} = 499{,}500$ comparisons. Due to this computation constraint I had to tune the parameters of my function (the cut-off value, λ_1 and λ_2) with a much smaller random sample of 100 repositories. The time complexity of this computation was $O((n * m)!)$, as I explained in the previous paragraph. This NP-hard complexity made necessary the use of cloud computing services, like Google Cloud [13]. The detection of code duplicates in the 1,000 repositories of this data set took approximately 12 days to compute, and it was computed entirely on a Google Cloud VM instance with two CPUs, which was paid using free credits I had with this service.

3.5 Detection Parameters

The correct tuning of parameters for Duplicate Ratio Function 3.1 is the most important and difficult to achieve part of this detection study. Incorrect settings can lead to very low values of precision and recall. In the next subsections I will discuss in depth how I tuned these parameters for optimal clone detection.

3.5.1 The Cut-Off Value

Problem

The correct operation of Duplicate Ratio Function 1 depends on the correct tweaking of some parameters, like the cut-off value. The goal of this parameter is to control which snippet of code is going to be labeled as a clone/duplicate, hence the name: cut-off. For example, Duplicate Ratio Function 1 will assign a real value between $[0, +\infty)$ for each two cells (A and B), called the DR (Duplicate Ratio). This real value signifies how similar two code cells are, e.g., if the ratio between A and B is zero, then it means that A is an identical (Type-1) clone of B and vice versa, hence B will be marked as a clone and its ratio of zero will be stored along with it in the database.

The real problem here is to find an optimal value that maximizes the number of true positives, while at the same time minimizes false positives; in other words, the optimal value for maximizing *recall*. This is not a trivial problem, since in my case I had no ground-truth labels or *oracled* data set [43] to which I could tune my function.

Tuning Methodology

In order to find the optimal cut-off value I selected a smaller random sample of 50 repositories, for which I retrieved all code cells. Then, from the total number of cells retrieved from these 50 repositories, I proceeded to select a random sample of 300 code cells. For this random sample of cells, I ran Duplicate Ratio Function 1 and computed which cells were duplicates using different thresholds of the cut-off value, e.g., 0.0-0.1, 0.1-0.2, ..., 0.8-0.9, until 0.9-1.0. So for instance, for 0.0-0.1: Duplicate

Ratio Function 1 detected all duplicates that had a DR in this interval and these detected duplicates were saved in a text file, that I later verified empirically.

```
def predictions_1(data):
    predictions = []
    for _, passenger in data.iterrows():
        if passenger['Sex'] == 'male':
            predictions.append(0)
        else:
            predictions.append(1)
    return pd.Series(predictions)
predictions = predictions_1(data)
```

```
def predictions_0(data):
    predictions = []
    for _, passenger in data.iterrows():

        predictions.append(0)


    return pd.Series(predictions)
predictions = predictions_0(data)
```

Figure 3.1: Example snippet derived with a cut-off value between 0.5 and 0.6. This particular snippet has a *Duplicate Ratio* (DR) value of 0.53.

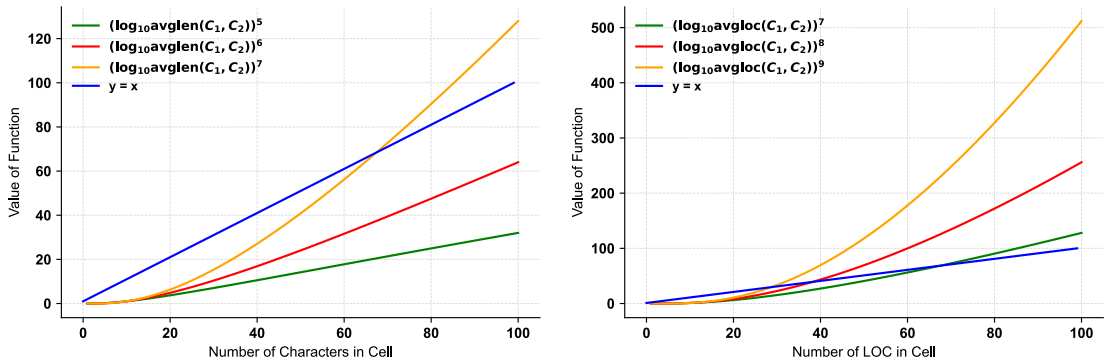
Solution

The result was as expected, the closer the DR value is to zero the similar the snippets are. Duplicate Ratio Function 1 detected snippets accurately up to a value of ≈ 0.8, to which after recall began to drop drastically. The takeaway of this heuristics was to come up with an optimal cut-off value, and 0.3 was the value I decided would yield the higher recall. It is worth noticing as well, that other values would have yielded optimal results as well, like 0.35, 0.40, and probably up to 0.55. This is important to note, since it may lead to an under-reporting of duplicates, especially Type-3 ones. I will cover this issue in the limitations section of this thesis. Refer to Appendix A for more figures depicting snippets detected at different thresholds.

3.5.2 Lambdas

These two parameters (λ_1 and λ_2) control the weights of the number of characters and the lines of code in a given snippet, respectively. As we can observe in Figure 3.2, different values of λ account for different function's growth.

I opted for a value of $\lambda_1 = 6$ (red line in Figure 3.2a) to assign less weight to small



(a) Growth of $\log_{10} \mathsf{avglen}(C_1, C_2)$ using a value of 6 for λ. The blue line is the linear progression and the red line is my log() function. Along with other values for reference.

(b) Growth of $\log_{10} \mathsf{avgloc}(C_1, C_2)$ using a value of 8 for λ. The blue line is the linear progression and the red line is my log() function. Along with other values for reference.

Figure 3.2: Lambda parameters.

snippets of code, but at the same time increase the weight as the snippet grew in size. This is important to filter-out short snippets, which are ubiquitous in Jupyter notebooks, like short print statements, short import statements, and others.

The same logic applies to $\lambda_2 = 8$ (also the red line in Figure 3.2b), which shows a steeper curve that for λ_1, and it is because lines of code (LOC) have more weight than number of characters, again for the same reason of emphasizing longer, more complex snippets of code and filtering-out trivial ones.

Example

In this section I will list two examples of how lambda values control the type of snippet detected as a clone. In Figure 3.3, I show two snippets of code which are very similar for the exception of the last two lines, the *Duplicate Ratio* is 0.63, which makes it too high to qualify as a clone according to the cut-off parameter, set as 0.3. This is because I used a λ_2 value which penalizes short clones that have a high Levenshtein distance, e.g., if I remove the last line **_ = ax. set_title ('Some title .')** from the left side snippet of Figure 3.3, the Levenshtein distance is reduced from 57 to 27 and

the *Duplicate Ratio* is also reduced to 0.35.

```
data = pd.DataFrame(
    data = {
        'Task #1 Average': [0, 2],
        'Task #2 Average': [0.375, 2.5]
    }
)
ax = data.plot.bar(cmap='PuBu')
_ = ax.set_title('Some title.')
```

```
data = pd.DataFrame(
    data = {
        'Task #1 Average': [0, 2],
        'Task #2 Average': [0.375, 2.5]
    }
)
print(data)
```

Figure 3.3: Example of two code cells detected as clones by the Duplicate Ratio Function 1. The Levenshtein distance between the two snippets is 57 and its *Duplicate Ratio* (DR) is 0.63.

```
data = pd.DataFrame(
    data = {
        'Task #1 Average': [0, 2],
        'Task #2 Average': [0.375, 2.5],
        'Task #3 Average': [1.375, 0.875],
        'Task #4 Average': [1.375, 0.875],
        'Task #5 Average': [1.375, 0.875],
        'Task #6 Average': [1.375, 0.875]
    }
)
ax = data.plot.bar(cmap='PuBu')
_ = ax.set_title('Some title.')
```

```
data = pd.DataFrame(
    data = {
        'Task #1 Average': [0, 2],
        'Task #2 Average': [0.375, 2.5],
        'Task #3 Average': [1.375, 0.875],
        'Task #4 Average': [1.375, 0.875],
        'Task #5 Average': [1.375, 0.875],
        'Task #6 Average': [1.375, 0.875]
    }
)
print(data)
```

Figure 3.4: Example of two code cells detected as clones by the Duplicate Ratio Function 1. The Levenshtein distance between the two snippets is 57 and its *Duplicate Ratio* (DR) is 0.27.

Now, in Figure 3.4 we have the exact same code as Figure 3.3, but with the difference that it contains 4 more lines of code. This fact of having more lines of code while preserving the same Levenshtein distance will lower the *Duplicate Ratio* from 0.63 to 0.27, which is within the range of the cut-off value, hence marking it as a *true positive*.

This is how the λ values control the detection of clones in Duplicate Ratio Function 1. There is a ratio between the Levenshtein distance and the length and lines of code of snippets. With the introduction of these weights into my detection function I aimed to introduce bias for complex routines instead of weighting all snippets equally.

The rationale for this decision was that, for Jupyter notebooks I observed that users introduce many quick and short debugging statements which add very little in terms of contributions to the actual code of a notebook. This is probably due to the fact that Jupyter notebooks were described as being "scratch pads", "preliminary work" and "short-lived", as described in the study conducted by Kery and Myers [7]. I think my solution weights in favor of complex routines, which are the ones I considered important enough to be counted.

3.6 Methodology

3.6.1 Detecting Code Cell Duplicates

I started with a random sample of 1,000 GitHub repositories containing 6,515 notebooks [4] . I cloned each repository and looked at the latest commit available. I then extracted all code cells from each notebook from each of the 1,000 repositories. Once I had extracted all code cells from a repository, I ran Duplicate Ratio Function 1 on every code cell, comparing each cell against all others in the repository. Based on the duplicate counts, I calculated a *Repository Duplicates Ratio*, which is the ratio of duplicated cells against the total number of code cells.

3.6.2 Inductive Coding of Detected Duplicates

Finally, once I computed all duplicates throughout the 1,000 repositories, I randomly selected a sample of 500 duplicates and thematically coded [44] them with the help of a colleague from my laboratory. The purpose of this task was to understand the main programming goal of these duplicates. During the thematic coding phase, both my colleague and I tried to answer questions like: *what is the snippet's goal* and *what is the snippet trying to compute*. This process was done after computing all possible duplicates.

This process was done in several iterations. At each iteration, we tried to refine my taxonomy by merging categories together, e.g., *Mathematics* which accounts

[4] All artifacts generated for this thesis are provided at https://doi.org/10.5281/zenodo.3836691

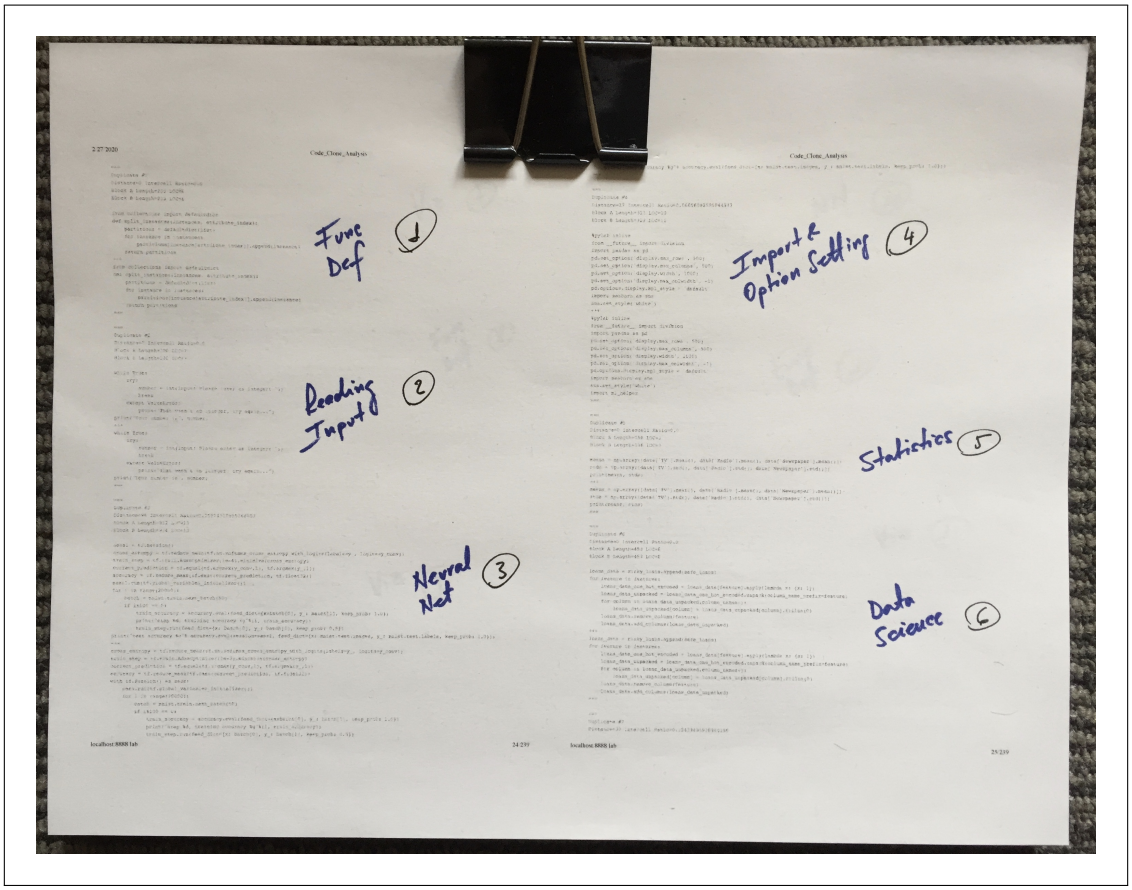


Figure 3.5: Image depicting the process of inductive coding performed by me and a colleague as part of this thesis.

for snippets of code computing math oriented tasks, like Linear Algebra, Numerical Analysis, and others, were merged together with *Statistics* under one main category: *Mathematics*. After our individual coding process was completed, we began to merge our categories together, there was substantial overlap in the categories we came up with, and in the case of differences, we analyzed each snippet individually, to later assign a category by consensus.

3.7 Results

I searched for duplicates using Duplicate Ratio Function 1 on 897 repositories, consisting of 6,386 notebooks containing 78,020 code cells. 103 repositories were no

longer available, and roughly half of the 897 repositories did not contain a single clone. Only 429 contained more than 28 code cells in total (across all notebooks in that repository). Since 28 was the median, and the number of code cells in a repository is exponentially distributed, I discarded repositories with fewer code cells to a) reduce the running time and b) ensure trivial repositories were not counted. Of these remaining 429 repositories with at least 28 code cells, roughly 80 did not contain duplicated snippets. From that analysis, I detected 5,872 Type-1, Type-2 and Type-3 code duplicates in total. My mining results show that 74% (4,355 out of 5,872) of the clones were Type-2 and Type-3, and the rest were Type-1. This result is quite interesting, because it shows that roughly 26% of all duplicates in Jupyter notebooks are exact duplicates (Type-1)! The number of code duplicates in a repository varies mostly between 0 and 100, with some outliers. I now discuss my findings for the distance between duplicates (their duplicate type), the distribution of duplicate ratio (DR), and duplicate purpose.

3.7.1 Duplicate Type

The Levenshtein distance (LD) between code cells follows an exponential distribution, with a median of 21, mean of 41.08, standard deviation of 59.66, minimum value of 0 and maximum value of 535. Most duplicates detected by my function were Type-1 and Type-2 (closer to zero), with a long fat-tail where some Type-3 (further away from zero) duplicates were detected.

At the intersection of the two dashed red lines are the Type-1 (26% $\approx$ 1,500) clones and the rest are Type-2 and Type-3 (74%) clones. From Figure 3.6 we can observe that the clones' Levenshtein distances follow an exponential distribution, and so most clones are closely related to each other. So, increasing the cut-off value would only add false positives, without affecting the overall count much.

3.7.2 Repository Duplicates Ratio

Duplicate ratio measures the number of duplicate code cells over the total number of code cells in a repository. It also follows an exponential distribution, with a median

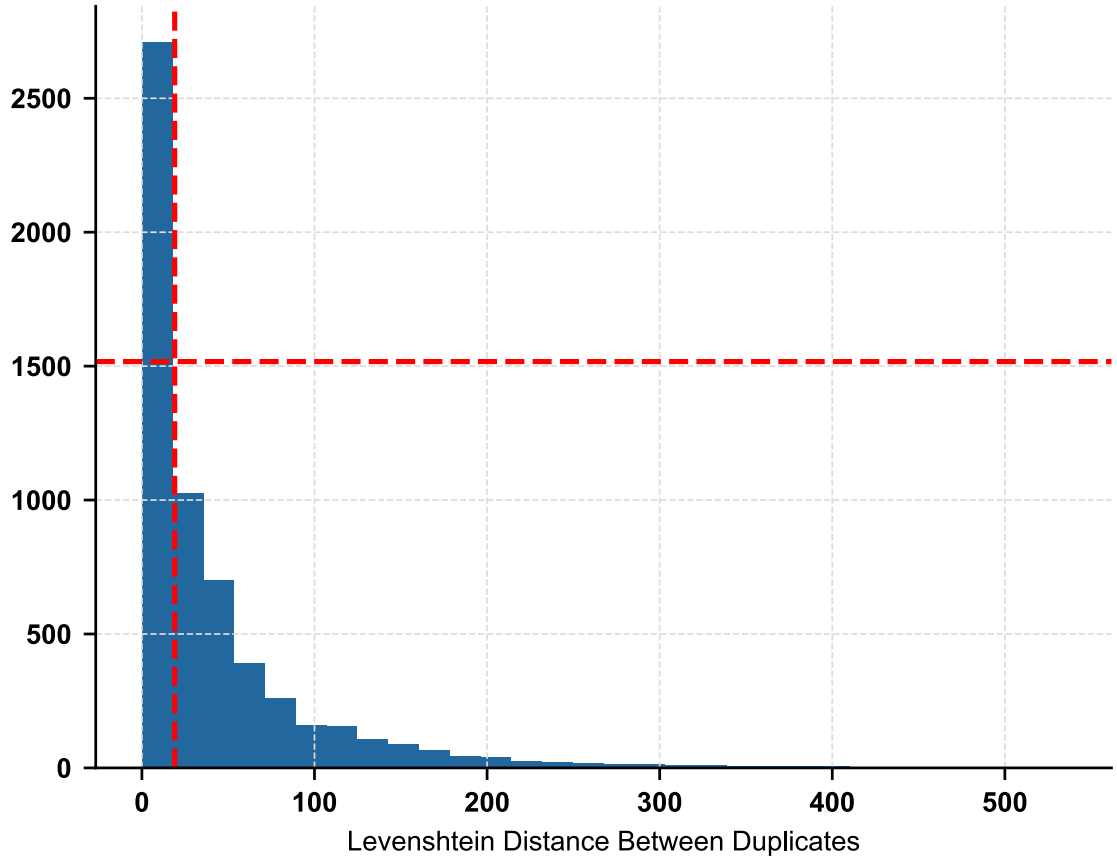


Figure 3.6: Histogram of Levenshtein distances as computed by Duplicate Ratio Function 1 for the *cut-off* value of 0.3. Since this figure corresponds to a histogram and I used bins of size 30, the intersection of the two dashed red lines show the number of Type-1 duplicates (Levenshtein distance equal zero).

ratio of duplicates per repository of about 5.0% with a mean and standard deviation of $\mu = 7.6\%$ (one in thirteen), $\sigma = 8.3\%$. The minimum ratio was 0%, e.g., a repository with no duplicates, and the maximum ratio was 47.5%, e.g., a repository where nearly half the code cells were duplicates.

In Figure 3.7, I plotted the number of notebooks per repository against number of code duplicates (left), and I did the same for code cells (right). We can observe that the number of code duplicates in a repository varies between 0 and 100, with some clear outliers, like the case of the top-right example, where a single repository contained $\approx$ 300 notebooks with $\approx$ 500 duplicated snippets of code. We can also observe that the number of notebooks in a repository seldom surpasses 100, with

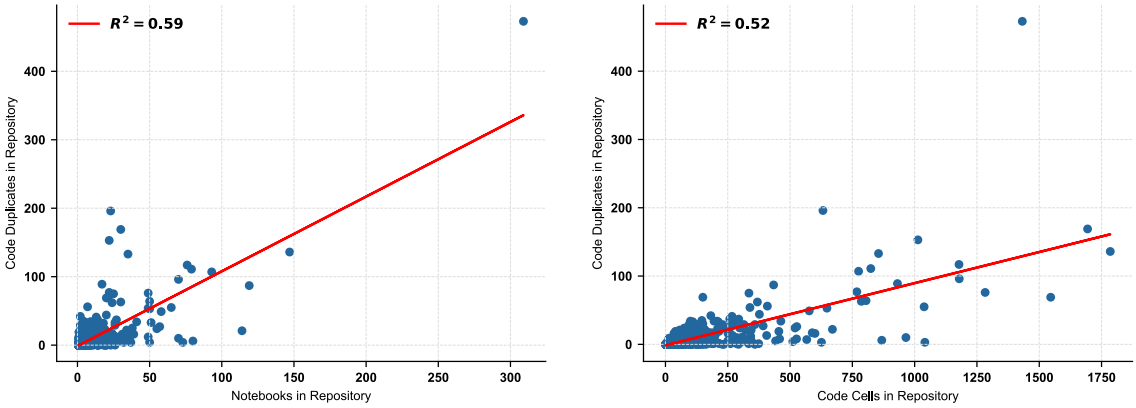


Figure 3.7: Jupyter notebooks against code duplicates per repository (left) and code cells against code duplicates per repository (right). The red lines corresponds to the Regression Line with their corresponding R^2 values.

most repositories containing between 0 and 100 notebooks. Now, in the case of cells per repository, we can observe that the bulk of repositories in this data set contain between 0 and 500 cells, with the majority concentrating at or below 250 cells. Mind that these are code cells per repository and other types of cells are not included!

3.7.3 Duplicate Span

A quality duplicates can have is their span — to how many notebooks inside a repository they were copied to. We can observe a very simple example illustrating this quality in Table 3.1, where the same block of code A is duplicated into notebooks one and three within the same repository; this will give a *span number* of two. The same for code B, which is duplicated into notebooks two and four, given a *span number* of two also.

The results returned by my study are that, for 897 repositories analyzed, the maximum *span number* was 80, which implies that a single snippet of code was duplicates across 80 different notebooks within the same repository. The median value is 1.0 with a mean and standard deviation of $\mu = 1.3$, $\sigma = 3.34$, respectively. Again, if we plot a histogram of this metric we would obtain an exponential distribution, with the majority of *spanning numbers* closer to 0 with a long tail. What this means is that most clones are replicated with or without variations to mostly just one other

Repository A →	**Notebook 1 →**	**Code Block A**
	Notebook 2 →	**Code Block B**
	Notebook 3 →	**Code Block A**
	Notebook 4 →	**Code Block B**

Table 3.1: Example of spanning of clones across notebooks in the same repository.

notebook in a repository. But, in some extreme cases, one clone can be replicated to many other notebooks by the same user.

3.7.4 Coding of Duplicates

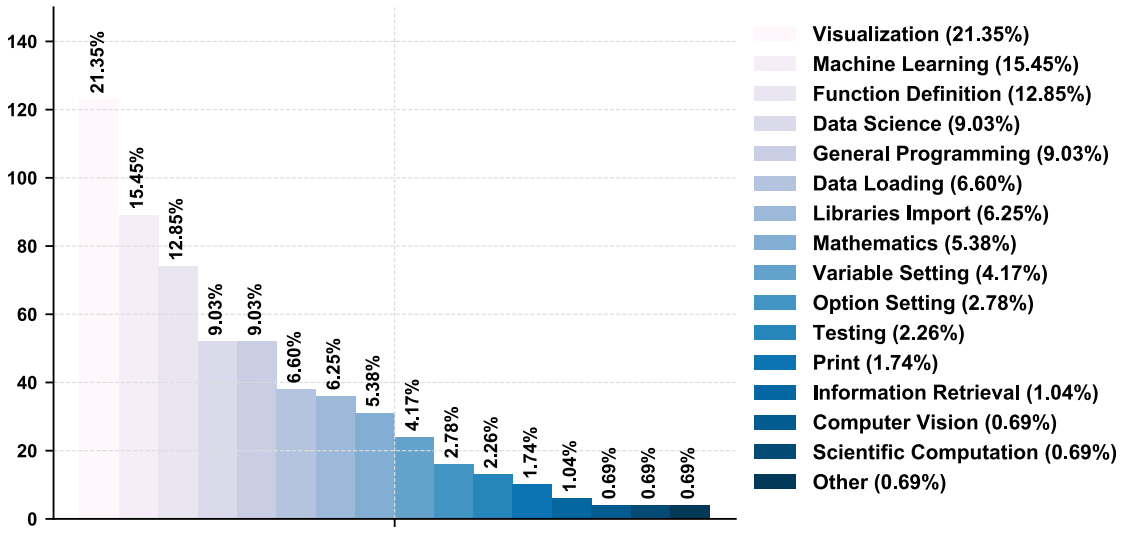


Figure 3.8: Inductive coding of cell code snippets marked as duplicates by my function.

Figure 3.8 shows the result of my inductive coding. Snippets of code that are duplicated the most within Jupyter notebooks are the ones whose main activity concerns *visualization* (21.35%), followed by *machine learning* (15.45%), *definition of functions* (12.85%) and *data science* (9.03%).

3.8 Limitations

3.8.1 Limitations of the Clone Detection Code

Counting unique instances of duplicated code cells in Jupyter notebooks has proven to be a cumbersome task from a programming perspective. Although cells in Jupyter notebooks can contain metadata and other identifiable information, they introduce the inconvenience of not possessing a unique identifier that I could use to uniquely identify a certain cell instance within a repository. In fact, my approach to counting cell duplicates is one that can lead to some under-reporting of duplicates. For each cell, I triangulated the SHA256 hash of its code, the notebook's file name and its number. That approach was used to prune cells that have been counted already, and so avoiding counting the same cell more than once. A very simple example will be a repository with 10 code cells and 5 Type-1 duplicates. In order to count these correctly, I will have to count all five as being duplicates, which will produce a Repository Duplicates Ratio of 0.5 ($\frac{5}{10} = 0.5$), but since Jupyter code cells have no unique identifiers, that I could extract and use to mark cells that have been computed already, I had to use the triangulation method mentioned above. This approach is not perfect, since there can be copies of a notebook that share these exact same characteristics.

This under-count of duplicates constitutes a limitation to my study, since it is my estimate that a small percentage of duplicates may have been omitted by my code. Future studies should take this Jupyter limitation into consideration while attempting to compute duplicates. Jupyter notebooks could introduce unique identifiers for cells — e.g., a timestamp of when the cell was created, or a unique hash based on cell creation information — in order to make the detection and counting of duplicates as accurate as possible. But for this study, it should be noted as a limitation of the study and treated as such when interpreting the results.

3.9 Discussion

Mining software repositories looking for how much duplication occurs in Jupyter notebooks is an important step if we want to understand reuse in this new medium of computation, since copying and pasting from online sources and other notebooks is the most basic form of reuse. Mining software repositories to analyze other types of reuse are needed. Quantifying how many notebooks contain imports to internal modules is necessary if one is to claim this study as complete, but lets consider this as a first step into understanding this form of reuse, viz. duplication.

Detecting duplicates and near-duplicates has proven to be cumbersome from the programming perspective, and a bit elusive as well, maybe that is why industrial clone detection studies varies so much in their detection percentages — between 5 to 20% and sometimes much more [5,38]. The majority of these studies focused on industrial or production software and not on Jupyter notebooks, which can be considered to be a different approach altogether from regular programming paradigms [7]. Given the transient and messy nature of Jupyter notebooks [33], I would have assumed there be a higher count of duplicates, since users of this tool are usually not worried about programming standards, recommendations and techniques. Maybe this is a hint in itself, maybe the right tools are not in place to foster this kind of reuse, or maybe users of this tool users are not prone to this kind of reuse. To dig deeper into these issues I designed and executed a second study, which I present in the next chapter of this thesis.

3.10 Chapter Summary

This first study provides an estimate to *how much* code is being duplicated across GitHub repositories containing Jupyter notebooks, as well as the main programming goal of these duplicates. On average 7.6% of a repository containing notebooks are cloned snippets, and their main programming goal is mostly some type of data visualization. This estimate contains some limitations which were explained in the Limitations section (Section 3.8) of this chapter, and should be taken into considera-

tion when interpreting the results. What this first study could not answer was *from where* these duplicates came from, or *how* users incorporated them into their notebooks. To answer these questions I had to conduct a second study — a Contextual Inquiry — which helped me understand more about users' reuse behaviour.

Chapter 4

Observing Users Using Jupyter Notebooks

> *The power of the unaided mind is highly overrated... The real powers come from devising external aids that enhance cognitive abilities.*
>
> —Donald Normal, *Things That Make Us Smart*

In order to better understand how users of computational notebooks reuse code, I conducted an observational study in my lab. I observed Jupyter users reusing code using Jupyter notebooks, *git* and web browsing.

I used this observational study to answer **RQ2: How does cell code reuse happen in Jupyter notebooks?** and **RQ3: What are the preferred sources for code reuse in Jupyter notebooks?**

4.1 Methodology

I observed the behaviour of eight participants (six M.Sc. and two Ph.D.). All were University students from Computer Science (6) and Chemistry (2); two were female; three last used notebooks over one month ago, five within the past day. Four reported intermediate programming skill, two advanced, and two beginner. I explicitly expressed to my participants that I was not measuring programming abilities. I recruited participants with experience with Jupyter notebooks, irrespective of their level

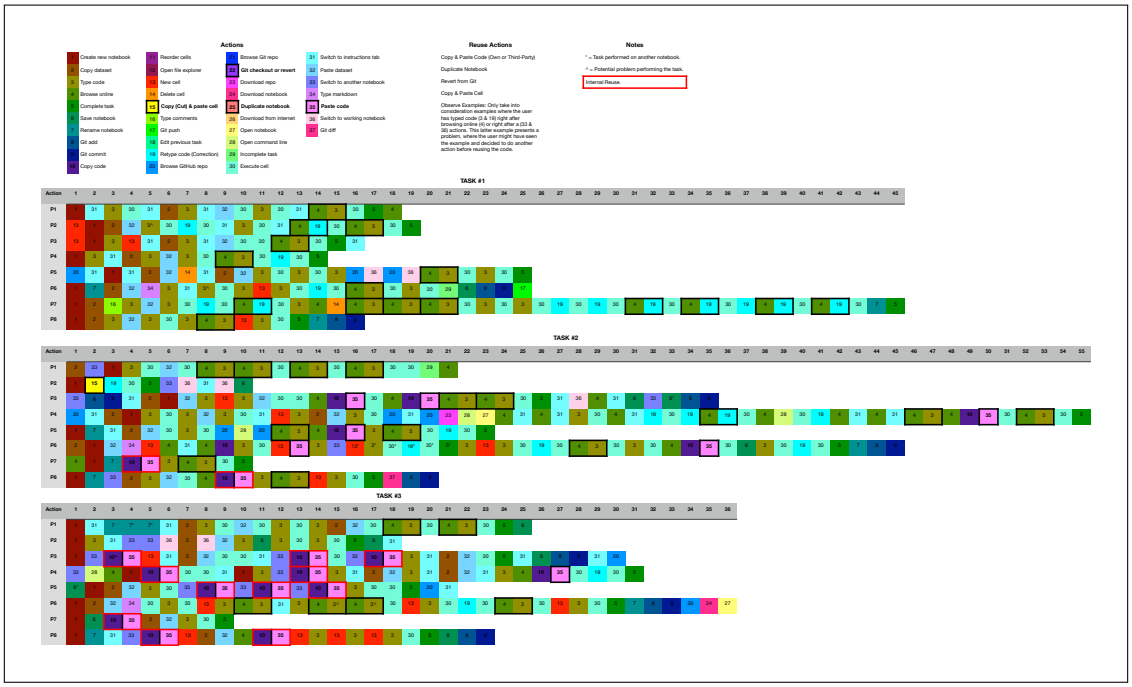


Figure 4.1: Coding of steps my participants made while completing the tasks for this observational study. Coded from video and audio recordings.

of proficiency with programming languages. I drew this convenience sample through personal contacts and email.

Each participant was asked to solve three different tasks using the lab's computers. These tasks were distributed as Jupyter notebooks according to the level of proficiency each participant reported having (Levels A, B, C, below). Each set of tasks were of varying difficulty. Each task was designed to take around 20 minutes to complete, but participants were given more time if needed. Full instructions for how to complete each task was given in full detail on each Jupyter notebook. In total, each participant received three Jupyter notebooks with instructions for each of the three tasks [1] . Two small data sets (10 elements) were also provided within each notebook with instructions. The tasks given to participants were (in the order that they were presented to the participant):

Proficiency Level A

[1] All artifacts generated for this thesis are provided at https://doi.org/10.5281/zenodo.3836691

1. Calculate the mean of a data set.
2. Calculate the sum of all elements of a data set.
3. Calculate the mean of data set #1 and calculate the sum of all elements of data set #2.

Proficiency Level B

1. Calculate the standard deviation of a data set.
2. Create and plot a histogram with all elements of a data set.
3. Calculate the standard deviation of data set #1 and create and plot a histogram with all elements of a data set.

Proficiency Level C

1. Create a function that calculates the mean of a data set.
2. Create a function that calculates the standard deviation of a data set.
3. Calculate the mean of a data set using a function and write the function in the notebook. Calculate the standard deviation of a data set using a function and write the function in the notebook.

Task 1 and 2 were designed to be completely independent of each other, while task 3 was an intersection of the previous two (e.g., participants could have re-used the solutions to task 1 and 2). Tasks that were based on data exploration, remained fairly generic, and did not rely much on external libraries.

The restrictions imposed on the participants on how to accomplish these tasks were minimal. I told them each task should be completed in a notebook different from the one given with the instructions. This allowed me to observe if users created new notebooks or reused old ones. Supported languages were Python, R and JavaScript, which the participant could choose at any time, or change in the middle of the task if needed. Participants were told they could use any resource they found online. I also linked the instructions to a GitHub repository that contained the solutions to each of the three tasks in its *commit tree* (See Figure 4.2).

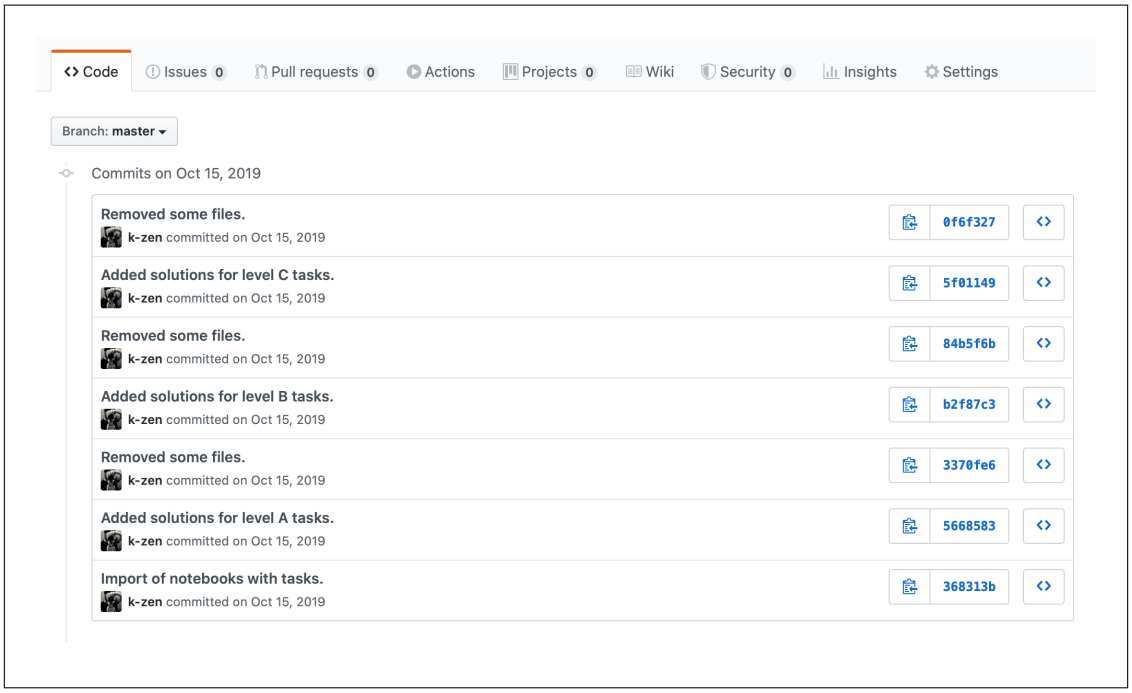


Figure 4.2: Picture showing the provided GitHub web interface with the repository's *commit tree*.

I observed each participant during a time frame of roughly 60 minutes. I took detailed notes of the behaviours each participant displayed. Audio and screen video was also recorded. The observational study was complemented with a questionnaire and a follow-up unstructured interview. Audio, video and notes were coded for qualitative and quantitative conclusions. Video coding was important to quantify the steps each participant took to complete each task, to understand order and to sequence and find patterns in participant behaviours. The video coding resulted in a detailed workflow analysis with tasks coded as shown in Figure 4.1. Audio was transcribed and coded to derive the qualitative aspect of the answers each participant gave.

4.1.1 Coding of Video Data

Video (screen recording) of each participant was analyzed exhaustively and each action performed by the participant was coded, irrespective of its importance. The coded information was: the action performed, the starting and ending time of each

task, and time of action. In Table 4.1 I listed all possible action codes. The coding was done in two stages, the first one was manually, using pen and paper and then all codes were verified and transcribed to a spreadsheet, producing the result displayed in Figure 4.1.

Code	Action	Type	Code	Action	Type
1	Create new notebook	NR	**21**	Browse Git repo	NR
2	Copy dataset	NR	**22**	Git checkout or revert	R
3	Type code	NR	**23**	Download repo	NR
4	Browse online	NR	**24**	Download notebook	NR
5	Complete task	NR	**25**	Duplicate notebook	R
6	Save notebook	NR	**26**	Download from internet	NR
7	Rename notebook	NR	**27**	Open notebook	NR
8	Git add	NR	**28**	Open command line	NR
9	Git commit	NR	**29**	Incomplete task	NR
10	Copy code	NR	**30**	Execute cell	NR
11	Reorder cells	NR	**31**	Switch to instructions tab	NR
12	Open file explorer	NR	**32**	Paste dataset	NR
13	New cell	NR	**33**	Switch to another notebook	NR
14	Delete cell	NR	**34**	Type markdown	NR
15	Copy (Cut) and paste cell	R	**35**	Paste code	R
16	Type comments	NR	**36**	Switch to working notebook	NR
17	Git push	NR	**37**	Git diff	NR
18	Edit previous task	NR			
19	Retype code (Correction)	NR			
20	Browse GitHub repo	NR			

Table 4.1: All codes corresponding to actions participants made while performing tasks. Cells marked in red correspond to reuse actions. The type column means NR=Non-Reuse and R=Reuse.

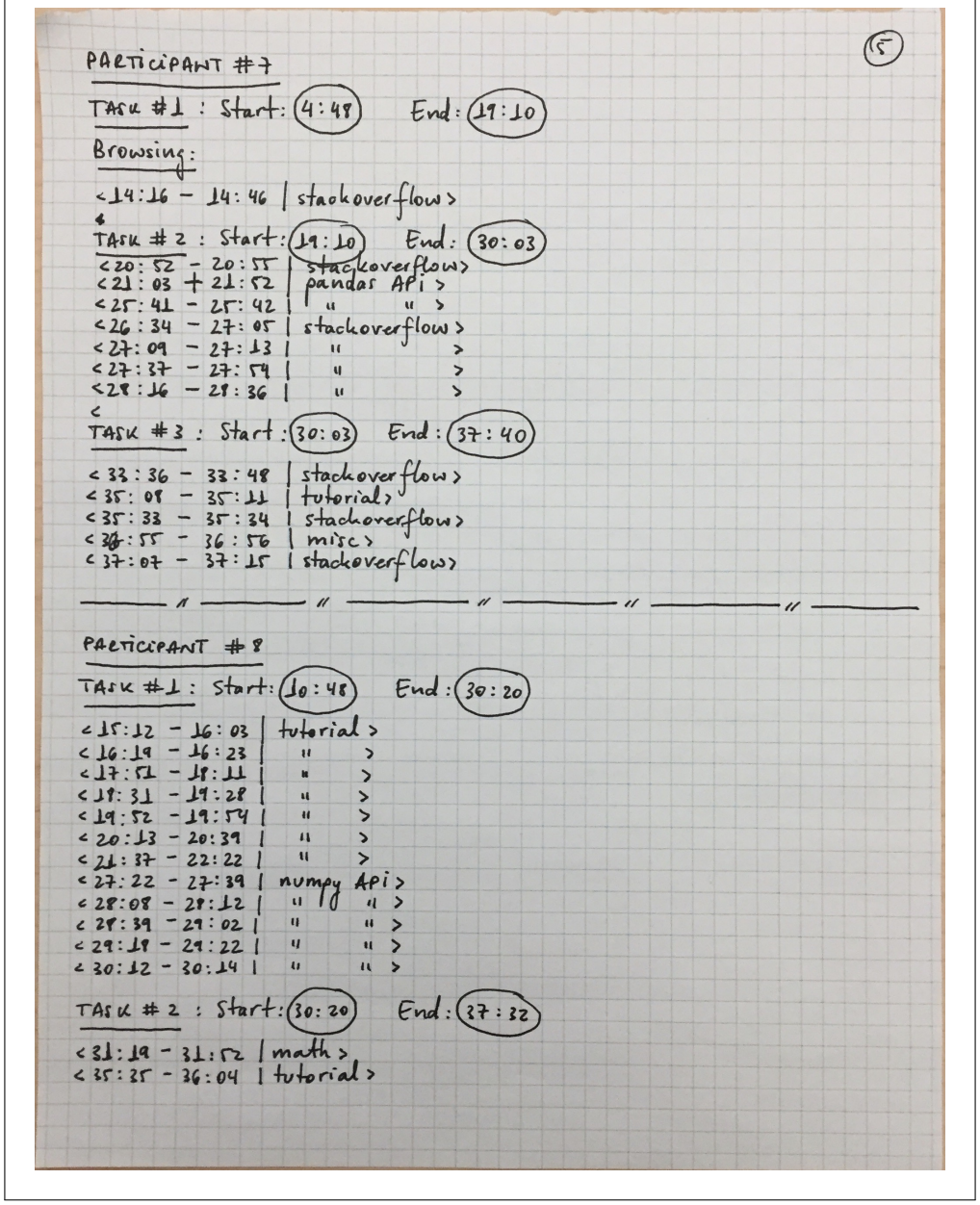


Figure 4.3: Picture showing the process of quantifying how much each participant reused from either internal or external sources. In the picture we can observe the coding of sites they visited while browsing online, along with their corresponding times.

4.1.2 Quantifying Internal and External Reuse

I use the times collected during the video coding to quantify how much time users spent browsing online for information. I collected the start and end times in total for the each task (1, 2 and 3) and then using the video recordings I coded each time a participant browsed online for information, irrespective of if they reused or not after browsing. I also collected which sites they visited and for how long. Once I coded all this information, I combined all browsing times into one final browsing percentage for each participant. This process was done manually using pen and paper (Figure 4.3).

4.2 Results

Participants duplicated code in a variety of ways. After analyzing the video artifacts, I created the following coding scheme to describe how they reused their code:

C&P: Copying and pasting lines of code.

CELL: Copying and pasting code entire cell (reuse from notebooks).

TYPE: Typing code written in another notebook of theirs, instead of **C&P** it.

DUPE: Duplicating a notebook of their own.

GIT: Reusing from *git*.

TYPE_ON: A special case where participants would browse online and would decide to type the code they extracted from the source instead **C&P** it.

NONE: No reuse, directly enter solution from memory.

All participants reused code quite extensively, and only one participant typed from memory (NONE). Most participants reused code from online sources. Foraging for code online is a popular form of code reuse among programmers and analysts, as was pointed out by Brandt *et al.* in [45]. I also observed that participants reused code from internal sources, like other notebooks. This was expected given they had easy access to previous tasks. None of the participants decided to reuse from *git* (GIT),

although full solutions to each task were readily available in the local *git* repository and on GitHub. The full count of each reuse code can be found in the next table (Table 4.2).

Code	Overall Count	Task #1	Task #2	Task #3
C&P	20 times	0 times	8 times	12 times
CELL	1 times	0 times	1 times	0 times
TYPE	0 times	0 times	0 times	0 times
DUPE	0 times	0 times	0 times	0 times
GIT	0 times	0 times	0 times	0 times
TYPE_ON	36 times	16 times	14 times	6 times
NONE	1 times	0 times	0 times	1 times

Table 4.2: Count of reuse codes for all participants and across all tasks. Highlighted in red are the highest counts.

4.2.1 Code Reuse from Other Notebooks

Four out of eight participants decided to reuse what they did in task 1 and 2 for task 3. The other four decided to perform task 3 from scratch, even though they had the necessary code for task 3 already implemented for task 1 and 2 (recall that task 3 is deliberately structured as the union of 1 and 2). When asked why, one participant (P1) said: "*Muscle memory. As a means of preserving knowledge.*"

P2 had trouble copying and pasting the code inside a cell. One explanation for this is that in Jupyter, when doing a right-click of the mouse, only copy or cut at the cell level is available. P4 and P6 both stated that they enjoyed typing.

4.2.2 Code Reuse from External Sources

All participants in this second study used the web extensively to assist in the completion of each task. The time spent browsing online accounted on average for 18%

of the total time spent working on tasks (Browsing time: $\mu = 233 \pm 180$ seconds). I also observed that they relied on online resources like API documentation and tutorials for repetitive tasks, e.g., some participants browsed online more than once for examples on how to import the same library, even though they performed the same import just minutes before (Browsing count: $\mu = 9.6 \pm 3.8$ times). I noted that some participants used the web as a *memory delegate* [45].

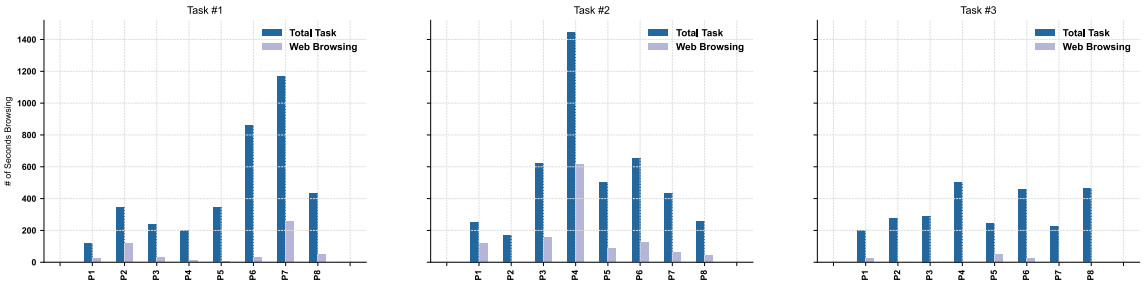


Figure 4.4: Time participants spent browsing online for information, segmented by task.

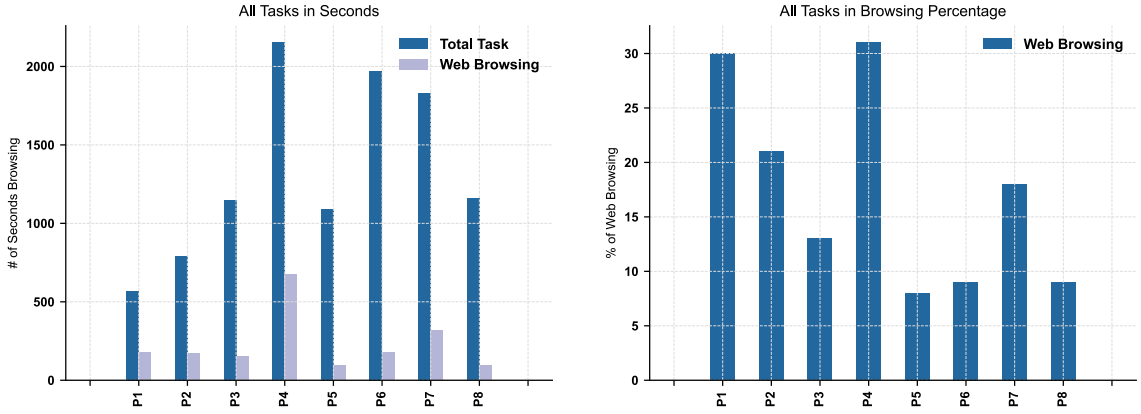


Figure 4.5: Average time participants spent browsing online for information.

Browsing and reusing common libraries in Python was also a popular resource among participants, as all of them relied heavily on the *numpy* library for solving the tasks. I noticed this reliance on external sources and libraries saved time and effort for the participants, since calculating the mean of a data set took them at most two lines of code to accomplish using the *numpy* library (including the import statement) instead of using *for loops*.

Participant	Total Time	Browsing Time	Percentage	Count
P1	570 sec	176 sec	30%	11 times
P2	792 sec	170 sec	21%	3 times
P3	1148 sec	155 sec	13%	7 times
P4	2152 sec	678 sec	31%	14 times
P5	1091 sec	95 sec	8%	6 times
P6	1972 sec	178 sec	9%	13 times
P7	1830 sec	317 sec	18%	14 times
P8	1158 sec	95 sec	9%	9 times

Table 4.3: Portion of study spent browsing online per participant. *Total Time* refers to the total amount of time performing all tasks and *Count* refers to the number of times participants opened a browser to browse for information.

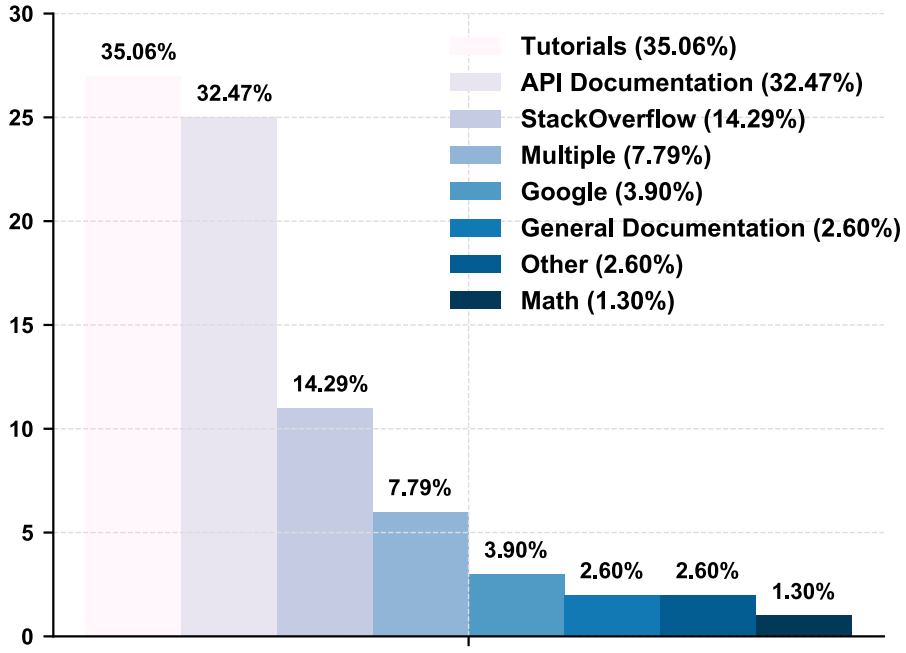


Figure 4.6: Inductive coding of sites participants visited while solving tasks. Note: *Google* implies information taken directly from Google's results page.

Figure 4.6 shows my coded responses for which sites were visited according to

primary role. The two most visited sites among participants were tutorials (e.g., tutorialspoint) and API documentation, followed by Stack Overflow. Participants queries were usually something short and precise, like "*mean numpy*" or "*histogram numpy*", but in some cases I observed longer, more natural language oriented queries, like "*what is git repo? and how to use it?*" or "*how to create Jupyter project*".

I observed two distinct browsing habits: half the participants visited web sites nine times or fewer (Table 4.3), spending overall 129 seconds on average. For example, P2 spent 21% of their time browsing online, but only did this three times. They took their time skimming the web page for code reuse. The other half had 11 or more visits, taking 337 seconds on average. This group used web resources like an external memory aid, going back and forth between the working notebook and the online resource multiple times.

4.2.3 Code Reuse from VCS

I observed that although some participants went to browse for the provided solutions on the study's GitHub repository, they either did not restore them from the *git* history, or lost interest after a few tries. The solutions were not readily available at the HEAD of the commit tree, so knowledge on how to traverse the commit tree and on how to move the HEAD of the tree to a particular commit or how to *checkout* a particular commit was necessary in order to access the solutions. That is, above average knowledge of *git* was necessary for reusing code from the local repository.

I received various answers from my participants when asked why they did not restore the provided solutions from *git*. Two out of eight participants stated "Sufficient Knowledge" as their answer. It was implied by these participants that the tasks were not sufficiently difficult to merit restoring them from *git*. And they perceived restoring from *git* as far more time consuming than actually coding the task.

There was also a perception of complexity in restoring from *git* as noted by one of my participants (P2), which I will discuss in the next section.

Perceived Complexity of Reusing from VCS

One behaviour that struck me as interesting while observing participants solving tasks, was the perceived complexity of restoring from *git*, which they seem to show each time they went for the solutions on the repository. As noted by one participant (P2), when asked why they did not restore from *git*:

> "*. . . if I got really stuck, then just look things up, because probably it would have taken me more time to find it in git, than actually do it myself.*"

Other recorded answers were:

P1: "Again is personal, like personal learning, where this is for more time crunch then maybe or if I had no idea on how to do it I probably would have seen the answer but because I want to kind of familiarize myself again looking straight at the answer didn't seem to make a hole lot of sense for me."

P5: "I went for it but I couldn't find the answer in this little thing here. I don't even know what any of the buttons mean, I don't know I just was following the link and imagined that it would just be right there."

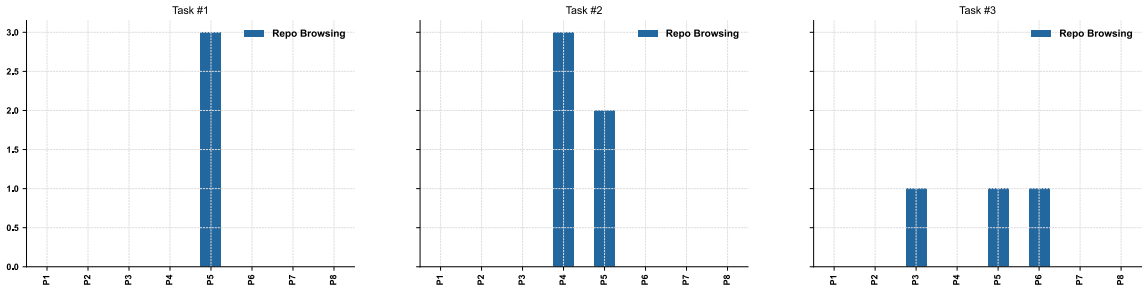


Figure 4.7: Participants who tried to reuse from *git*.

In Figure 4.7 we can observe that some participants tried multiple times with no success to restore the solutions from the repository. These tries account for intents using either the command line *git* tools or the GitHub web interface.

It is important to mention that two participants claimed to have missed the part in the instructions where it mentioned the solutions, and two participants claimed

to have misunderstood the instructions, they claimed to have understood that the solutions *needed* to be in the repository, which it may explain why they pushed their code into the repository.

And finally, one participant (P4) went looking for the solution for task 2, but they had problems finding it. When asked, they stated that the solution was not there, because they had looked only at the last commit and failed to look for it in the commit tree.

4.2.4 Internal vs. External Reuse

Internal reuse entails foraging for information in previously created notebooks (e.g., previously completed tasks for the study), while *external reuse* entails foraging for information online (e.g., web sites, forums, Stack Overflow, etc.). As we can observe in Figure 4.8, participants foraged more from external sources for tasks 1 and 2, while the balance shifted a bit more towards internal sources for task 3. This behaviour is not surprising since it is only natural to assume given the setup of the experiment that participants will reuse what they did for tasks 1 and 2 on 3. One participant (P2) reused from memory, as we can observe in task 3 in Figure 4.8.

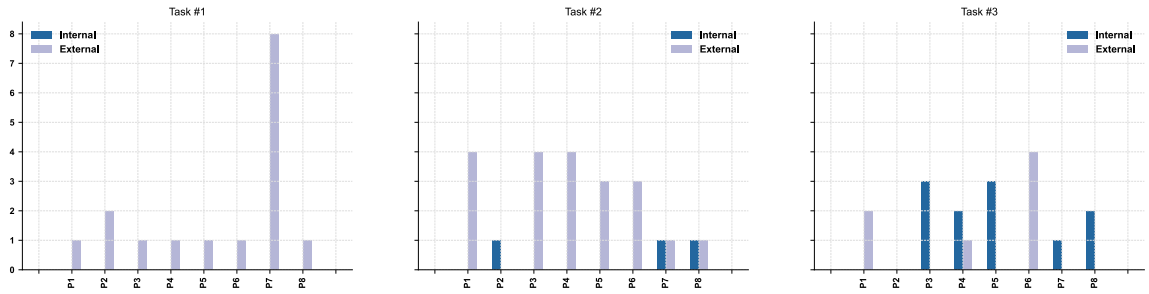


Figure 4.8: Reuse from internal sources vs. external ones, segmented by task.

4.2.5 C&P vs. TYPE_ON vs. NONE Reuse

In Figure 4.9 I listed statistics for the number of times participants decided to use **C&P** instead of **TYPE_ON**. For task 1, participants typed the solutions. For task 2, the balance is almost equal, having participants reused with both methods almost

equally. And for task 3, participants decided to **C&P** more, with the exception of only one participant (P6), who decided to redo entirely task 3 instead of reusing what they did before. One participant (P2) typed from memory (**NONE**, as we can observe for task 3 in Figure 4.9.

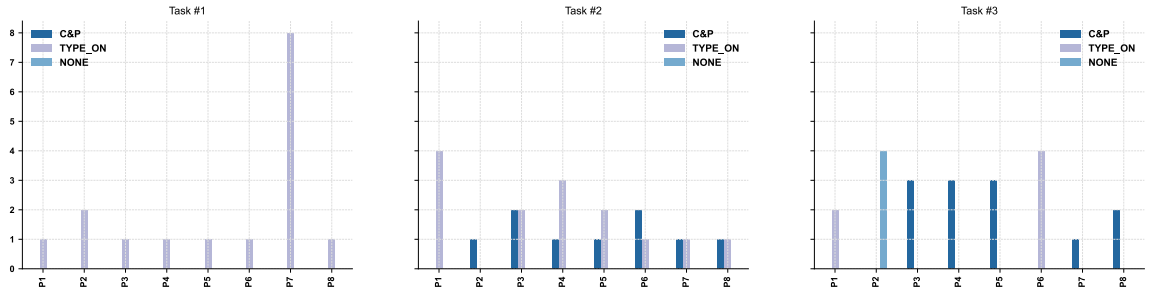


Figure 4.9: Number of times participants went browsing online for information, segmented by task.

Overall participants followed the desired pattern of reusing what was done in previous tasks to complete the final one, which may imply, that during exploration tasks users tend to reuse from the latest and most available notebook they worked on, interwoven with online browsing.

4.2.6 Writing to *git*

Three out of eight (37.5%) participants decided to commit their completed tasks to the local *git* repository, although that was not a requirement for any of the tasks. This behaviour reinforces the hypothesis I had at the beginning of my study that *git* is being treated commonly as a "write-only" medium of storage, as a safety measure for user's code, where all changes are being written to, in case they will be needed in the future, but they are seldom read from.

After asking these participants about their behaviour, they answered the following:

P3:

Me: Why did you commit to the repo, if it wasn't in the instructions?

Participant: *Just to keep track of the work.*

Me: Why did you wanted to keep track of the work?

Participant: *Because if, sometimes, when working on any project I get lost, if I made an error in the... is very simple, just a habit of doing it.*

Me: So when you work on projects using notebooks you usually use *git*?

Participant: *Yes*

Me: So, you tend to commit to *git* a lot?

Participant: *Yes, when I work in completing little functions, I commit.*

P8:

Me: So, why did you added your code to the repo?

Participant: *I don't know, just habit. Like I work on many machines so, I like storing things in GitHub, um, that way if I do go and make a change to something and ends up not being good I can just revert it and go back to a "working good state".*

Me: So, you do revert to previous versions of your notebooks?

Participant: *Um, I wouldn't say I do it very often with notebooks, um, but with code or any plain text in general... - the problem again with can get into it if you want... but the problem with the notebooks is that it contains all that metadata, it's kind of a double edge sword, right? So sometimes is good to have that metadata because I've worked on some notebooks that are fairly computationally complex and the output is saved and my solution too... it depends if you are working in a Jupyter notebook with one person or more than one person... If you are the only person doing commits then it's OK, but if you are working with other people that are doing commits, well, even if you are the only person, if you want to see what a particular commit did, it's very hard to look for just the code, right? Like if I want to look at a diff of code. I like to look at diffs of code to see what it is.*

4.3 Limitations

4.3.1 Observer-expectancy Effect

Some of the participants of this study might have been affected by the observer-expectancy effect [46, 47], due to the fact that some participants knew me and probably wanted to boost their effectiveness by performing tasks more rapidly that they would normally have done. Although, I explicitly explained before the study, that no programming skills were going to be measured during this study, this might have influenced the result of it.

4.3.2 Limitations of GitHub's Interface

GitHub provides a function for rendering notebooks on-site, which during the experiment was not functioning properly. This was noted by most of the participants who decided to browse for the solution. It does not present an impediment to finding the answer, but it does hinder notebook exploration.

4.4 Discussion

In this section I will discuss my results and post-study observations. I observed three distinctive reuse behaviours while conducting my observations: different sources of reuse are harder to attain, users used the web as a sort of *memory delegate* [45] and finally, how users treated *git* as a "write-only" tool for redundancy.

4.4.1 Foraging for Information

Information Foraging Theory is a theory of how humans seek information in information intensive tasks. It applies ideas from Optimal Foraging Theory to understand how human users search for information. This latter theory is based on the assumption that, when searching for information, humans use "inherent" foraging mechanisms that evolved to help them hunt for food in the wild [48].

Optimal foraging theory is a theory in biological science of how animals hunt for food in the wild. Pirolli *et al.* found similarities between users' information search patterns and animals' food foraging strategies. They used these similarities to develop IFT (Information Foraging Theory) [48, 49]. One similarity between both theories is that humans are predators who search for information in the same way they search for preys. IFT is based on what the authors described as *information scents*. Humans assess how much useful information they can attain on a given information source or *patch*, against the effort required to attain that information. They use *information features*, e.g., folder names, variable names, web links, which the user can process to gain knowledge. When the chances of obtaining useful information from a particular *patch* dims or when the cost of attaining information from that particular *patch* is too high, they move on to a different one. Humans perform this assessment very effectively and unconsciously, and it is my assumption that it is directly related to how users navigate folders, source code repositories, the web, and computational notebooks. In each *patch*, predators make one of three choices: (i) to forage for information within the patch, (ii) to forage to a new patch or (iii) to engage in enrichment by modifying their environment. These choices are informed by the predator's *information scent*, gathered from cues in the environment.

Understanding how humans search for information can help improve the usability of the Jupyter notebook interface, and foster the creation of new plugins and extensions that might enable users of computational notebooks to find and reuse code effectively and efficiently. I have seen this to be especially true when it comes to reuse behaviour linked to some sources of information, such as: version control systems and source code repositories (*git*, SVN, Mercurial, others), potentially due to interface complexities.

After conducting my observations I noticed behaviours that could be further explained and understood by applying IFT concepts, like when some participants tried to reuse code from version control systems (*git*). When this occurred, I observed that some participants tried multiple times with no success to restore the solutions from the repository, and according to IFT: even though the *information scent* is strong enough for the VCS *path*, the effort required to attain it is higher than the reward,

hence the participant decides option (ii) *to forage to a new patch*, which in the case of this study I observed to be online reuse of code. It appears to be much easier for users of computational notebooks to launch a web browser and look for code examples on sites like Stack Overflow, than it is to navigate complex interfaces like *git* through the command line. This, I noticed to be especially true for simple tasks involving just a few lines of code. The more complex the reuse necessity becomes, the more I assume the reward of the *path* would be, and users will probably try harder to attain it, but more studies are needed to validate this idea.

4.4.2 External Memory and the Google Effect

The *Google Effect* or sometimes also called *digital amnesia* is the effect of forgetting information which is easily available on the internet [50]. During my observations I noticed this behaviour in almost all of the participants. I observed that they relied heavily on search engines and other sites like Stack Overflow and tutorials for very simple tasks, some which participants had completed just minutes before. For instance, I noticed one participant searching online multiple times for how to import a specific library into Jupyter, when just minutes before they performed that exact same task. This has lead me to infer, that users of Jupyter notebooks rely heavily on online browsing to perform their tasks and analyzes irrespective of the complexity of them. Brandt *et al.* in [45] referred to this as a *memory delegate*, and it aligns with my observations, for many participants did use the internet as a sort of *memory delegate* while completing their assignments.

4.4.3 VCS as Write-Only

An emerging trend I noticed for some participants, was how they treated *git* as a "write-only" medium of storage. It seems they committed their code into the repository out of habit or in case they needed to go back to retrieve code again. This was despite the fact that the simplicity of the tasks did not merit such an elaborate redundancy.

Perhaps, these participants were used to committing their code on a daily basis

for other projects, and this behaviour was ingrained in them, so much that they ported it to this study. Overall, what I observed was that writing their changes to the *git* repository did not took them much time or effort, so, it did not affect their performances at all, they were still able to complete the tasks in time, and they felt comfortable with the process of committing, kind of like a peace of mind.

4.5 Chapter Summary

This second study shows *how* and *from where* code is being reused for Jupyter notebooks. In this study, I tried to answer **RQ2** and **RQ3** by using an observational study (Contextual Inquiry), complemented by a questionnaire and a brief unstructured interview.

The main takeaway of this second study is that it appears that code from online sources seems to be easier to attain, outweighing other ones like reuse from other notebooks and of version control systems. As for the preferred method of reuse, there seems to be an implication in the results that **C&P** outweighs other methods, like typed reuse (**TYPE_ON**), notebooks duplication (**DUPE**), and the others.

There is still much to be done in order to fully understand the *how* and *where* questions posed as part of this work, but these will be mentioned in the future work section in the last chapter of this thesis.

Chapter 5

Discussion, Limitations & Implications

5.1 Discussion

For my research into "Code Duplication and Reuse in Jupyter Notebooks" I used two research strategies to triangulate my understanding of code duplication and reuse. In my first study, I conducted a computational data analysis of existing GitHub notebooks, with the intention of quantifying and understanding duplicated snippets of code. In my second study, I focused on how that duplication occurs with human observation in a controlled lab setting. I observed that in both studies duplicated code was important for users of this tool to support their exploration sessions and achieve their results in the least possible time. I also identified some curious behaviours regarding the use of version control in conjunction with Jupyter notebooks, and finally, I revealed some patterns in the use of external sources for reuse, viz. reuse from web sites.

In this section I will discuss my findings through an analytical lens, explaining my results and findings, and discussing the limitations of both my studies and how these should be taken into consideration when interpreting my results.

5.1.1 Code Duplicates and Their Programming Objectives

Mining software repositories looking for how much duplication occurs in Jupyter notebooks is an important step if we want to understand reuse in this new medium of computation, since copying and pasting from online sources and other notebooks is the most basic form of reuse. Quantifying how many notebooks contain imports to internal modules is necessary if one is to claim this study as complete, but lets consider this as a first step into understanding this form of reuse, viz. duplication.

Figure 5.1 shows an example of a Type-2 snippet of code detected by my function (Duplicate Ratio Function 1). We can observe in the figure that both blocks of code are similar, with the exception of the variables passed as arguments to the plotting function. My first study analyzed and categorized these types of duplicates thoroughly, by first counting how many duplicates were present in a repository and finally by inductively coding these duplicates according to their main programming goal.

```
plt.figure(figsize=(7,7),dpi=400)
ax = plt.subplot(2,1,1)
plot(PPT, Nash_Flow, 'bo',
markersize=3)
title('PPT', fontsize=14.,
y=1.02, fontweight='bold')
ax = plt.subplot(2,1,2)
plt.hist(PPT)
np.corrcoef(PPT,Nash_Flow)
```

```
plt.figure(figsize=(7,7),dpi=400)
ax = plt.subplot(2,1,1)
plot(H1, r2_Sed, 'bo')

title('H1', fontsize=14.,
y=1.02, fontweight='bold')
ax = plt.subplot(2,1,2)
plt.hist(H1)
np.corrcoef(H1,r2_Sed)
```

Figure 5.1: Example of a Type-2 duplicate detected by Duplicate Ratio Function 1 with Levenshtein distance of 42 and Duplicate Ratio of 0.27. The main programming goal of this particular snippet was coded as *Visualization*.

My results show that on average 7.6% of a repository is self-duplicated code (26% exact duplicates and 74% near-duplicates), spanning the same duplicate in some cases up to 80 different notebooks within a single repository, which clearly indicates that users tend to reuse the same notebook on different occasions and for different purposes. There was an extreme case of cell reuse, where a single import statement was duplicated across 80 different notebooks, sometimes exactly and other times with

little modifications to it. In this particular case, it would be beneficial to the user to put those imports into modules, which can be imported and called from a script inside each notebook, instead of copying and pasting it across them. In that particular case, putting the same import into a script would reduce clutter in the notebook, while at the same time creating a single point of modification.

My first study also shows that snippets regarding some form of data visualization are the ones to be duplicated the most, roughly 21.35% of the time. It was my intuition that given the ease of use and visualization functions provided by Jupyter notebooks, that they would be used mostly for some form of data visualization supporting analysis and exploration. Of course, the fact that visualization snippets are duplicated the most, does not imply in any way that this is the main use given to notebooks.

Detecting Jupyter cell code duplicates has proven to be cumbersome from the programming perspective. Jupyter notebooks have a limitation, in which they don't store information that could be used to uniquely identify a particular cell in them. This limitation made almost impossible to count the exact amount of duplicates in a repository, which is something I discussed thoroughly in the limitations section of Chapter 3. Irrespective of this limitation, I am most certain I was able to approximate the actual count of duplicates per repository within an acceptable margin of error. Another aspect that made the detection of duplicates a cumbersome task was the tuning of parameters that Duplicate Ratio Function 1 uses to detect duplicates, these were tuned according to manual inspection after runs with smaller samples, eventually leading to the selection of parameters that produced the best results. This parameter selection process could be further augmented by using an *oracled* data set [43]. An *oracled* data set of clones is one classified and inspected manually, by which a detection algorithm could be tested for precision and recall.

Overall, this clone detection study shed some light into the question of how much of a notebook is duplicated code, and what is the goal or nature of those duplicates. It is a first try and much work still is needed in order to have a specific count of duplicates in Jupyter notebooks. Modifications to notebooks could also be introduced in order to support this counting process, since, as I mentioned in the previous paragraph, Jupyter notebooks do not support unique identification of code cells, hence making

the process of counting much harder.

5.1.2 Methods of Reuse

Participants in my study reused code using almost all available methods, with the exception of duplicating notebooks (**DUPE**), this was a surprise since I expected they would duplicate notebooks as they progressed through the tasks, but instead they decided to create new notebooks for each task and fill them with the necessary code to achieve the result. Copying and pasting (**C&P**) code, either from other notebooks and from online sources was common, as was typing code (**TYPE** and **TYPE_ON**). In some instances, I observed participants observing code present in web sites for later to re-type that same code instead of copying and pasting it, which was strange, since it is much faster and convenient to just select the relevant code, copying it into the clipboard and pasting it in the desired notebook. I also observed that one participant had a few inconveniences with the shortcuts in place for copy and paste, which may explain why they resort to typing the code. These inconveniences were related to the JupyterLab environment, which provides shortcuts to copy and paste at the cell level (**CELL**) instead of individual lines of code. Only one participant decided to reuse from memory (**NONE**) instead of relying on previous code, but this can be explained due the to simplicity of some tasks. Some participants intended to reuse from the repository in which the code resided but were not able to do so, possibly due to inconveniences with either *git* or GitHub's interface, and some did not read fully the instructions where it stated that this type of reuse was available.

Adding better support for reusing previously used snippets will greatly enhance reuse in Jupyter notebooks. One such missing function is one offered by the Google Colab's team, called "Code snippets" (See Figure 2.2). This function permits users to index and search previously used notebooks, from where code can be reused with a single click. It is my opinion that this type of reuse function will greatly facilitate reuse and speed up exploration in notebooks, and it is one missing from JupyterLab.

Another missing function from JupyterLab is one that allows users to pass parameters to notebooks, much like a script would do. It is possible though to achieve this

functionality but some modifications and scaffolding is needed first, akin to Papermill [31]. Passing parameters to Jupyter notebooks allows one to design notebooks that are more flexible, which in turn may foster better programming practices at the time of designing these notebooks. By passing parameters to notebooks at runtime, one might infer that the nature of these tools will change more towards a fully functional reusable script than a transient scratch-pad.

5.1.3 Internal Code Reuse

While conducting my observational study I observed participants reusing quite often what they did before from internal sources. This kind of *internal* reuse is one that comes from one's own notebooks; for the participants in my observational study these were previous tasks. It also came to my attention that some participants did not exploit the full potential of reuse tools offered by JupyterLab, viz. duplicating a notebook. Not using these functions may add redundant time to one's exploration and in some cases produce sub-optimal results. Past studies had researched about *in-notebook* reuse, offering tools that supported history traversing and reutilization [23, 51], but after interviewing participants I could understand that most issues aroused when they had to find past code inside their own codebase. This lead me to hypothesize that better indexing and searching tools are required, along with ways to quickly execute a notebook from outside the programming environment. This quick execution of notebooks is already being offered on GitHub and nteract [16], where it is possible to quickly and conveniently render a notebook for visualization. Interfaces like this foster internal reuse, but come at the expense of having to save the notebook with its output. Sometimes this is not an issue, but when the output is considerably in size (hundreds of megabytes), then it could become an issue.

5.1.4 External Code Reuse

I found the use of external sources very common. In Figure 4.6, I reported on the most frequent types of sources used. Participants frequently skimmed these sites and used them as external aid, and accessed them for short periods of time (browsing

statistics per participant can be found on Table 4.3).

I also looked at what types of duplicates were most common in Figure 3.8. Given the results, it seems like a possible correlation exists between task familiarity and importance. Visualization snippets, for example, are frequently duplicated, because they are vital to the analysis process, but often unfamiliar to the analyst, who may not have extensive visualization training, and instead relies on support from libraries such as Altair or GGPlot.

Duplication initially seems like a major time saver — since the chart, for instance, can be quickly reproduced — but eventually adds to technical debt and maintainability issues [29]. At that point, the duplication can be refactored into a common module, e.g., the corporate visualization module that defines fonts, themes, label sizes, etc., or common functions and classes used across notebooks. Commercial data science teams at places like Netflix have begun to support this process with extensive scaffolding around the basic notebook metaphor, for example, with Netflix's Metaflow or AWS Step Functions. Even further back, scientific workflow software [52] has been managing data processing models for many years.

Identifying technical debt can be difficult for people unfamiliar to software engineering best practices, that is why I recommend creating modules as soon as possible. Jupyter notebooks offer a "magic" function called *autoreload*[1], which can be used to reload imported libraries automatically. Its use is pretty straightforward, and many examples can be found on the internet on how to use it. The benefit of this *autoreload* function is to allow for one to modify a module outside of Jupyter with changes ported automatically to the kernel running the notebook, which makes the overall process smoother. So, one can use an external IDE like PyCharm[2] or other to work on the *backend* code/module and use Jupyter notebooks only as *frontend* — an interface from where to execute the module and retrieve results. This workflow will surely reduce technical debt, with only the minor inconvenient of needing a bit more work to set up the initial environment.

[1] https://ipython.readthedocs.io/en/stable/config/extensions/autoreload.html

[2] https://www.jetbrains.com/pycharm/

5.1.5 Use of Version Control

Version control for notebooks has been the focus of several notebook plugins, many blog posts and feature requests (e.g., jupyterlab-git[3], verdant[4], nbdime[5]), and several research studies [23, 34, 51, 53]. Some notebook services like Nextjournal[6] value the importance of preserving history in computational notebooks so much, that they offer automatic versioning of the notebook and related artifacts. My first study did not examine the role of version control as a source for duplication, in part because identifying duplication from version control is tricky (since files can be renamed, sections moved, etc.). However, part of my lab study was intended to explore how version control systems (VCS), specifically *git*, were used in code reuse. I saw that users struggled with the interaction model of *git* and were unable to use it for duplication purposes.

5.2 Limitations

5.2.1 Construct Validity

The main constructs I discuss are *code duplicate* and *code reuse*. I used the Levenshtein distance between code cells to detect duplicates, similar to Duala-Ekoko and Robillard in [42], which is different than using an *Abstract Syntax Tree (AST)*, which is more common in other code cloning research. This was due to the nature of Jupyter notebooks, which can support multiple kernels and programming languages, so I required a cross-language solution. I also set thresholds for duplicates, which are determined empirically based on soundness of the duplicates. However, for a different sample, these thresholds would provide sub-optimal results. In the future, an improvement would be to use a systematic analysis or grid search to find the parameters for duplicate detection and compare the detection results with an *oracled* data

[3] https://github.com/jupyterlab/jupyterlab-git

[4] https://github.com/mkery/Verdant

[5] https://nbdime.readthedocs.io/en/latest/

[6] https://nextjournal.com

set [43]; this is especially true for λ_1 and λ_2, which control weights for the length and lines of code. However, I am not claiming general results for code duplication in all notebooks, and my findings should be seen as restricted to this sample. Also, given the emphasis I assigned on larger snippets and quantity of lines of code, there could be some under-reporting of duplicates, especially of shorter, more concise snippets of code, like short print statements, and other debugging techniques.

As I observed users to detect code reuse, it is possible that my small tasks did not test all forms of reuse. For example, my protocol did not allow for reuse from other participants. Similarly, users were constrained to use lab equipment. Using their own devices may have shown different reuse techniques (e.g., local copies of documentation).

5.2.2 Internal Validity

I used 897 randomly selected repositories, consisting of 6,386 notebooks and eight convenience sampled students. My random sample of notebooks relied on the data set created by Rule in [54]. Because I sampled 897 repositories and only considered inter-repository reuse, I may have missed reuse derived from external sources, such as popular repositories, tutorials, and training material. This almost certainly led to an under-reporting of code reuse.

Convenience sampling does not support statistical generalization, but since this was an exploratory study, generalization of the observed phenomena was not one of my objectives. While it is possible the behaviours I report on were unique to this sample, the participants all engaged in data analysis for several years in undergraduate and graduate courses at the university.

I asked for self-assessed proficiency with notebooks and version control. This has a potential observer-expectancy bias because I was known to the participants, who may have wanted to inflate their self-assessment. As an exploratory study, this is acceptable, but to test a specific hypothesis, my measures should be less prone to bias.

The tasks were scaled for the estimated skill of my participants. For example,

np.mean(lst) is sufficient to solve task 1 part 1, and for an experienced data scientist, could be retrieved from working memory. However, the tasks were designed for students, and if I were to use professionals, the tasks would have been more challenging. These tasks were designed to stimulate external/distributed cognition. I used three levels of complexity for my study, so that experienced analysts were given tasks commensurate with their skill. However, this is an imperfect matching process.

5.2.3 External Validity

I sampled from notebooks on GitHub, which may differ from notebooks used in corporate settings. Similarly, the use of students as subjects makes it difficult to draw generalizations about industry practitioners [55]. That being said, the students in my sample reported using notebooks (and the other tools) frequently, and information on how professionals use notebooks is still limited.

5.3 Implications

I conclude this chapter with implications for practice and research based on the results of both studies I conducted.

5.3.1 Implications for Code Duplication and Reuse

Practice Implications

After conducting my study, I argue in favor of tools that support this type of reuse, whether they are offered through functions like Google Colab's *Code snippets* or similar ones designed for other types of notebooks. I also argue, that when a notebook's codebase becomes to extensive, and to avoid incurring in technical debt, the user could create independent modules which can be imported locally using JupyterLab's *autoreload* function. This function allows to automatically port changes made to local libraries without the need to reload them manually. This type of reuse by using modules should facilitate the reuse of a codebase across other notebooks in the

same project as well, which will decrease technical debt and maintainability issues exponentially.

JupyterLab could introduce an automatic clone counter, alerting users of the accumulating technical debt and recommending them to create modules to allocate snippets that are duplicated the most into routines. PyCharm offers a similar function, where it detects and alerts the user of duplicated lines of code. A function like this will permit the user to encapsulate snippets into routines which could be called directly from notebooks.

Research Implications

I observed that while this type of duplication was present in my sample, users struggled to easily make use of previous cells, even when these cells solved exactly the same problem. This suggests that merely providing a mechanism to duplicate code has to overcome barriers of ease of use. It seems to be simpler, for some cases, to copy and paste from online sources (like Stack Overflow) than to do the same thing from one's own work (reinventing the wheel attitude).

5.3.2 Implications for VCS with Jupyter Notebooks

Practice Implications

Based on my limited study, version control of notebooks is less important for future code reuse than for archival purposes or collaboration. For single-user notebooks, in particular, complex tools for version control and diff might be replaced with simpler save and restore functionality like in backup interfaces (such as Apple's Time Machine model). In part, this is already supported in JupyterLab, using its checkpoint function, which makes a checkpoint of the notebook at regular intervals.

Research Implications

Evidence from other notebook studies [3, 33, 54], and my observations in this thesis show that code duplication and reuse using *version control history* is a challenge. However, *git* was designed for software development on the Linux operating system,

and evidence suggests the code reuse scenario is low on the list of reasons developers use version control. As Codoban *et al.* reported, software developers instead use software history for debugging, program understanding, and collaboration [56]. This use case is different than searching for previous solutions and may explain why version control tools like *git* are a bad fit for exploratory programming. Future research should devise new tools to bridge this gap by studying new interfaces to skim through the commit tree more easily, rendering notebooks as they appear, similar to the work done by Kery *et al.* in [57], where they studied methods for effective foraging by data scientist in order to find past choices.

5.3.3 Implications for External Reuse

Practice Implications

Social media and external (web) sources are widely recognized as a vital part of modern programming [58]. Programming support in notebooks should recognize this and support it, possibly by offering Stack Overflow recommendations from within JupyterLab based on the code being typed in a cell. Ponzanelli *et al.* had studied this before, where they devised two Elipse plug-ins called SEAHAWK [59] and PROMPTER [60] to offer within IDE Stack Overflow discussions based on the code being typed inside the IDE.

5.3.4 Implications for Internal Reuse

Practice Implications

Indexing and searching tools might provide a way to quickly traverse previously used notebooks in search of routines and functions, which could reduce substantially the time and effort needed for this, and in return fostering reuse from internal sources. Better ways to name and organize notebooks could improve reusability of notebooks in one's own codebase. Designing new ways to support this process could enhance and foster reusability without compromising other software engineering best practices.

Chapter 6

Conclusions & Future Work

6.1 Summary of Research

In this chapter I would like to summarize my results by research question and end this document with a final remark.

RQ1: How much cell code duplication occurs in Jupyter notebooks? And what is the main programming goal of these duplicates?
According to my results, approximately one in thirteen code cells in Jupyter notebooks contain a duplicate of other cell in the repository. The type of snippet that gets duplicated the most, are the ones involving some form of visualization of data, followed by snippets performing machine learning techniques. These two types of snippets comprise approximately 36% of all duplicated snippets within Jupyter notebooks.

RQ2: How does cell code reuse happen in Jupyter notebooks?
The preferred method of reuse in Jupyter notebooks is by copying and pasting code, with copy by typing of code being also a popular way to reuse. The forms of reuse that were used the least was via *git checkout* or *git revert*, and by duplicating an entire notebook. My studies also suggest that more work needs to be done in order to promote and highlight current tools and commands in Jupyter, like duplication of

notebooks, copying and pasting of entire cells and others.

RQ3: What are the preferred sources for code reuse in Jupyter notebooks? My results indicate that the main source of code reuse in Jupyter notebooks is by online browsing, in other words, most code comes from web sites and online forums. And the least effective source of reuse are version control systems like *git*. Also, surprisingly to me, reusing from old owned notebooks didn't appear to be as popular as I thought at the beginning of my studies.

6.2 Final Remarks

I examined how code duplication and reuse happens in Jupyter notebooks. My first study looked at how much self-duplication (e.g., within the repository) exists. I discovered that on average 7.6% of code in repositories is self-duplicated. However, this did not explain how or from where code was duplicated to begin with.

I therefore conducted a lab study with eight participants and deliberately crafted tasks designed to encourage reuse behaviour. I observed how participants reused code to solve data science tasks, and how they leveraged version control, online sources and other notebooks. Reusing code from online sources proved to be the preferred method of reuse for my participants, with 18% of their time spent browsing for code examples online, and version control systems proved to be the least effective method of reuse. Snippets of code that visualize data are the ones that are duplicated the most.

I conclude this thesis by discussing observations and implications from my studies. First, while code duplication is clearly common in notebooks, the source of that duplication is important. Second, although much attention focuses on version control, for code reuse, other sources, such as API examples, are more important. Finally, these external sources are used for various tasks. Notebook interfaces should support modularization and reuse to improve cognitive support for data scientists.

6.3 Future Work

The studies I conducted for this thesis investigated code duplication and reuse in Jupyter notebooks at a broader scale. It is recommended, that if we are really interested to determine the most effective and efficient method of reuse in Jupyter notebooks, then several studies should follow this one. For example:

- A study that measures the exact time users spend using each source of reuse, would show with much more accuracy which information source is easier to attain and under which circumstances. This can be done by logging to file, events performed by users while using notebooks in real settings instead of a controlled lab experiment. As a complement, future studies could observe users while conducting their work using notebooks with their own devices and while solving real problems. It would be best if observations could be made as much for academic environments as for industrial ones, observing everyday problems users of this platform have.

- Also, future studies should consider harder and longer tasks for participants to solve, and measure how many lines of code are being reused from the different sources, instead of measuring only if reuse happened or not. This would add more granularity to the results presented in this thesis. Also, due to the simplicity of some tasks in my study, I was not able to observe particular cases of complex routines' reuse, which may show reuse from other sources not shown here in my studies.

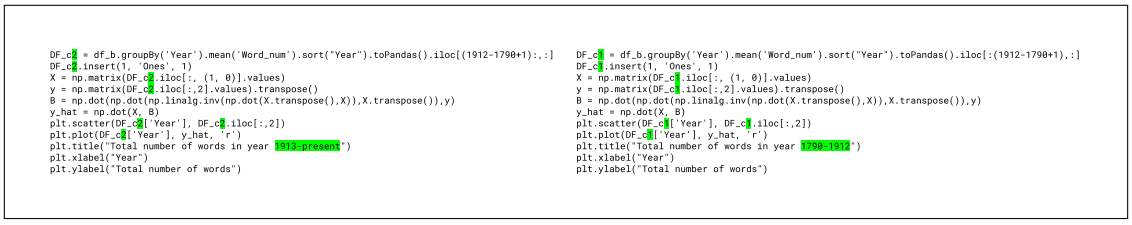
All these questions and other possible research paths would need to be conducted before we could better understand reuse behaviour in Jupyter notebooks. Doing so will entice better tools to be build for code reuse, allowing developers and analysts to speed up their exploration sessions, while keeping up with software engineering best practices and extracting the full potential of this new medium of computation.

Appendices

Appendix A

Examples of Duplicated Snippets

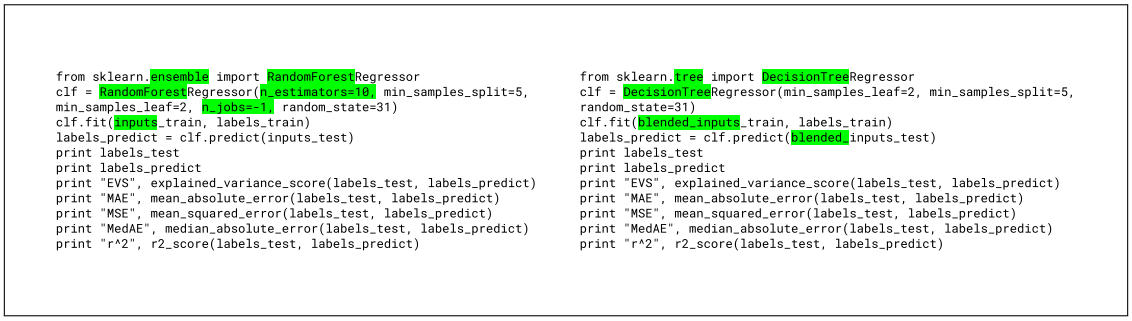
The following figures are examples of duplicates that Duplicate Ratio Function 1 detected. They were computed using different cut-off value thresholds for the purpose finding the optimal cut-off value to use as a detection parameter (See Section 3.5.1).



```
DF_c1 = df_b.groupBy('Year').mean('Word_num').sort("Year").toPandas().iloc[(1912-1790+1):,:]
DF_c1.insert(1, 'Ones', 1)
X = np.matrix(DF_c1.iloc[:, (1, 0)].values)
y = np.matrix(DF_c1.iloc[:,2].values).transpose()
B = np.dot(np.dot(np.linalg.inv(np.dot(X.transpose(),X)),X.transpose()),y)
y_hat = np.dot(X, B)
plt.scatter(DF_c1['Year'], DF_c1.iloc[:,2])
plt.plot(DF_c1['Year'], y_hat, 'r')
plt.title("Total number of words in year 1913-present")
plt.xlabel("Year")
plt.ylabel("Total number of words")
```

```
DF_c1 = df_b.groupBy('Year').mean('Word_num').sort("Year").toPandas().iloc[:(1912-1790+1),:]
DF_c1.insert(1, 'Ones', 1)
X = np.matrix(DF_c1.iloc[:, (1, 0)].values)
y = np.matrix(DF_c1.iloc[:,2].values).transpose()
B = np.dot(np.dot(np.linalg.inv(np.dot(X.transpose(),X)),X.transpose()),y)
y_hat = np.dot(X, B)
plt.scatter(DF_c1['Year'], DF_c1.iloc[:,2])
plt.plot(DF_c1['Year'], y_hat, 'r')
plt.title("Total number of words in year 1790-1912")
plt.xlabel("Year")
plt.ylabel("Total number of words")
```

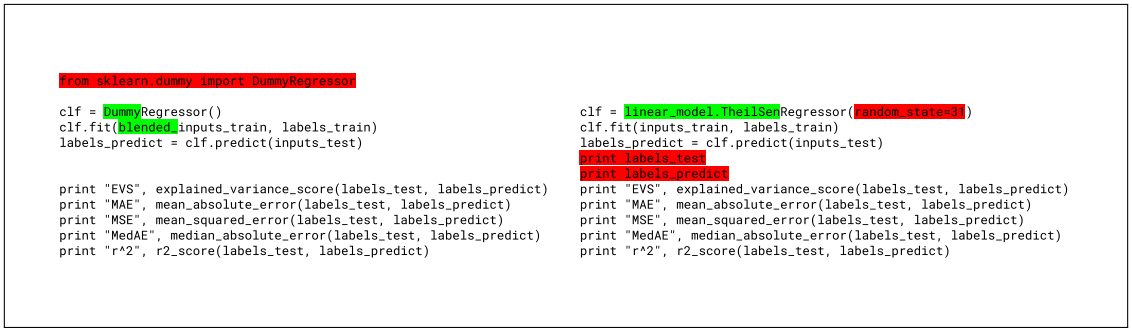
Figure A.1: This image shows two snippets of code marked as clones by my Duplicate Ratio Function 1. Threshold 0.0-0.1. This particular snippet has a *Duplicate Ratio* (DR) value of 0.04.



```
from sklearn.ensemble import RandomForestRegressor
clf = RandomForestRegressor(n_estimators=10, min_samples_split=5,
min_samples_leaf=2, n_jobs=-1, random_state=31)
clf.fit(inputs_train, labels_train)
labels_predict = clf.predict(inputs_test)
print labels_test
print labels_predict
print "EVS", explained_variance_score(labels_test, labels_predict)
print "MAE", mean_absolute_error(labels_test, labels_predict)
print "MSE", mean_squared_error(labels_test, labels_predict)
print "MedAE", median_absolute_error(labels_test, labels_predict)
print "r^2", r2_score(labels_test, labels_predict)
```

```
from sklearn.tree import DecisionTreeRegressor
clf = DecisionTreeRegressor(min_samples_leaf=2, min_samples_split=5,
random_state=31)
clf.fit(blended_inputs_train, labels_train)
labels_predict = clf.predict(blended_inputs_test)
print labels_test
print labels_predict
print "EVS", explained_variance_score(labels_test, labels_predict)
print "MAE", mean_absolute_error(labels_test, labels_predict)
print "MSE", mean_squared_error(labels_test, labels_predict)
print "MedAE", median_absolute_error(labels_test, labels_predict)
print "r^2", r2_score(labels_test, labels_predict)
```

Figure A.2: This image shows two snippets of code marked as clones by my Duplicate Ratio Function 1. Threshold 0.1-0.2. This particular snippet has a *Duplicate Ratio* (DR) value of 0.18.



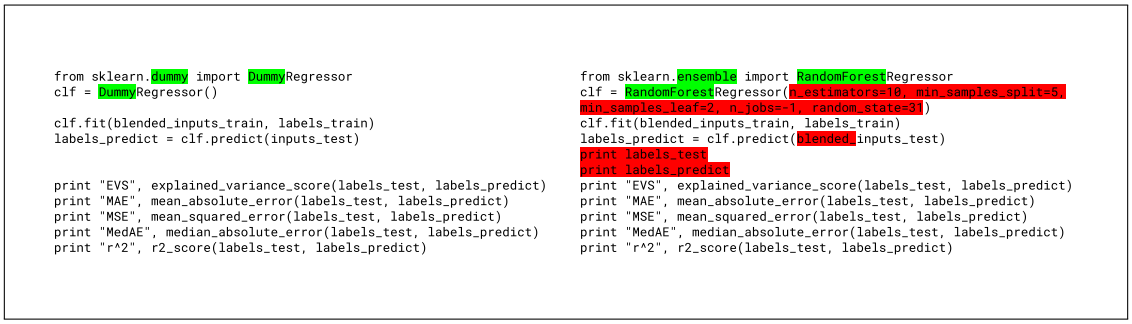
```
from sklearn.dummy import DummyRegressor

clf = DummyRegressor()
clf.fit(blended_inputs_train, labels_train)
labels_predict = clf.predict(inputs_test)


print "EVS", explained_variance_score(labels_test, labels_predict)
print "MAE", mean_absolute_error(labels_test, labels_predict)
print "MSE", mean_squared_error(labels_test, labels_predict)
print "MedAE", median_absolute_error(labels_test, labels_predict)
print "r^2", r2_score(labels_test, labels_predict)
```

```
clf = linear_model.TheilSenRegressor(random_state=31)
clf.fit(inputs_train, labels_train)
labels_predict = clf.predict(inputs_test)
print labels_test
print labels_predict
print "EVS", explained_variance_score(labels_test, labels_predict)
print "MAE", mean_absolute_error(labels_test, labels_predict)
print "MSE", mean_squared_error(labels_test, labels_predict)
print "MedAE", median_absolute_error(labels_test, labels_predict)
print "r^2", r2_score(labels_test, labels_predict)
```

Figure A.3: This image shows two snippets of code marked as clones by my Duplicate Ratio Function 1. Threshold 0.2-0.3. This particular snippet has a *Duplicate Ratio* (DR) value of 0.25.



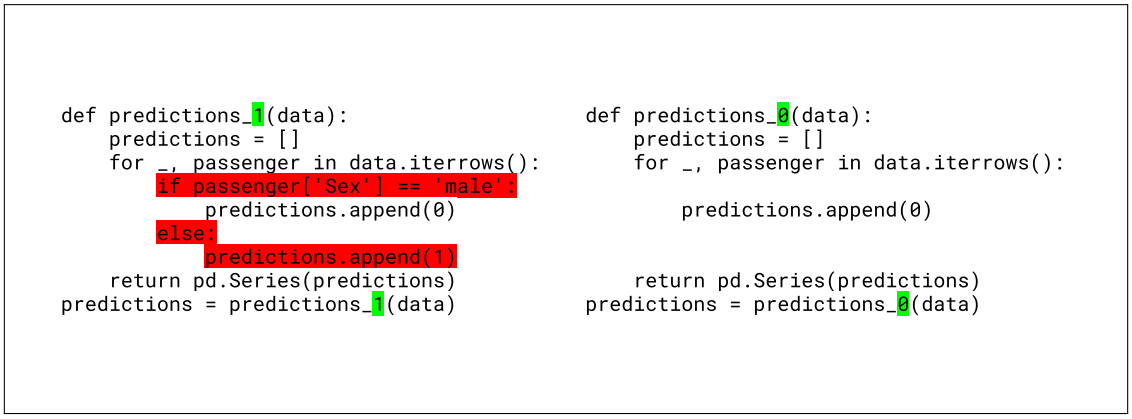
```
from sklearn.dummy import DummyRegressor
clf = DummyRegressor()

clf.fit(blended_inputs_train, labels_train)
labels_predict = clf.predict(inputs_test)


print "EVS", explained_variance_score(labels_test, labels_predict)
print "MAE", mean_absolute_error(labels_test, labels_predict)
print "MSE", mean_squared_error(labels_test, labels_predict)
print "MedAE", median_absolute_error(labels_test, labels_predict)
print "r^2", r2_score(labels_test, labels_predict)
```

```
from sklearn.ensemble import RandomForestRegressor
clf = RandomForestRegressor(n_estimators=10, min_samples_split=5,
min_samples_leaf=2, n_jobs=-1, random_state=31)
clf.fit(blended_inputs_train, labels_train)
labels_predict = clf.predict(blended_inputs_test)
print labels_test
print labels_predict
print "EVS", explained_variance_score(labels_test, labels_predict)
print "MAE", mean_absolute_error(labels_test, labels_predict)
print "MSE", mean_squared_error(labels_test, labels_predict)
print "MedAE", median_absolute_error(labels_test, labels_predict)
print "r^2", r2_score(labels_test, labels_predict)
```

Figure A.4: This image shows two snippets of code marked as clones by my Duplicate Ratio Function 1. Threshold 0.3-0.4. This particular snippet has a *Duplicate Ratio* (DR) value of 0.39.



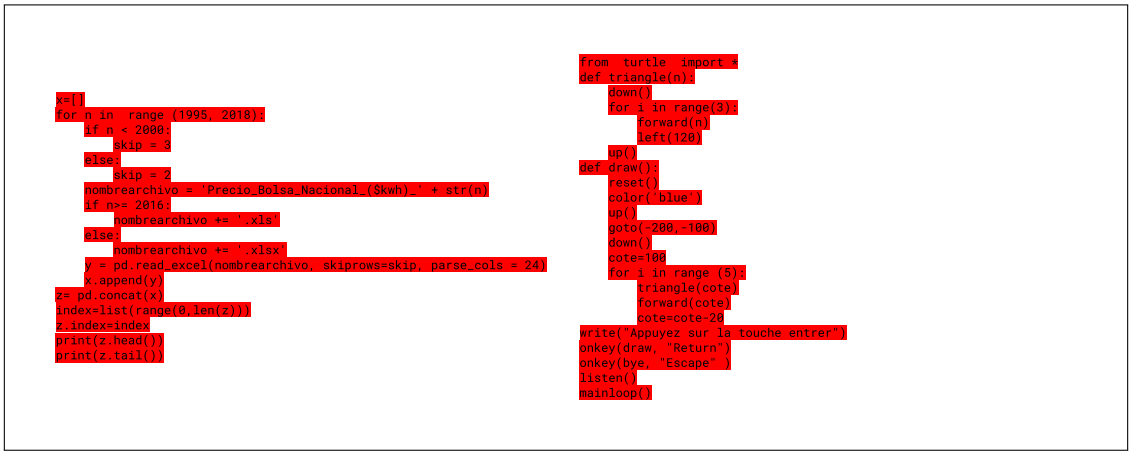
```
def predictions_1(data):
    predictions = []
    for _, passenger in data.iterrows():
        if passenger['Sex'] == 'male':
            predictions.append(0)
        else:
            predictions.append(1)
    return pd.Series(predictions)
predictions = predictions_1(data)
```

```
def predictions_0(data):
    predictions = []
    for _, passenger in data.iterrows():

        predictions.append(0)


    return pd.Series(predictions)
predictions = predictions_0(data)
```

Figure A.5: This image shows two snippets of code marked as clones by my Duplicate Ratio Function 1. Threshold 0.5-0.6. This particular snippet has a *Duplicate Ratio* (DR) value of 0.53.



```
x=[]
for n in  range (1995, 2018):
    if n < 2000:
        skip = 3
    else:
        skip = 2
    nombrearchivo = 'Precio_Bolsa_Nacional_($kwh)_' + str(n)
    if n>= 2016:
        nombrearchivo += '.xls'
    else:
        nombrearchivo += '.xlsx'
    y = pd.read_excel(nombrearchivo, skiprows=skip, parse_cols = 24)
    x.append(y)
z= pd.concat(x)
index=list(range(0,len(z)))
z.index=index
print(z.head())
print(z.tail())
```

```
from  turtle  import *
def triangle(n):
    down()
    for i in range(3):
        forward(n)
        left(120)
    up()
def draw():
    reset()
    color('blue')
    up()
    goto(-200,-100)
    down()
    cote=100
    for i in range (5):
        triangle(cote)
        forward(cote)
        cote=cote-20
write("Appuyez sur la touche entrer")
onkey(draw, "Return")
onkey(bye, "Escape" )
listen()
mainloop()
```

Figure A.6: This image shows two snippets of code marked as clones by my Duplicate Ratio Function 1. Threshold 0.8-0.9. This particular snippet has a *Duplicate Ratio* (DR) value of 0.88.

Appendix B

Observational Study Tasks

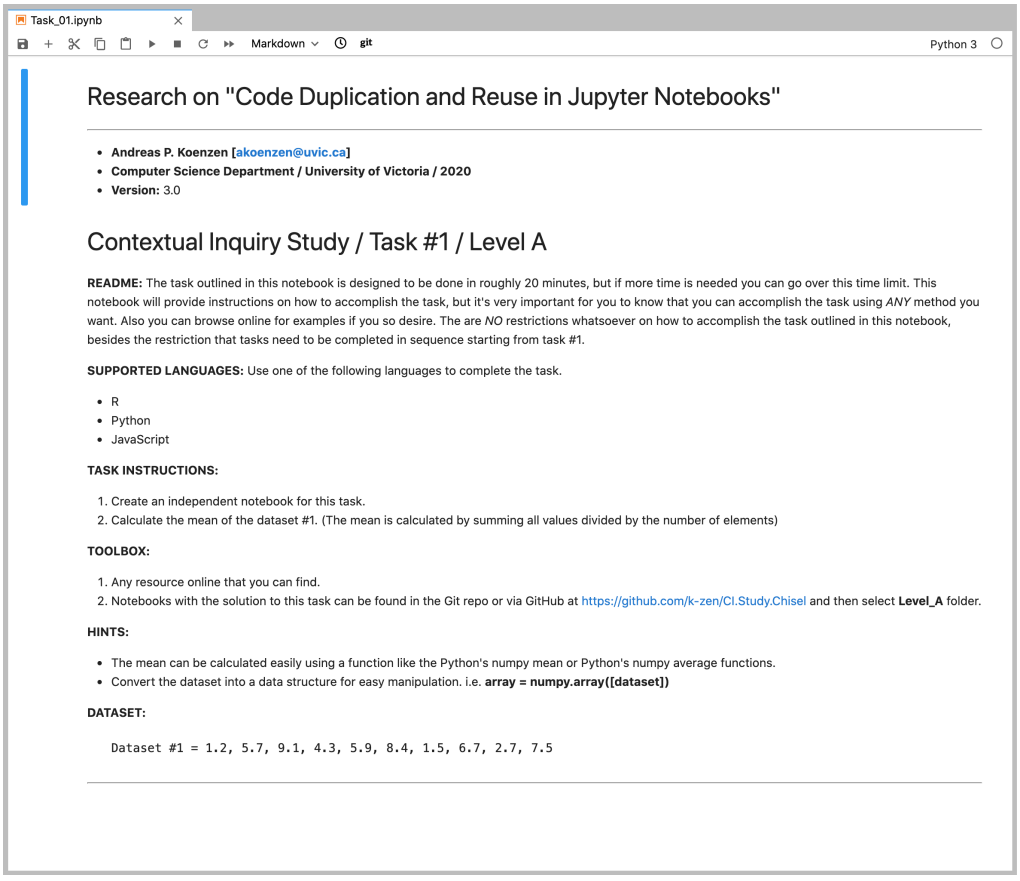


Figure B.1: Jupyter notebook describing what the participant had to do during the observational study for Task #1 (Level A).

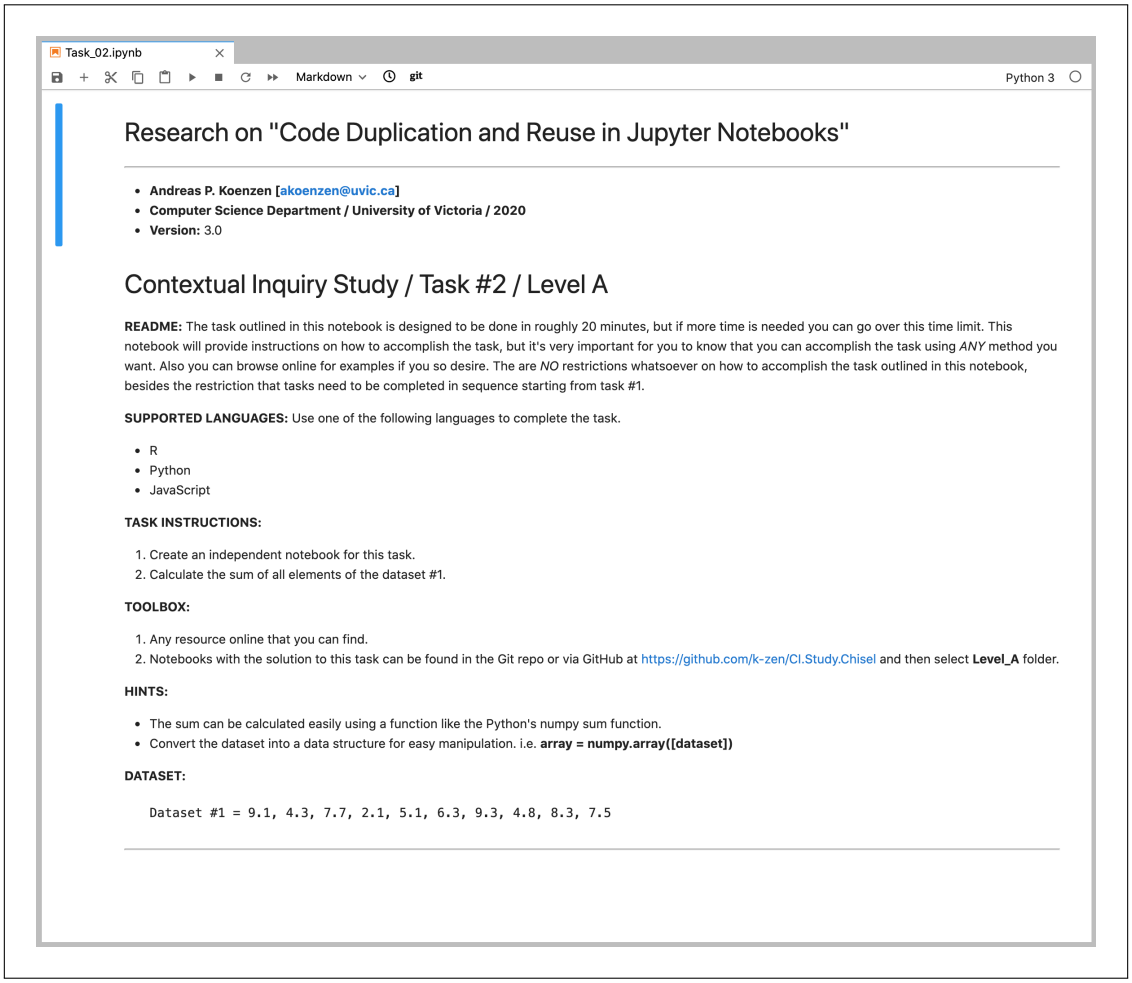


Figure B.2: Jupyter notebook describing what the participant had to do during the observational study for Task #2 (Level A).

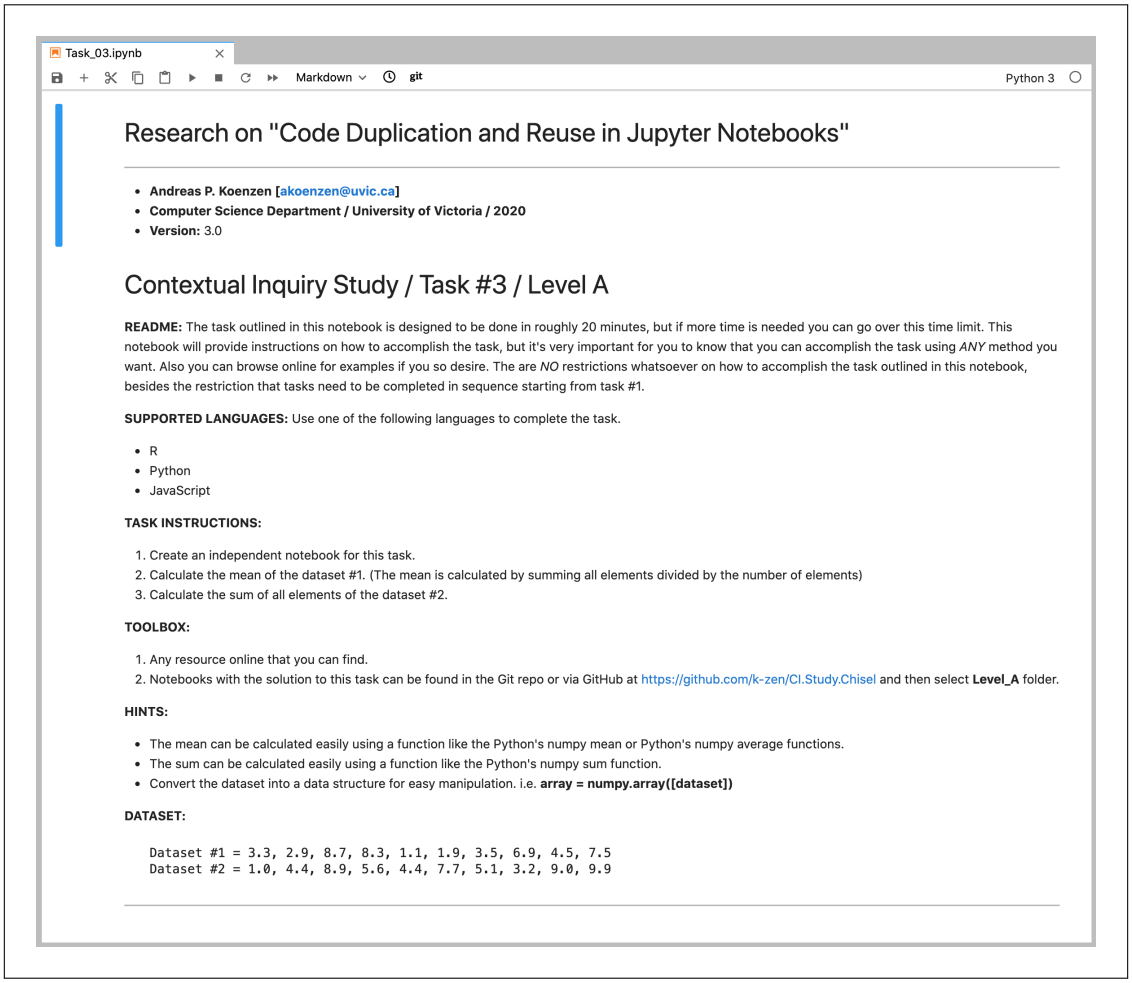


Figure B.3: Jupyter notebook describing what the participant had to do during the observational study for Task #3 (Level A).

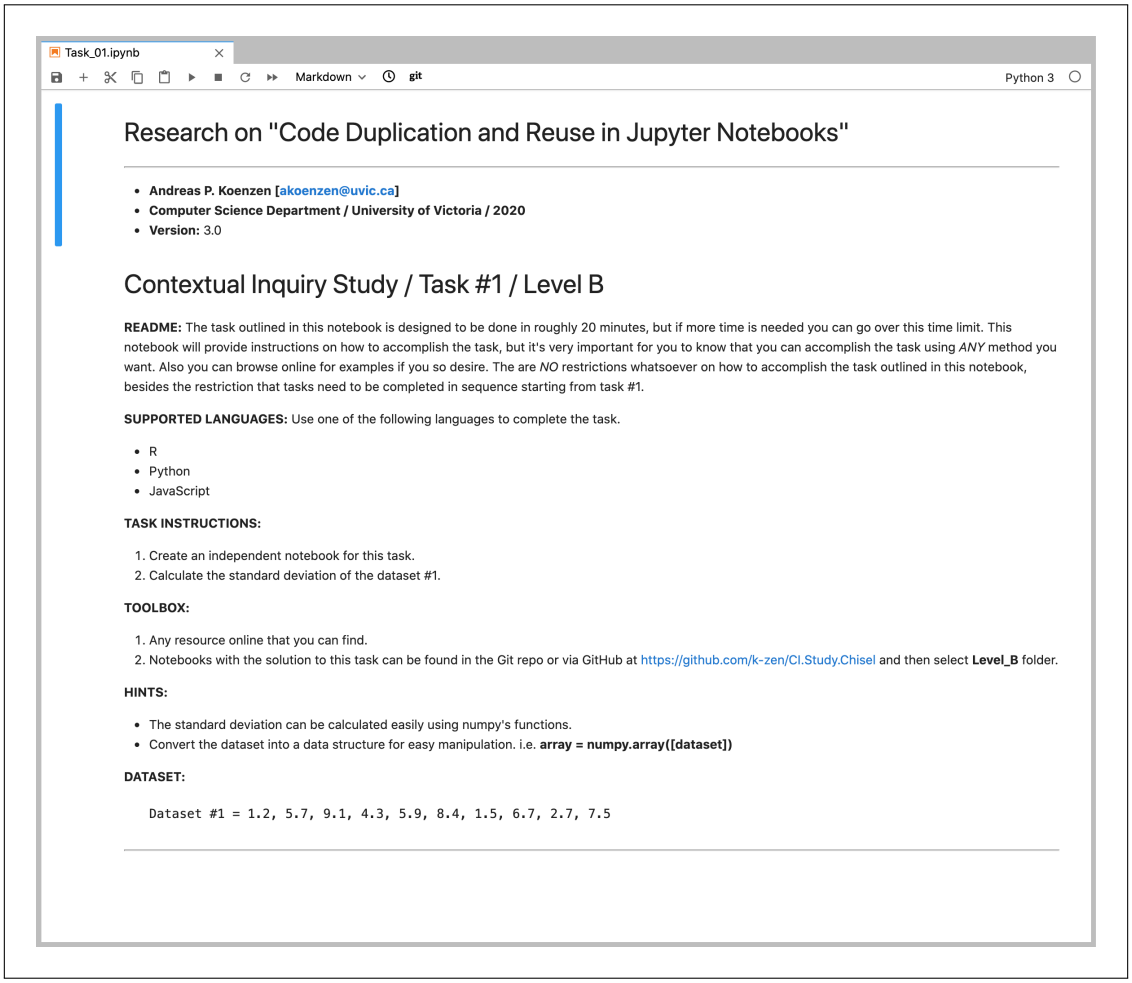


Figure B.4: Jupyter notebook describing what the participant had to do during the observational study for Task #1 (Level B).

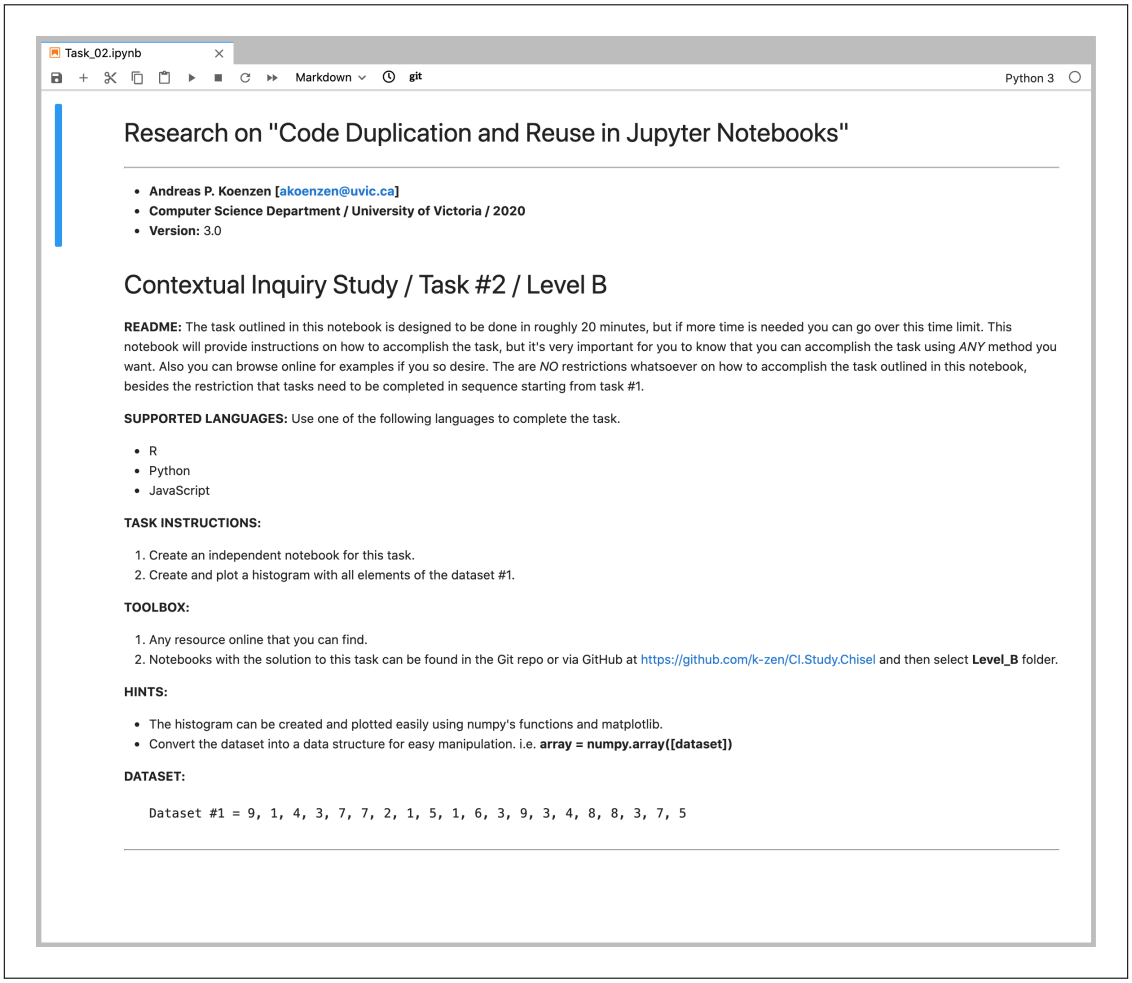


Figure B.5: Jupyter notebook describing what the participant had to do during the observational study for Task #2 (Level B).

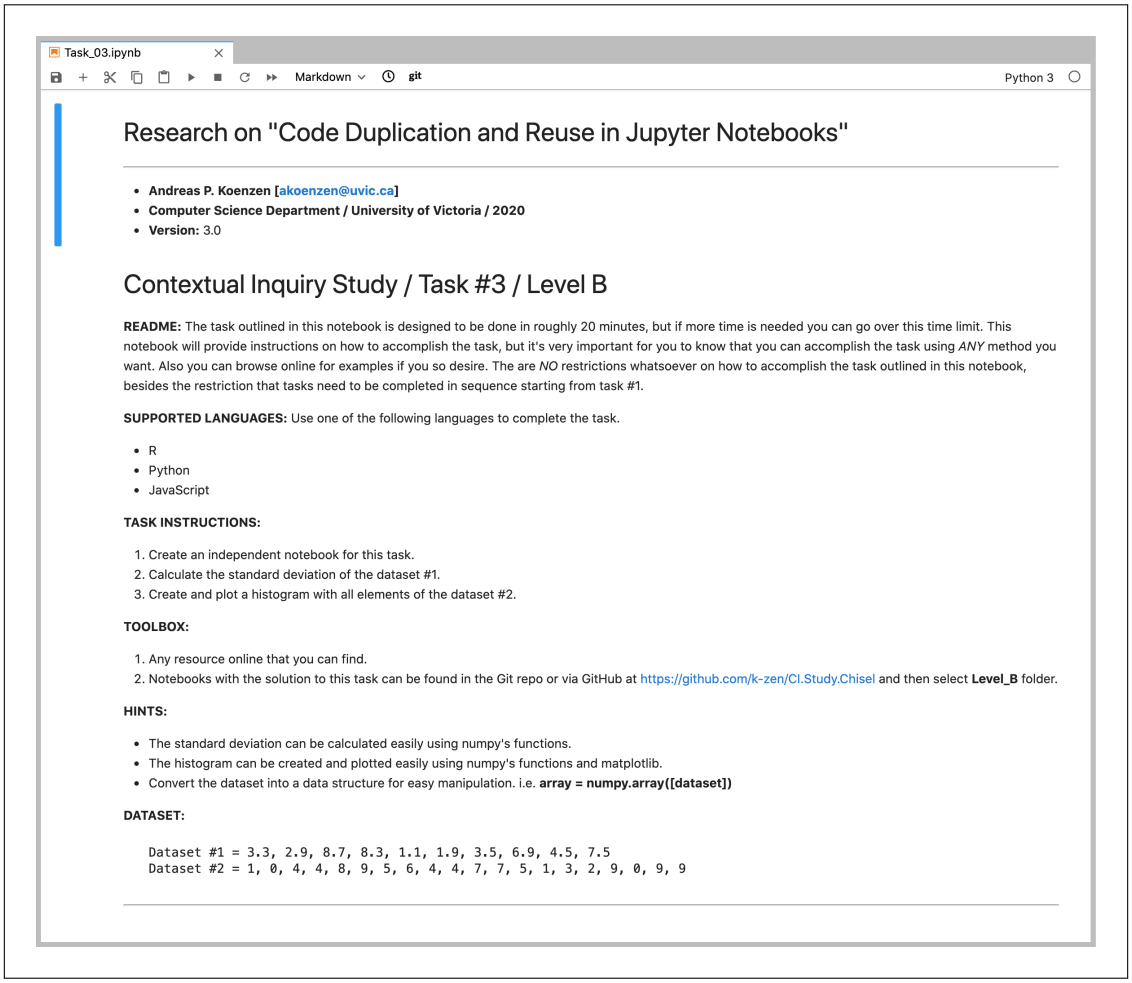


Figure B.6: Jupyter notebook describing what the participant had to do during the observational study for Task #3 (Level B).

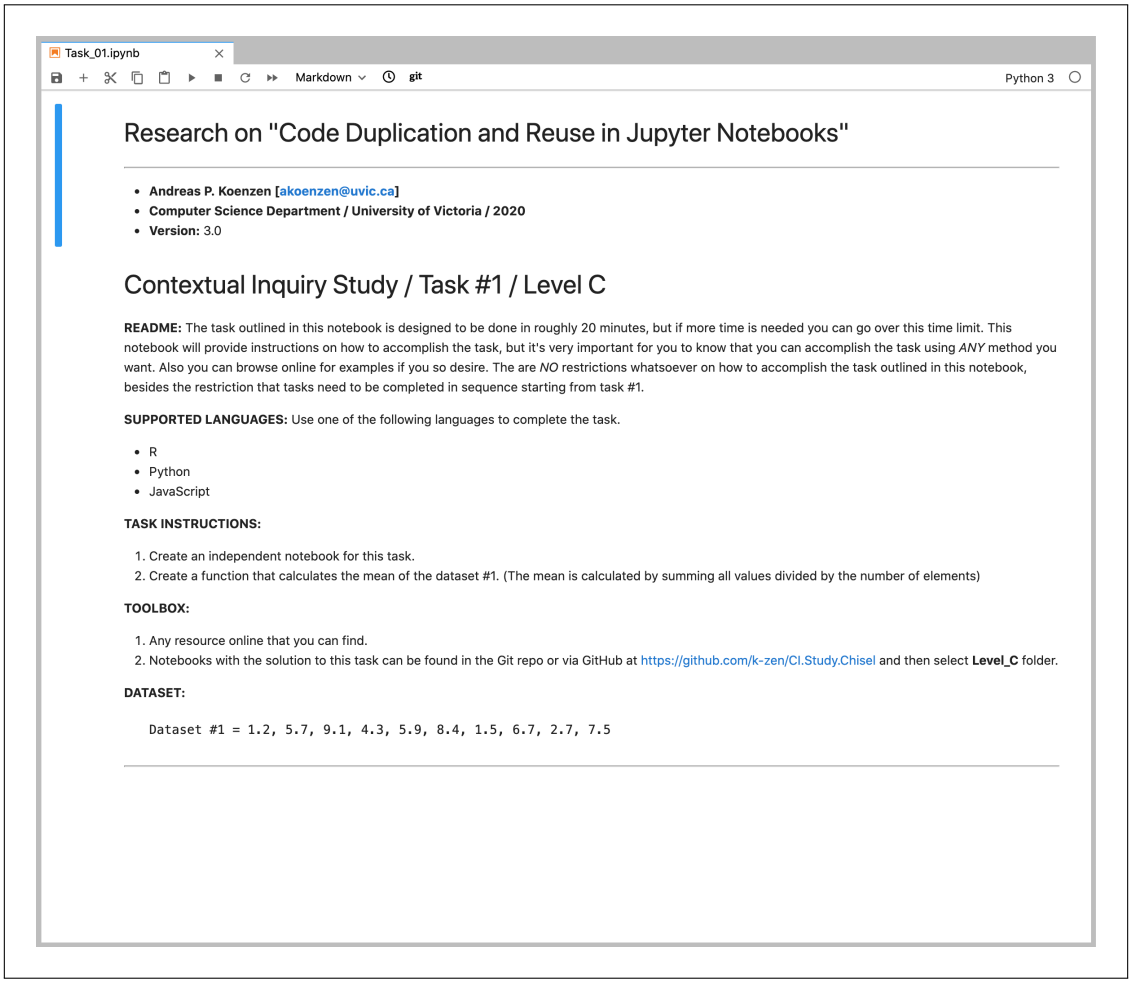


Figure B.7: Jupyter notebook describing what the participant had to do during the observational study for Task #1 (Level C).

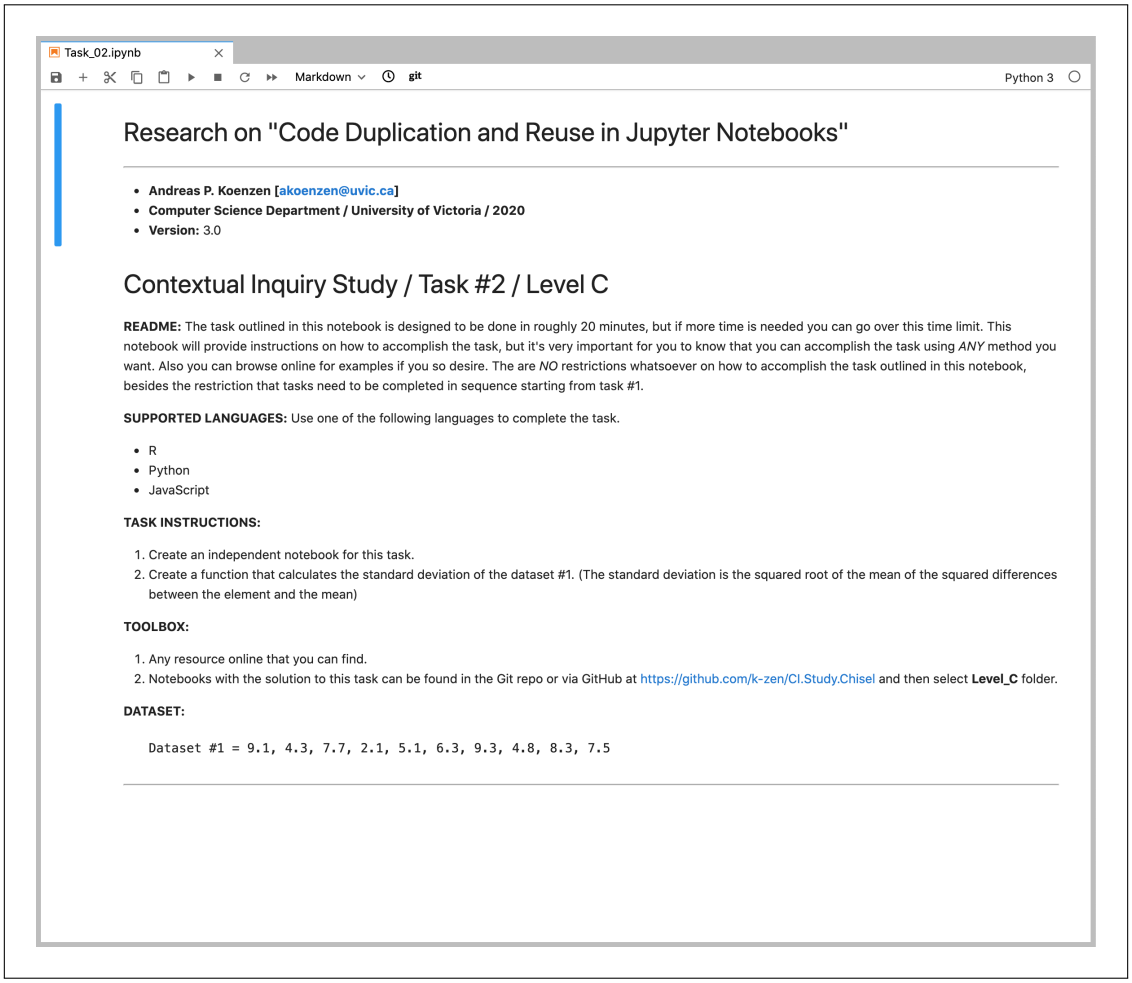


Figure B.8: Jupyter notebook describing what the participant had to do during the observational study for Task #2 (Level C).

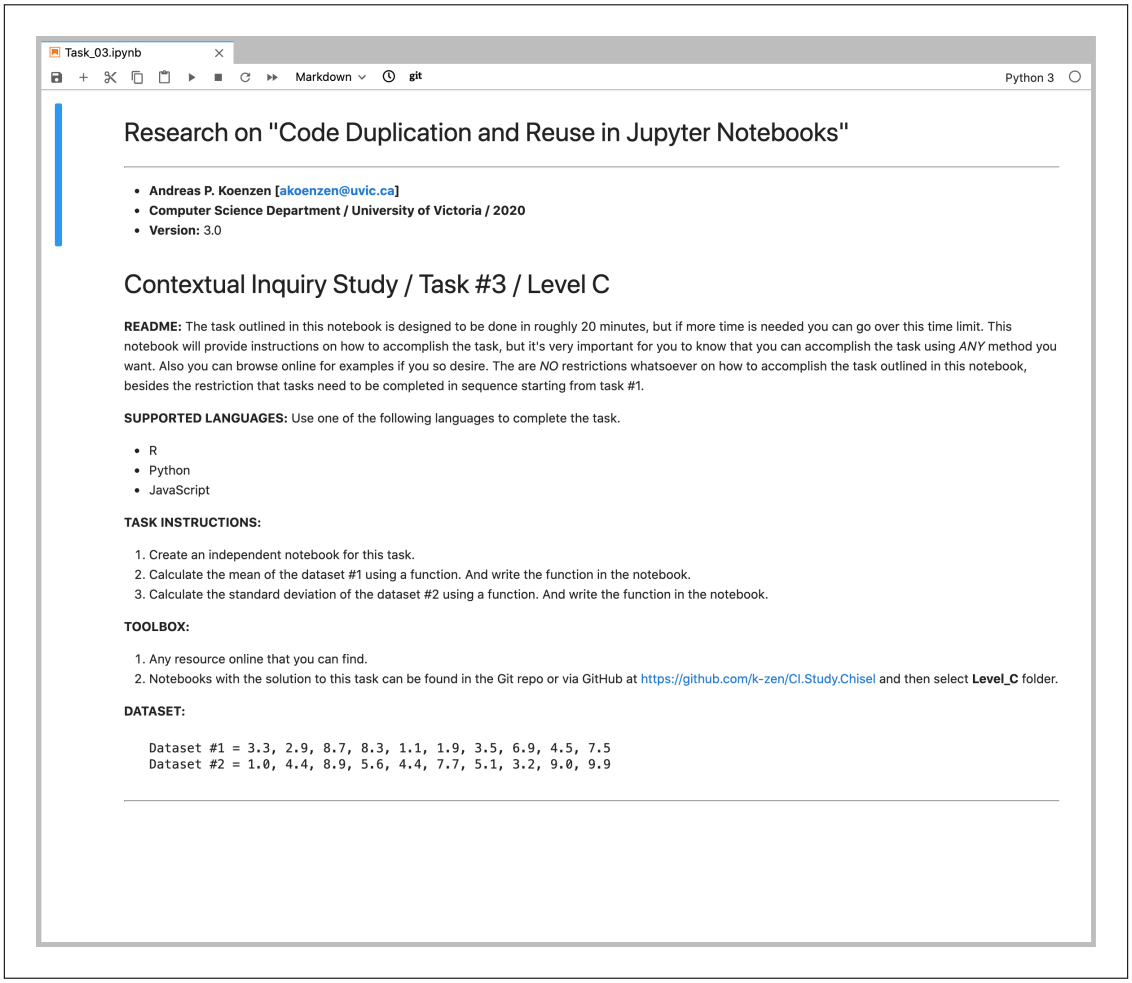


Figure B.9: Jupyter notebook describing what the participant had to do during the observational study for Task #3 (Level C).

Appendix C

Observational Study Questionnaire

C.1 Background

1. What is your occupation?

   ☐ Academia (Professor, researcher)

   ☐ Student

   ☐ Industry

   ☐ Other: _____

2. What is your domain?

   ☐ Computer Science

   ☐ Life Science

   ☐ Mathematics

   ☐ Other: _____

C.2 Questions about Experience

3. How would you rank your proficiency with computational notebooks?

   ☐ None

   ☐ Novice (basic knowledge and usage)

   ☐ Intermediate (being able to setup the notebook's environment)

   ☐ Advanced (installation of extensions and widgets, extending the notebook)

4. How long have you been using computational notebooks?

   ☐ Less than one year

   ☐ Between one and three years

   ☐ More than three years

5. How much experience do you have using *git*?

   ☐ None

   ☐ Novice (basic commands, pull, push commit)

   ☐ Intermediate (merging, solving conflicts, branching)

   ☐ Advanced (rebasing, filtering, resetting, tagging)

6. (If applicable) How long have you been using *git*?

   ☐ Less than one year

   ☐ Between one and three years

   ☐ More than three years

7. Which tool/process do you normally use for versioning your computational notebooks, and how would you rate your experience with each of these tools/processes?

   _____ Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

   _____ Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

   _____ Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

8. Which computational notebook software do you normally use, and how would you rate your experience with each of these tools?

   _____ Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

   _____ Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

   _____ Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

C.3 Questions about Computational Notebooks

10. Computational notebooks are simple to use.

    Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree

11. When working with computational notebooks, would you say you work alone or in a team?

    Alone | Team

12. What would you say is the main task for which you use computational notebooks?

    _____

    _____

    _____

C.4 Questions about Version Control

13. Would you say that you normally revert to previous versions of notebooks?

    Yes | No

C.5 (If applicable) Questions about *git*

14. *git* can be used effectively to version computational notebooks.

    Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree

15. Traversing previous versions of notebooks are easy with *git*.

    Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree

16. Have you used any following interfaces for interacting with *git* while working on Jupyter? (Check all that apply)

    ☐ Command Line
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

    ☐ GitHub
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

    ☐ Git Desktop App
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

    ☐ GitKraken
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

    ☐ jupyterlab-git extension
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

    ☐ nbdime extension
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

    ☐ jupyterlab-github extension
    Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

☐ Verdant extension

Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

☐ Other: _____

Very Unsatisfied | Unsatisfied | Neutral | Satisfied | Very Satisfied

Appendix D

Observational Study Interview

D.1 General

Did you ever had any problems managing versions of your notebooks? (If so, which problems)

_____

_____

_____

_____

_____

Appendix E

Observational Study Questionnaire Responses

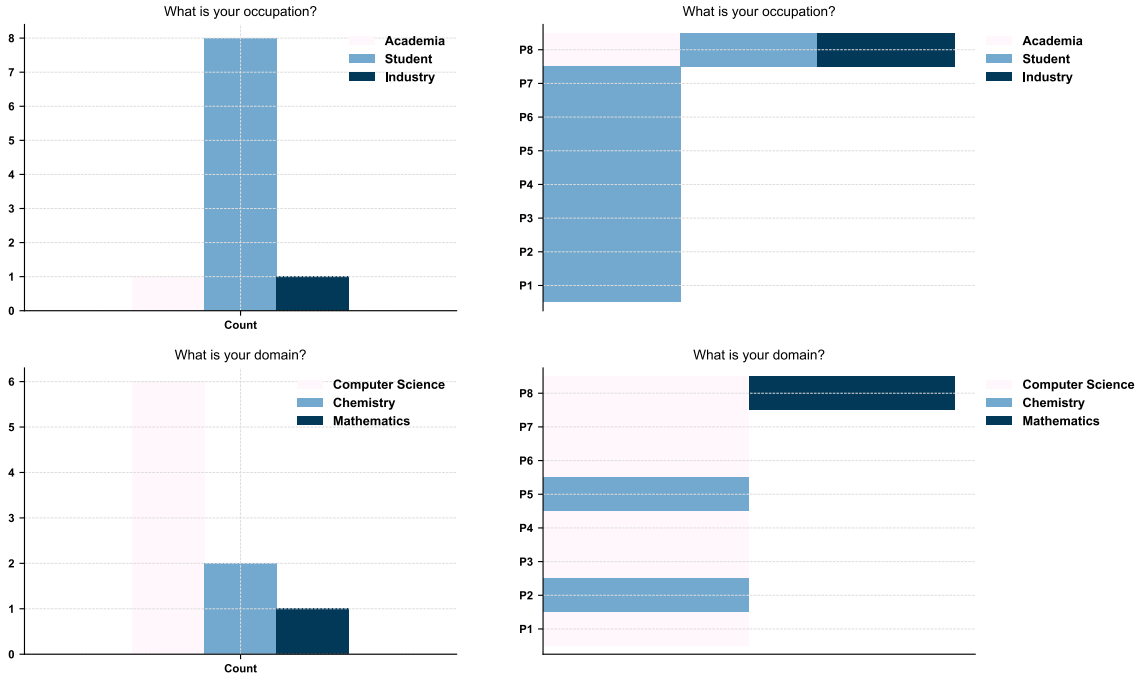


Figure E.1: Coded answers for **questions 1 and 2** of the questionnaire.

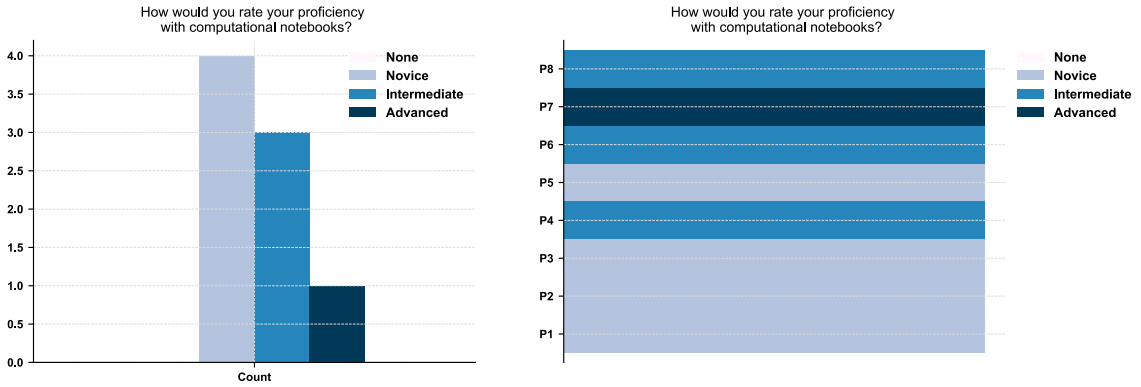


Figure E.2: Coded answers for **question 3** of the questionnaire.

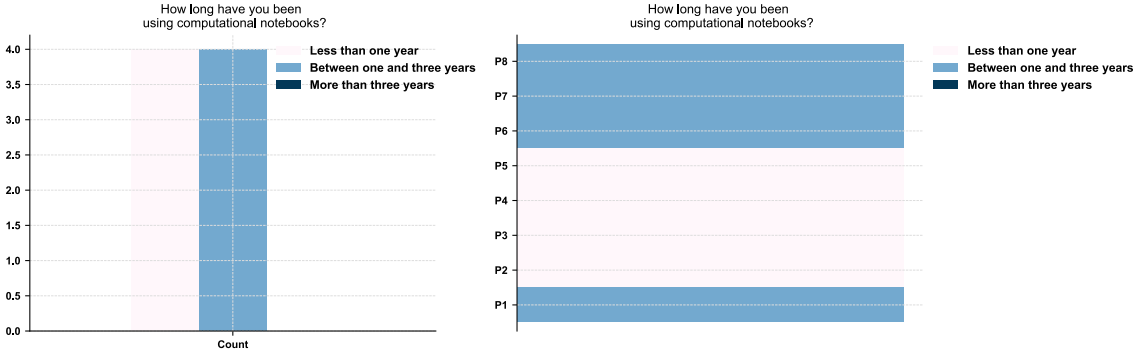


Figure E.3: Coded answers for **question 4** of the questionnaire.

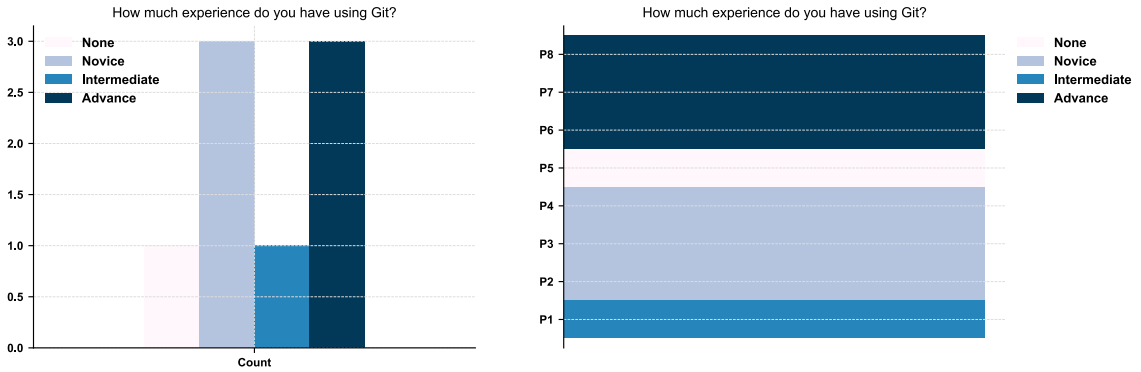


Figure E.4: Coded answers for **question 5** of the questionnaire.

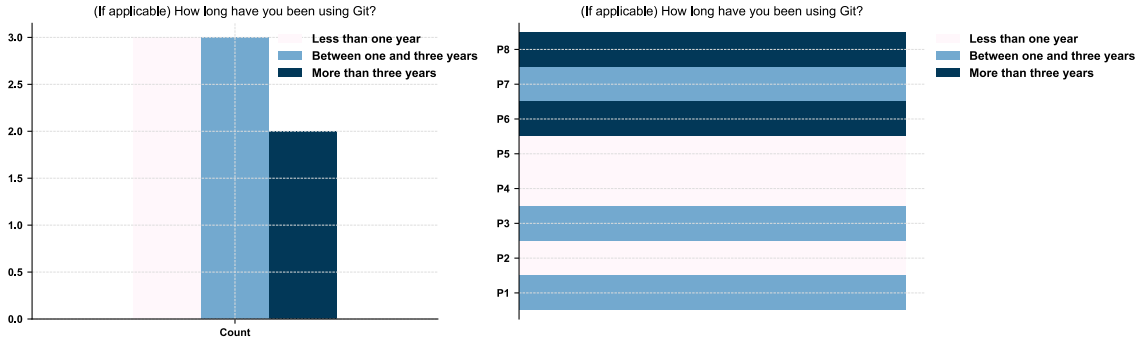


Figure E.5: Coded answers for **question 6** of the questionnaire.

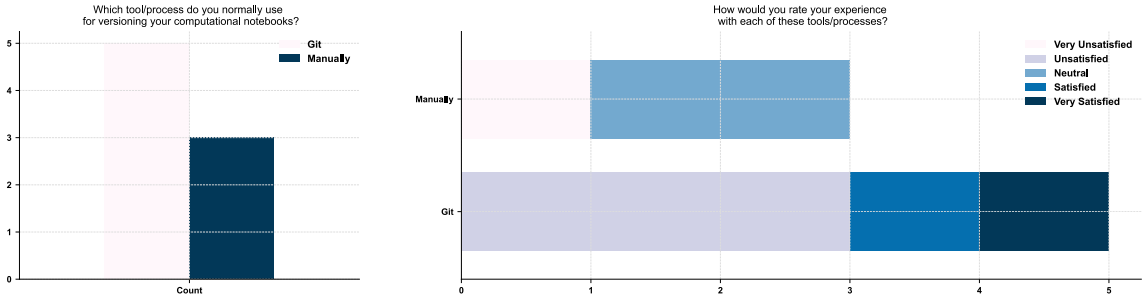


Figure E.6: Coded answers for **question 7** of the questionnaire.

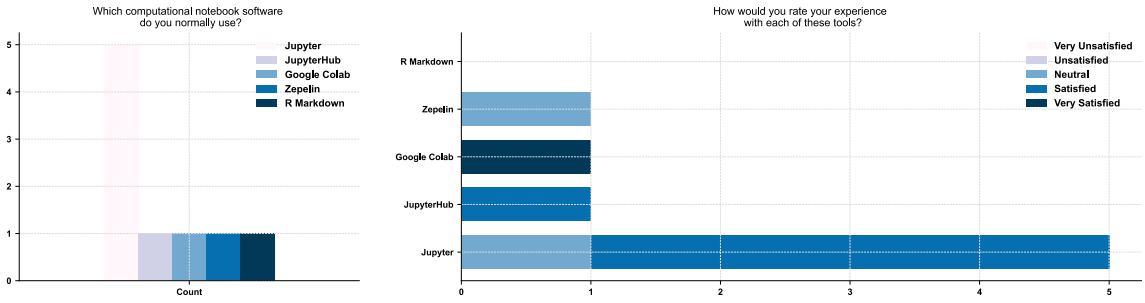


Figure E.7: Coded answers for **question 8** of the questionnaire.

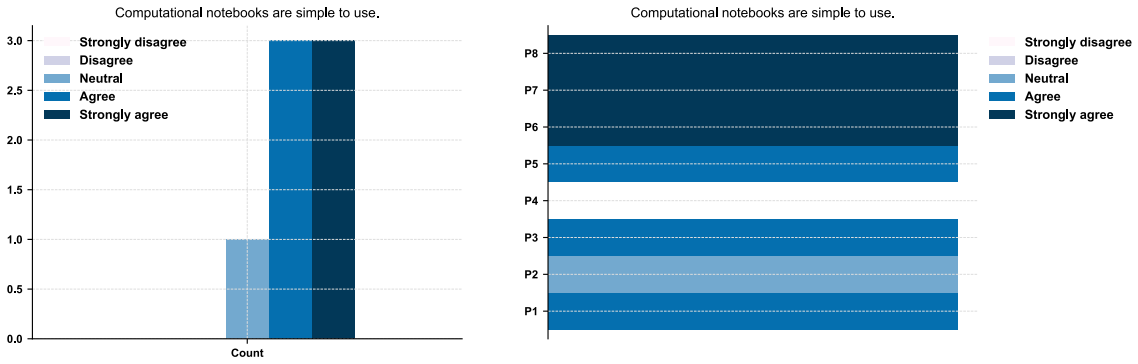


Figure E.8: Coded answers for **question 10** of the questionnaire.

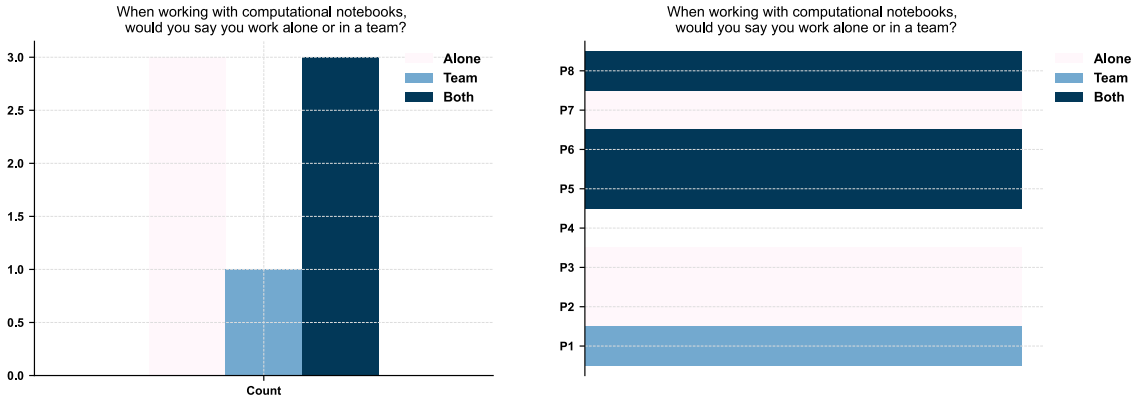


Figure E.9: Coded answers for **question 11** of the questionnaire.

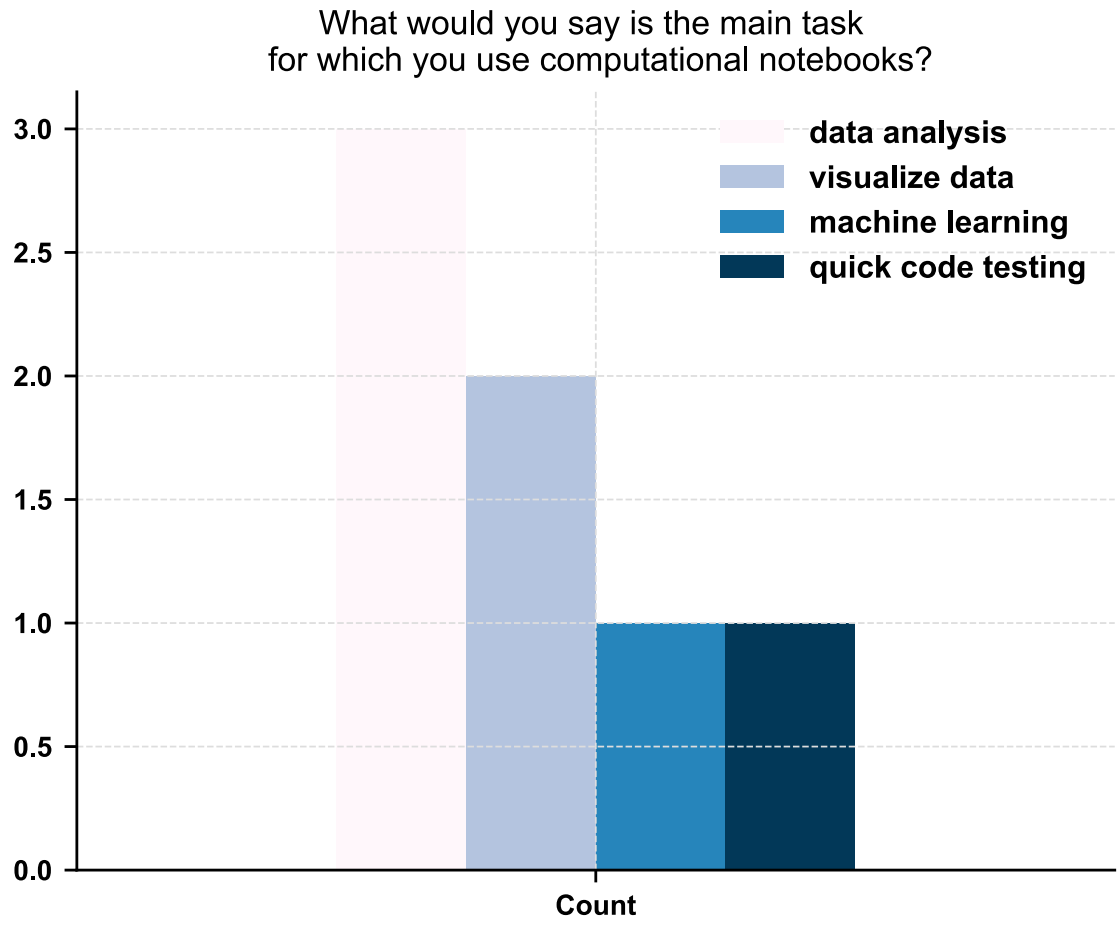


Figure E.10: Coded answers for **question 12** of the questionnaire.

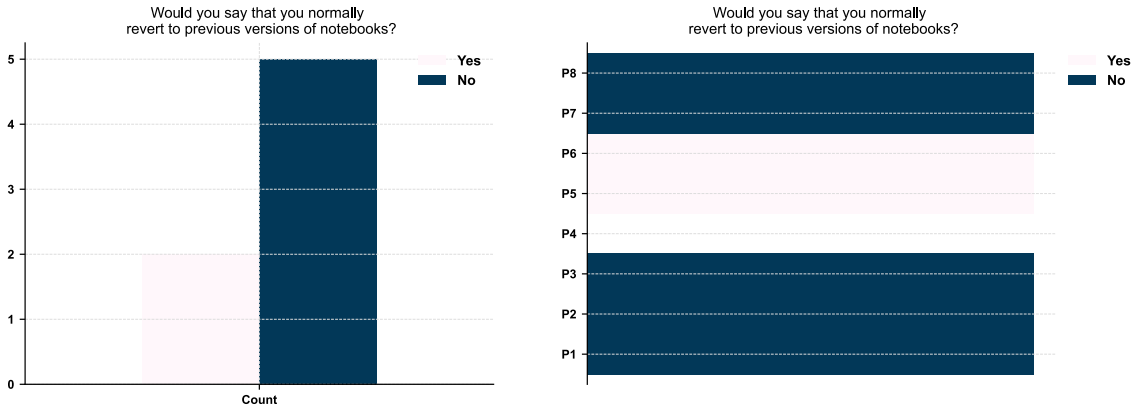


Figure E.11: Coded answers for **question 13** of the questionnaire.

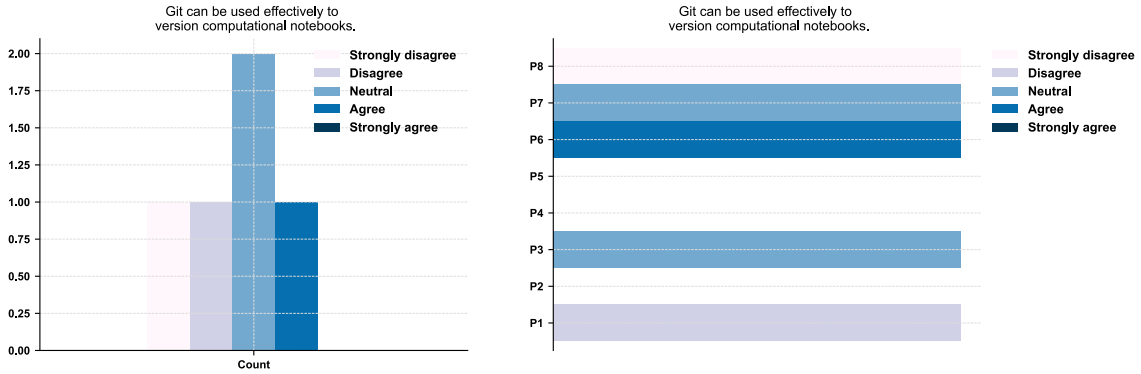


Figure E.12: Coded answers for **question 14** of the questionnaire.

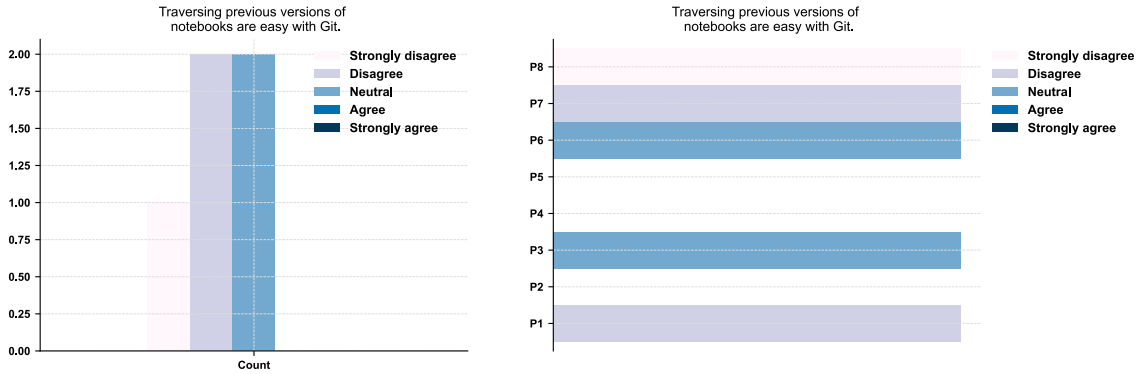


Figure E.13: Coded answers for **question 15** of the questionnaire.

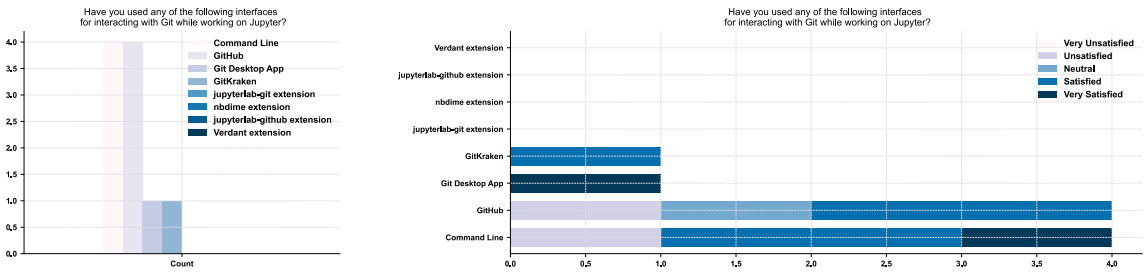


Figure E.14: Coded answers for **question 16** of the questionnaire.

Appendix F

Observational Study Interview Responses

These are the audio-transcribed responses given by the participants of the open-ended supplemental interview which had only one question: **Did you ever had any problems managing versions of your notebooks? If so, which problems.** For which the participants responded the following:

P1: "Yeah so, managing versions, we did that previously with Git and it was challenging because when you store different versions it just make no sense once you uploaded to GitHub, like hum, you can like if you compare versions on GitHub it was just basically a bunch of random text that was next to each other as opposed to normally when you use version control on GitHub it's very clear, ah, I had three lines here or I deleted two lines, but using Jupyter notebooks and Git it was completely just random and then we did some research and we found this kind of a semi-workaround, we you strip out all the output and if you strip out the metadata and then use Git to store your version, it improved it quite a bit, so it wasn't as bad, but I mean ideally if there was a way where let's say you use Git with your Jupyter notebooks when the result is the same as any other type of language like Python or Java then that would be the ideal solution to version control."

P2: "Umm, either I just keep working on just one notebook that has 100 lines or I when I find something that works I take that bit and start a new notebook and

then continue."

P3: "Not applicable. Never managed versions of notebooks before."

P4: "Not applicable. Never managed versions of notebooks before."

P5: "There's just too many of them. Organization is, I don't know if this is a quality of myself or of Jupyter, um but they are kind of all over the place in terms of sectioning and what scripts are appropriate for what folders in my own life. I'm very bad at naming the notebooks, I don't know how much would be included in one notebook vs making a new notebook, you know. Like if I'm graphing something and a lot of my graphs I pull data from a database, so doing that like pre-work I can just going and going if I want, tagging on and on. And then if I make a new notebook it seems like a pain cause I just send like copy and pasting the first chunk of code from all my other notebooks."

P6: "Yeah, I did. So what I generally do is I my most of the time I use Google Colab notebooks, so yeah, Jupyter would be the second one and often times I work on Colab because everything is properly set up inside the Google Colab and the versioning issue how I handle it over there is I duplicate my notebook and rename with the new date or a maybe if I'm doing some machine learning modelling if I'm getting some accuracy for that model then maybe I'll version that like OK, accuracy this, this notebook, accuracy that notebook. That's not good because it doesn't I mean, I don't know I'm kind of satisfied but neutral with that, basically if I'm storing my notebook, if I'm giving some dates to those like if I dated today but if I'm working on maybe 5 to 10 notebooks today, I can not use dates and I should add time. So finding them later can be a problem. If I want to OK day after tomorrow if I sit and I want to look at my previous version then manually just looking at the names of the files and you know seeing those timestamps, I don't find that very much appealing."

P7: "Yeah, the thing is I always have versioned controlled with Git so whenever notebook has like different fails, so even if I don't want a particular kind of, so whenever you run the hole notebook that one differs from the previous one right.

If you have something different in the... say for example if I have some errors or some duplication messages or something like that, it doesn't have to be errors, can be some messages from the Python libraries or something, so those things show in the output column, output cell and now we have a different, even if you had done nothing different in the notebook, you have a different notebook from the previous version."

P8: "Yes. If somebody just changes code, let's say there, you working collaborative with somebody and they do a PR or some change request and you need to review it, well, depending on how much the data they've changed, if they've just changed a line of code and haven't ran it, then maybe the only diff is that change in the line of code. And depending on how they were sneaky about making that code change and whether they tested it or not, you know if they've ran that notebook and then did they diff on that, then it would be a totally different thing, so for me to review or look at code and say OK yeah this makes sense what you are doing is hard, because of all the metadata and the output and everything else. So the solution was to rip all that metadata out, now that became complex in the one instance that I can think of, because we were doing some fairly complex computations, and it will take hours to run some of these things."

Appendix G

H.R.E.B. Ethics Approval

In the next page I have attached the approval for my research study by the Human Research Ethics Board at the University of Victoria. In total 5 documents were submitted for approval, and those are:

- **Appendix 01 - Invitation to Participate (Public - Email):** Template for emails sent to potential participants in all areas.
- **Appendix 02 - Invitation to Participate (Private - Email to Organizers):** Template for emails sent to professors at the University of Victoria for recruitment of participants.
- **Appendix 04 - Verbal Consent:** Text read to participants before they participated in the study. Contains instructions and minor disclaimers.
- **Appendix 05 - Signed Consent (Contextual Interview):** Document containing the consent form that each participant reviewed and signed before participating in the study.
- **Appendix 06 - Questions Draft (Contextual Interview):** Document with questions asked to participants during the study.

University of Victoria

Office of Research Services I Human Research Ethics Board
Michael Williams Building Rm B202 PO Box 1700 STN CSC Victoria BC V8W 2Y2 Canada
T 250-472-4545 I F 250-721-8960 I uvic.ca/research I ethics@uvic.ca

Certificate of Approval - Amendments

PRINCIPAL INVESTIGATOR	**Neil Ernst** (Supervisor)	**ETHICS PROTOCOL NUMBER** Expedited review - delegated	**18-1283**
PRINCIPAL APPLICANT	**Andreas Koenzen** **Master's student**	ORIGINAL APPROVAL DATE	08-Aug-2019
UVIC DEPARTMENT	**Computer Science**	APPROVED ON	04-Oct-2019
		APPROVAL EXPIRY DATE	07-Aug-2020

PROJECT TITLE **Version Control for Computational Notebooks**

RESEARCH TEAM MEMBERS
Margaret-Anne Storey - Co-PI, UVic

DECLARED PROJECT FUNDING **None**

DOCUMENTS INCLUDED IN THIS APPROVAL
APPENDIX 05 - Signed Consent (Contextual Interview).pdf - 30-Sep-2019
APPENDIX 02 - Invitation to Participate (Private - Email to Organizers).pdf - 30-Sep-2019
APPENDIX 01 - Invitation to Participate (Public - Email).pdf - 30-Sep-2019
APPENDIX 06 - Questions Draft (Contextual Interview).pdf - 07-Jul-2019
APPENDIX 04 - Verbal Consent.pdf - 07-Jul-2019

CONDITIONS OF APPROVAL

This Certificate of Approval is valid for the above term provided there is no change in the protocol.

Modifications
To make any changes to the approved research procedures in your study, please submit a “Request for Modification” form. You must receive ethics approval before proceeding with your modified protocol.

Renewals
Your ethics approval must be current for the period during which you are recruiting participants or collecting data. To renew your protocol, please submit a “Request for Renewal” form before the expiry date on your certificate. You will be sent an emailed reminder prompting you to renew your protocol about six weeks before your expiry date.

Project Closures
When you have completed all data collection activities and will have no further contact with participants, please notify the Human Research Ethics Board by submitting a “Notice of Project Completion” form.

Certification

This certifies that the UVic Human Research Ethics Board has examined this research protocol and concluded that, in all respects, the proposed research meets the appropiate standards of ethcis as outlines by the University of Victoria Research Regulations Involving Human Participants.

Dr. Rachael Scarth
Associate VP Research Operations

Certificate Issued On: 04-Oct-2019

Bibliography

[1] J. M. Perkel, "Why jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, pp. 145–146, Oct 2018.

[2] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer, "Enterprise data analysis and visualization: An interview study," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 2917–2926, Dec 2012.

[3] M. B. Kery and B. A. Myers, "Exploring exploratory programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 25–29, Oct 2017.

[4] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming," *Proceedings of the 4th international workshop on End-user software engineering - WEUSE '08*, 2008.

[5] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *School of Computing TR 2007-541, Queen's University*, vol. 115, 2007.

[6] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[7] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, (New York, NY, USA), pp. 174:1–174:11, ACM, 2018.

[8] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez, and et al., "Ten simple rules for writing and sharing computational analyses in jupyter notebooks," *PLOS Computational Biology*, vol. 15, p. e1007007, Jul 2019.

[9] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019.

[10] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, and A. Walenstein, eds.), no. 06301 in Dagstuhl Seminar Proceedings, (Dagstuhl, Germany), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[11] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K.-i. Matsumoto, and H. Kudo, "Software analysis by code clones in open source software," *Journal of Computer Information Systems*, vol. XLV, pp. 1–11, 04 2005.

[12] *Google Colab*. https://colab.research.google.com.

[13] *Google Cloud AI Platform*. https://cloud.google.com.

[14] *Microsoft Azure Notebooks*. https://notebooks.azure.com.

[15] *Databricks*. https://databricks.com.

[16] *nteract*. https://nteract.io.

[17] *Apache Zeppelin*. https://zeppelin.apache.org.

[18] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, pp. 97–111, Feb 1984.

[19] K. E. Iverson, "A programming language," *Proceedings of the May 1-3, 1962, spring joint computer conference on - AIEE-IRE '62 (Spring)*, 1962.

[20] F. Perez and B. E. Granger, "Ipython: A system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[21] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and et al., "Jupyter notebooks - a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016.*, pp. 87–90, 2016.

[22] A. Rule, *Design and Use of Computational Notebooks*. PhD thesis, UC San Diego, 2018.

[23] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting exploratory programming by data scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, (New York, NY, USA), pp. 1265–1276, ACM, 2017.

[24] D. W. Sandberg, "Smalltalk and exploratory programming," *SIGPLAN Not.*, vol. 23, pp. 85–92, Oct. 1988.

[25] A. Silberschatz and A. Tuzhilin, "What makes patterns interesting in knowledge discovery systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 970–974, 1996.

[26] S. Phillips, J. Sillito, and R. Walker, "Branching and merging: An investigation into current version control practices," in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, (New York, NY, USA), pp. 9–15, ACM, 2011.

[27] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, pp. 21:1–21:44, Apr. 2011.

[28] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[29] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, pp. 1–34, Mar 2009.

[30] J. Wang, L. Li, and A. Zeller, "Better code, better sharing: On the need of analyzing jupyter notebooks," 2019.

[31] *Papermill*. https://github.com/nteract/papermill.

[32] M. B. Kery, "Towards scaffolding complex exploratory data science programming practices," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 273–274, Oct 2018.

[33] A. Head, F. Hohman, T. Barik, S. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Managing Messes in Computational Notebooks*, ACM, May 2019.

[34] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in *CHI*, 2020.

[35] M. B. Kery, "Tools to support exploratory programming with data," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 321–322, Oct 2017.

[36] *Google Cloud Source Repository*. https://source.cloud.google.com.

[37] K. Herzig and A. Zeller, *Mining Your Own Evidence*, ch. 27. O'Reilly Media, Inc., Oct. 2010.

[38] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 291–301, 2017.

[39] S. Josefsson *et al.*, "The base16, base32, and base64 data encodings," tech. rep., RFC 4648, October, 2006.

[40] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 2014.

[41] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, pp. 707–710, 1966.

[42] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," *29th International Conference on Software Engineering (ICSE'07)*, May 2007.

[43] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[44] D. Spencer, *Card sorting: Designing usable categories.* Rosenfeld Media, 2009.

[45] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming," *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, 2009.

[46] B. Kantowitz, H. Roediger, and D. Elmes, *Experimental Psychology.* Cengage Learning, 2008.

[47] R. Rosenthal, *Experimenter Effects in Behavioral Research.* Irvington, 1976.

[48] P. Pirolli and S. Card, "Information foraging in information access environments," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, (New York, NY, USA), pp. 51–58, ACM Press/Addison-Wesley Publishing Co., 1995.

[49] P. L. T. Pirolli, *Information Foraging Theory.* Oxford University Press, May 2007.

[50] B. Sparrow, J. Liu, and D. M. Wegner, "Google effects on memory: Cognitive consequences of having information at our fingertips," *Science*, vol. 333, pp. 776–778, Jul 2011.

[51] M. B. Kery and B. A. Myers, "Interactions for untangling messy history in a computational notebook," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 147–155, Oct 2018.

[52] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *IEEE Computer*, vol. 40, pp. 24–32, Dec. 2007.

[53] P. J. Guo, *Software tools to facilitate research programming*. PhD thesis, Stanford University, 2012.

[54] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, (New York, NY, USA), pp. 32:1–32:12, ACM, 2018.

[55] R. Feldt, T. Zimmermann, G. R. Bergersen, D. Falessi, A. Jedlitschka, N. Juristo, J. Münch, M. Oivo, P. Runeson, M. Shepperd, D. I. K. Sjøberg, and B. Turhan, "Four commentaries on the use of students and professionals in empirical software engineering experiments," *Empirical Software Engineering*, vol. 23, pp. 3801–3820, Nov. 2018.

[56] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, "Software history under the lens: A study on why and how developers examine it," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept. 2015.

[57] M. B. Kery, B. E. John, P. O'Flaherty, A. Horvath, and B. A. Myers, "Towards effective foraging by data scientists to find past analysis choices," *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*, 2019.

[58] M.-A. Storey, L. Singer, B. Cleary, F. F. Filho, and A. Zagalsky, "The (r)evolution of social media in software engineering," in *Proceedings of the on Future of Software Engineering - FOSE 2014*, 2014.

[59] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the ide," in *Proceedings - International Conference on Software Engineering*, pp. 1295–1298, 05 2013.

[60] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 102–111, 2014.