Code Cloning Habits Of The Jupyter Notebook Community

Ulf Sigvardsson
Abstract

**Code Cloning Habits Of The Jupyter Notebook Community**

*Ulf Sigvardsson*

Code reuse has the benefits of saving time and resources but poses a risk when attempting to tailor copied code for a new purpose or in cases when such copies are buggy or otherwise faulty. In the field of data science, the web application Jupyter Notebook is a popular tool for creating computational notebooks, documents containing both plain text and code snippets, many of which are publicly available on code hosting sites such as GitHub. This thesis describes the acquisition of approximately 2.6 million computational notebooks and analysis of this data set. By hashing the contents of every code snippet, using the MD5 hashing algorithm, cloned snippets were found through snippets producing identical hashes. By subsequently mapping the snippets to their corresponding notebooks, the relative originality of a notebook could be determined. This analysis shows that nearly 95% of notebooks are written in some version of Python. Furthermore, nearly 54% of notebooks in the data set are comprised of code blocks also found in other notebooks and, on average, approximately 70% of the code in any given notebook is copied from elsewhere.
# Contents

1 Introduction .......................... 1
   1.1 Data Science .......................... 1
   1.2 Jupyter Notebook ...................... 2
      1.2.1 Jupyter Concepts ................... 2
   1.3 Problem Statement ...................... 2
   1.4 Purpose and Goals of Work ............ 3

2 Related Work .......................... 5

3 Methodology ........................... 7
   3.1 Data Acquisition ....................... 7
      3.1.1 Acquiring URLs ..................... 8
         3.1.1.1 Partitioning the search space .... 9
         3.1.1.2 Removal Of Duplicate URLs ....... 9
      3.1.2 Downloading Notebooks ............. 10
   3.2 Database Construction ................ 11
   3.3 Limitations and Simplifications ..... 14

4 Data Analysis .......................... 15
   4.1 Language Distribution ................. 15
   4.2 Clones ................................ 15
   4.3 Common Python Libraries .............. 20
   4.4 Concluding Remarks .................... 21
      4.4.1 Amount of Clones ................. 25
      4.4.2 Library Usage ...................... 25

5 Criticism .............................. 27
   5.1 Threats to Validity .................... 27

6 Conclusions ........................... 29
   6.1 Discussion ............................ 29
   6.2 Future Work ........................... 30

Appendices .............................. 35
   A MySQL Database ....................... 36
      A.1 Installing MySQL and Importing Database ....... 36
      A.2 Example Usage ........................ 37
      A.3 Database Schema ...................... 38
   B Tables ................................. 39
      B.1 Various statistics ................... 39
      B.2 Common Libraries ................... 40
   C Scripts ................................ 43
C.1 GitHub Mining ................................................................. 43
C.2 Notebook Downloading Script ........................................... 43
C.3 Figures ........................................................................... 43
1. Introduction

This study investigates to what degree code copying occurs in the field of data science by analyzing a code base consisting of over two million source files written in the web application Jupyter Notebook. It is intended as a preliminary study, with the main contribution being the acquisition of a large data set for future research to analyze further.

1.1 Data Science

The field of data science aims, among other things, to find algorithms and processes to discover useful patterns in large collections of data but also delves into ways of transforming said data from an un-ordered state into an ordered one [11], suitable for analysis. Data science is closely associated with machine learning and data mining, but where the latter indeed focus on algorithms for pattern recognition and analysis of structured data respectively, data science also attend to ethical aspects of data management and deal with unstructured data sets [11]. As such, a data scientist needs to be well versed in the fields of mathematics and statistics in order to correctly interpret and manipulate raw data. Furthermore, as the sheer volume of data and parameters are far too great to compute manually, a certain level of programming knowledge is required. One popular tool within the data science community is Jupyter Notebook [26], a web-application suitable for handling and presenting analysis of large data sets, combining markup language with common programming languages such as Python and Scala.
1.2 Jupyter Notebook

Jupyter Notebook is an open-source web application under Project Jupyter \(^{[26]}\) that allows for the creation and sharing of documents, or notebooks, containing code, data, graphs and visualizations. In the field of data science, these notebooks are frequently used to process and visualize data \(^{[23]}\). Therefore, Jupyter Notebook has become a popular user interface in cloud computing with cloud providers such as Google’s Colaboratory \(^{[28]}\) and Amazon’s SageMaker Notebooks \(^{[13]}\) adopting the interface or providing derivative tools of their own. The Jupyter Project originates from the IPython Project, used to provide interactive online access to Python \(^{[26]}\). Jupyter Notebooks have retained the .ipynb file suffix but, unlike its predecessor, supports other languages in addition to Python \(^{[26]}\).

Python, being a high-level object-oriented dynamically typed scripting language, is a popular choice among novice programmers due to its high level of abstraction. In addition, the relative ease at which data may be visualized and processed through a plethora of abstract Python libraries makes it a popular choice for Jupyter Notebook users. As inexperienced programmers tend to utilize tutorials and use preexisting solutions not specifically tailored for the problem at hand, the author hypothesizes that the proposed Jupyter Notebook corpus have significant potential of containing clusters of duplicated code.

1.2.1 Jupyter Concepts

A notebook is a document comprised of cells, or snippets, containing statements in either markdown text or code. This allows for source code and figures to be comprehensively narrated by the author, documenting steps and methods used for the benefit of themselves or to other readers. An example of the general Jupyter interface can be seen in Figure \(1.1\). Every notebook is associated with a kernel, an execution environment processing the language used in the notebook \(^{[26]}\). Jupyter Notebook supports over 40 programming languages.

1.3 Problem Statement

Conclusions drawn from analysis of big data are highly dependent on the data being processed and interpreted in a correct manner. It follows that logical errors in
the source code processing said data may have a significant impact on the final result. StackOverflow [19] is a popular website within the programming community where questions regarding coding problems are frequently asked and answered. StackOverflow contains a documented amount of outdated code and incorrect solutions as well as plainly buggy code [19]. This habit of copy-and-pasting programming tutorials and sharing of potentially unreliable source code may risk faulty code propagating throughout the perhaps somewhat unaccustomed data science community. To identify the presence of any such errors would benefit the field of data science. In order to perform this research, a substantial body of notebooks to analyze is required as well as confirmation that Jupyter Notebook users display habits of code cloning.

1.4 Purpose and Goals of Work

The goal of this paper is to acquire a substantial body of notebooks with the purpose of understanding code produced by the data science community. Furthermore, this study attempts to profile the data set in order to identify certain pat-
terns and characteristics, specifically segments of identical code originating from distinct notebooks. By grouping such duplicates into corresponding clusters, one can determine the degree of shared code bases within the Jupyter Notebook community and point to potentially erroneous code being used *en masse*, as well as the characteristics of such code, in order to identify suitable starting points for future research. Furthermore, this paper attempts to determine the 30 most commonly used Python imports, the frequency at which they are used as well as any correlations between them.
2. Related Work

The research group at Uppsala University involved in this preliminary study, has several publications related to program tracing and analysis. In 2014, Åkerblom et al. [2] traced the execution of 19 programs and logged a total of 7.4 million events, such as getting attribute values or changing the attribute of an object. Another 2015 study by Åkerblom et al. [3] examined how polymorphism is used in Python programs. They ran 36 open source Python programs a total of 552 times, finding that some classes do change during runtime, making typing complicated as types are difficult to predict. Stephan Brandauer and Tobias Wrigstad [4] presents, in a 2017 study, the tool Spencer, an API framework for dynamic analysis using a set of program traces, applicable on a shared data set through queries to the API. This tool was used by the Brandauer and Wrigstad in another 2017 study, to map the properties of running programs, objects and references in 9 Java programs from the DaCapo Benchmark Suite. They found that "a significant amount of static invariants relating to aliasing and immutability exists in unaltered Java programs" [4].

In a 2018 study at UC San Diego by Adam Rule et al. [23], approximately 1.3 million publicly available Jupyter Notebooks were downloaded and profiled with regards to lines of code, number of cells, code-to-text ratio etc. Their goal was to determine the degree of explanation of code and results from the corresponding notebook authors as well as conduct interviews with analysts to better understand their usage of notebooks. This study focuses on the level of narrative provided by authors and to what degree they provided discussions for their reasoning or results. These interviews suggest that some analyst use notebooks solely for documentation purposes and not for actual research. It also found that analysts keep,
often messy and undocumented, old code for reuse in other notebooks.

A 2017 study by Petr Maj et al. examined the rate of duplicate source files on GitHub from 4.5 million files written in Java, C++, Python, and JavaScript. The study found a high percentage of duplicated code, with 71% of Python files being exact duplicates of other files [12]. Maj et al. used a larger data set for their analysis than this study, but does not delve into the semantics of the data, such as what libraries are being used. While this study includes programming languages supported by Jupyter Notebook, it does not include actual notebooks.

In a 2011 study, Ossher et al. [17] examined the extent of code cloning in Java files using MD5 hashes, similarly to this study. They studied 13 000 project containing around 2 million files, finding that over 10% of files being pure clones and 15% of all project containing at least one cloned file.

In 2007, Mockus [14] explored code reuse in 38 000 open source projects containing 5 million files. This definition of duplication was based on file names shared between projects. Findings show that approximately half of the files occur in more than one project. It also attempted to identify groups of files reused between projects and determine the code that is most commonly reused. Most of the reused code were small and the most common bundle was language translations for user messages.

In their 2019 article [9], Gharahyazie et al. developed a code-cloning detection tool able to spot duplicated code in projects on GitHub. In selected projects, they found that between 69% and 91% of all clones were so called within-project clones, depending on the token size of the chunks of code, between 34% and 7% being cross-project clones and clones of any type making up between 5% and 10% of the entire code base. They detected clones from files with identical abstract syntax trees, where this study does not construct any ASTs but instead rely on hashes of file contents.
3. Methodology

This section describes the methodology used for retrieval of meta data regarding Jupyter Notebooks, i.e., their URL, via the GitHub API and the subsequent downloading of the notebooks themselves, thus compiling a corpus for further analysis.

3.1 Data Acquisition

GitHub is, at the time of writing, the leading code-hosting platform [18] offering both private and public repositories for hosting source code. One approach to acquiring the data set is simply finding a public repository or database with a sufficiently large code base. One such database is the results of the Adam Rule et al. study described in Section 2 and contains approximately 1.3 million Jupyter Notebooks, claimed to be nearly all Jupyter Notebooks publicly available on GitHub at the time [22]. A drawback of using this data however, is the discrepancy of nearly two years between this study and the writing of this paper. That data set is not necessarily representative of the current Jupyter Notebook community.

Instead, via the GitHub Search API [8], repository information can be obtained through queries specifying repository owner, file extensions, programming language etc [23]. By scraping GitHub for all such files, a more up-to-date and, potentially, larger corpus can be acquired. Requests for publicly accessible content, i.e. files in public repositories, can be made through queries to the GitHub Search API [8]. Responses are sent as a JavaScript Object Notation (JSON) file. A JSON file has the structure of a dictionary, with human readable key-value pairs that are
easily parsed by programs [8]. This response contains a list of results, matching the specified constraints of the query. Individual list items state the URL of the file as well as meta data such as repository owner, commit history, issues etc. An excerpt of such a response is seen in Figure 3.1.

3.1.1 Acquiring URLs

The URLs were acquired with a Python script written by Adam Rule et al. [23], making a series of queries, requesting files meeting various criterion.

GitHub labels repositories with a language tag corresponding to the language used in it [1] and are returned with the API responses. The language of a Jupyter Notebook repository is simply classified by GitHub as Jupyter Notebook, regardless of the programming language specified in its kernel. The files of interest to this study are thus those with the .ipynb suffix and written in the language Jupyter Notebook, both which can be stated as constraints in an API query. The language constraint is required to distinguish IPython repositories from Jupyter Notebook ones and the suffix constraint ensures that only actual notebooks are found. A third condition used by Adam Rule et al. is requiring "ipynb" in the path of the files [1] and was included in this study for the possibility of comparison of data sets. As repositories may only be tagged with one language, determined by GitHub as the most suitable one, some notebooks may be missed if their repositories contain a too great amount of code written in other languages.

The GitHub API may be used either authenticated or unauthenticated. Unauthenticated users are limited to 10 requests per minute, whilst authenticated users may make up to 30. For this purpose, GitHub access tokens can be generated from an existing GitHub account. This token, when passed as a header in an HTTP request, is used to authenticate the user, effectively increasing the search rate by a factor of 3 [8].

Queries resulting in more than a certain amount are distributed over several pages, with the default value being 30 results per page [27]. To minimize the number of pages required to iterate over, and as a result the total number of requests, the number of results per page may be set to the maximum allowed [18] of 100, specified by a parameter in the query. Results spanning multiple pages are indicated by the response and separate processing is required for each page.

---

1 This condition is trivially met by the .ipynb file extensions of Jupyter Notebook
3.1.1.1 Partitioning the search space

GitHub only allows for a maximum of 1000 results to be processed at a time [25]. In order to limit the number of results in a single query, the queries were made with additional dynamic file size constraints to keep the number of results, to the furthest possible extent, below 1000. GitHub places a strict file size limit of 100 megabytes [29], and the full span of the search space was thus from zero to 100 megabytes. By making an initial request of a certain range, examining the number of results and, if in excess of 1000, decreasing the range accordingly until a sufficiently small result is found, this range was mostly covered, with a few exceptions. Conversely, whenever a range resulted in less than 500 results, the range for the subsequent query was doubled. In the rare cases when more than 1000 results were found for an interval of zero bytes, these ranges were simply written to a log file and ignored. These intervals are shown in Table 3.1 totalling 11,914 notebooks.

As previously stated, the API returns any query results in JSON format, effectively providing a dictionary of meta data for virtually all files of interest hosted publicly on GitHub. An excerpt of the content of such a JSON file is shown in Figure 3.1.

3.1.1.2 Removal Of Duplicate URLs

Since the acquisition of meta data spanned over several days, files may have changed in size in-between queries and might therefore appear in multiple results. Before any analysis, all such duplicates must be discarded. By iterating over
Table (3.1)  Missing ranges of meta data

<table>
<thead>
<tr>
<th>Range in bytes</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 874–15 874</td>
<td>1 383</td>
</tr>
<tr>
<td>15 933–15 933</td>
<td>1 439</td>
</tr>
<tr>
<td>16 254–16 254</td>
<td>1 161</td>
</tr>
<tr>
<td>16 768–16 768</td>
<td>1 086</td>
</tr>
<tr>
<td>17 320–17 320</td>
<td>1 399</td>
</tr>
<tr>
<td>17 355–17 355</td>
<td>1 404</td>
</tr>
<tr>
<td>17 427–17 427</td>
<td>1 170</td>
</tr>
<tr>
<td>18 004–18 004</td>
<td>1 398</td>
</tr>
<tr>
<td>46 419–46 419</td>
<td>1 474</td>
</tr>
<tr>
<td>Total</td>
<td>11 914</td>
</tr>
</tbody>
</table>

all meta data files and filtering out duplicate URLs, 44 617 duplicates were found among the 2 814 381 meta data entries. In total, 2 769 303 unique URLs were acquired.

3.1.2 Downloading Notebooks

Downloads of notebooks were made using the raw URLs resulting from the metadata search, covered in Section [3.1.1]. The contents found at these addresses were saved as individual notebooks using scripts by Adam Rule et al. The sheer volume of the data set made downloading from a single computer impractical. For this reason, several cloud computers were used, each handling a distinct set of notebook URLs.

As previously mentioned, GitHub limits the number of requests made per minute by a single user. As the downloads were made using regular HTTP requests without the GitHub API, the remaining requests per minute was left unchecked, increasing the chances of exceeding the maximum hit limit allowed by GitHub. As this could effectively block the IP that made the request, measures had to be taken to stay within the allowed limits.

Thus, downloads were made in batches divided by file sizes, adding control of the download rate for different batches. In order to stay within the allowed hit limit, the download rate was slowed for smaller size ranges by halting for a fraction of a second between each request, thus avoiding the longer delay of a hit rate
timeout or a permanent block as a result of hit rate abuse. This was generally not necessary for batches of files larger than 50kB, since the time for processing these files in general covered the shorter delay. Thus, using an explicit halt for such files would have caused an unnecessary delay.

Another benefit of partitioning the downloading process is to enable intermediate quality control and code modifications, such as handling of exceptions, and the practical benefits of avoiding a single concurrent process requiring weeks of running time.

In total, 903 GBs of data was downloaded over the span of eight weeks.

### 3.2 Database Construction

Analysis of the data set is made by first constructing a database containing entries for every notebook and code cell with additional metadata such as programming language and clusters of identical code. A high level model of the database is shown in Figure 3.2. For a full database schema, see Appendix A. The purpose of this database is to assist in identifying subsets of particular interest for further research and to provide a general profile for the corpus with regards to parameters such as common programming languages and duplication ratios with respect to notebook sizes and number of snippets. The entire analysis is static in nature and while there are other possible methods, such as dynamic analysis, which the Uppsala University research group has documented experience in, this cannot be applied prior to the basic analysis and localization of clones, for which static analysis is better suited.

When comparing snippets, string matching is not a practical method, requiring quadratic complexity to compare every snippet to all others. Instead, by feeding the contents of a snippet to a hashing algorithm, snippets with identical hashes may be mapped to corresponding *clusters*.

The hashing algorithm used is the *MD5 hashing algorithm* [6], producing a 128 bit hash generated from a string, in this case; the source code making up the snippet. The choice of MD5 over algorithms generating hashes of greater length, such as SHA1, was made with a trade off between the probability of two distinct object producing the same hash and the time needed to generate the hashes. For example, SHA1 produces 160 bit hashes but requires 80 operations to do so [20].
Figure (3.2) ER diagram of MySQL database used for analysis

while MD5 only uses 64 operations to produce its hashes [21]. The risk of two distinct object producing the same hash and thus polluting the results is well within the tolerance of this paper. See Section 5.1 for a more detailed justification of this claim.

As snippets may contain comments, two identical snippets with respect to code content may produce different hashes. Thus, every snippet was stripped of any comments by discarding the remainder of a line after encountering a '#' symbol. The remaining code is then stripped of all whitespaces and newline characters, resulting in a standard formatted string by which snippets can be compared.

In order to trace the origins of the snippets, a notion of which snippets belong to what notebook is required. One option considered was having each snippet pointing to the one following it in its notebook, and keeping a reference in each notebook to its first snippet although traversing a linked structure was deemed inefficient. Instead, each snippet has a reference to its notebook of origin as well as its index within the cells object of its notebook. This design allows for access in constant time without traversing a linked list while still preserving the order
of snippets.

Database notebook entries initially contained a unique ID, the path to the notebook within the folder structure of the project, the programming language used and the total number of code snippets. Due to the frequent need to join tables for analysis, this was later expanded to include the number of unique snippets, total code characters and the ratio of duplicated snippets. As every Cluster entity keeps track of how many members it has, every Snippet entity may indicate if it is a clone by examining if its corresponding cluster has an Instances attribute greater than 1. This information is then stored in the Snippet attribute IsUnique, a boolean value representing whether or not a snippet is unique. The number of characters of a snippet is redundantly kept in each Snippet entity in order to relay this information to the total notebook character count without having to perform additional joins.

The UniqueSnippets of each notebook is computed by summing the IsUnique columns of every snippet associated with a Notebook entity, resulting in the number of duplicated snippets, and then subtracting this number from the total snippet count. The duplication ratio of each notebook is then trivially calculated as

$$\text{DupRatio} = \frac{\text{NoOfSnippets} - \text{UniqueSnippets}}{\text{NoOfSnippets}}$$.

The language of a notebook is obtained from a key in the file. This key varies depending on the Jupyter version. It is also possible, but somewhat rare, that no such key exists in which case the language of the notebook cannot be trivially extracted. It is possible to obtain this information from the individual cells. Due to time constraints, and since these cases only appear in approximately 1.4% of notebooks, such entries are assigned an unknown language value. Additional snippet metadata includes the number of code characters. This is a more suitable metric than the actual size of the file, as images and data structures may take up a disproportionate amount of the space, unrelated to the programming logic.

Several notebooks were excluded from the database due to merge conflict headers being present in the JSON code or plain syntactical errors, thus resulting in an error when parsing. In addition to parsing errors due to merge headers, 23,442 notebooks could not be processed due to the files being empty. This is likely a result of files being deleted or modified between meta data acquisition and downloading of the file. This hypothesis is supported by the significant higher error
frequency among smaller files. Parsing errors also occurred from files consisting of an empty JSON object as these are technically not empty files and therefore were not caught by the initial cleaning. In total, the database consists of 2,603,321 notebooks and 32,288,102 snippets. This is roughly twice the number of notebooks gathered by Adam Rule et al. [23].

3.3 Limitations and Simplifications

This method does not exclude forks at the stage of URLs which naturally impacts the clone analysis results.
4. Data Analysis

This chapter describes clone analysis of the entire corpus using the database described in Section 3.2 as well as a mapping of common libraries in subsets of Python notebooks. This analysis covers 2,603,321 notebooks and 32,288,102 snippets. Analysis is done excluding empty snippets, as these are technically clones of each other but uninteresting such.

4.1 Language Distribution

A vast majority of notebooks are written in various versions of Python, as shown in Figure 4.1. Among the remaining notebooks, determining the language was unsuccessful for 1.431% of notebooks, although Scala, Julia and R are among the most common, with 0.16%, 0.794% and 0.785% of all notebooks respectively. Analysis shows that virtually all notebooks in the data set contain 77 or fewer snippets and is comprised of 16,820 characters of code or less. This allows research to be made on severely reduced ranges while still covering the bulk of the data set. Various percentiles for these metrics are shown in Table 4.1.

4.2 Clones

Out of the 32,288,102 snippets in the database, 24,087,476 (74.6%) are found in more than one notebook, although these numbers do not account for the presence of
forks. When profiling duplicating patterns of the four most common languages, a pattern of either completely original or cloned code is observed, as shown in Figure 4.2. Notebooks written in Python displays least originality with 12.5% of them, excluding empty notebooks, being comprised of distinct snippets, and R the most original with 18.3% distinct notebooks. For Julia and Scala this number is 16.6% and 18% respectively. In contrast, 53.7% of non-empty Python notebooks, 50.2% of Julia notebooks, 52.1% Scala notebooks and 48.7% of R notebooks are pure clones, completely consisting of snippets also found in other notebooks. On average, the percentage of duplicated snippets for Python, R, Julia and Scala is 69.8%, 62%, 63.6% and 64% respectively. The entire corpus has an average clone ratio of 69.8%. The median duplication ratio, in regards to notebooks, for the entire corpus is 100% as well as the third quartile, displaying a high number of complete clones. Out of 32,288,102 snippets, 8,200,626 are unique.

<table>
<thead>
<tr>
<th>Metric</th>
<th>10th</th>
<th>50th</th>
<th>90th</th>
<th>99th</th>
<th>100th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of snippets</td>
<td>2</td>
<td>9</td>
<td>28</td>
<td>77</td>
<td>1,358</td>
</tr>
<tr>
<td>Number of characters</td>
<td>206</td>
<td>1,278</td>
<td>5,543</td>
<td>16,820</td>
<td>7,991,943</td>
</tr>
</tbody>
</table>

Table 4.1  Percentiles for total number of characters and snippets in notebooks for the entire data set.

Examenting the correlation of the rate of duplicated snippets with the number of
snippets and the total character count in a notebook (excluding notebooks without any unique snippets), a pattern of higher duplication ratios is observed in notebooks of high snippet count and low character count, as seen in Figure 4.3. The concentration of clones is most notable in notebooks under 850 characters, containing more than 30 snippets, although a cluster of roughly 50% clone ratio can be seen in notebooks containing between 9 350 and 10 200 characters and having between 22 and 30 snippets. When including completely cloned notebooks, the same pattern is observed, tending to display higher clone ratios towards notebooks of low character count containing high numbers of snippets. A cluster of complete clones can be seen in notebooks between 15 300 and 16 150 characters and between 22 and 30 snippets, as seen in Figure 4.3.
Figure (4.3)  Duplication ratio for notebooks up to the 99th percentiles of snippet and character count for all languages. Including fully cloned notebooks.
Figure (4.4) Duplication ratio for notebooks up to the 99\textsuperscript{th} percentiles of snippet and character count for all languages. Excluding fully cloned notebooks.
4.3 Common Python Libraries

An important aspect in identifying coding patterns and the fields in which the Jupyter Notebook community operate is determining what libraries are frequently used, and in which combinations of libraries that are often used simultaneously. As 92% of notebooks are written in Python, this analysis focuses on Python notebooks and their respective libraries.

This analysis was done by first selecting several subsets of Python notebooks over different sizes, ranging from 0–50kB, 50–100kB and 1–30MB, determining their respective use of external libraries, their use of said libraries in conjunction with other libraries as well as the most frequently used library functions. The reason for merely operating on these subsets is a lack of computational power, and each subset was subsequently limited to at most 100 000 notebooks. These subset display different characteristics, both in regards to the most common libraries but also in the nature of combinations of libraries. The 30 most common libraries for the respective subsets can be seen in Appendix B.2.

When analyzing combinations of library uses, it is of interest to investigate the complementarity between them. A high frequency of library A, given library B, suggests some functionality in A being complementary to some functionality in B. For example: one library may perform complex calculations while another handles the plotting of the results. For this paper, the normalized level of complementarity of library A given library B is defined as \( P(A|B) - P(A) \), where \( P(A|B) \) is the number of encountered instances of A in notebooks also containing B divided by the total number of instances of A, and \( P(A) \) the number of instances of A divided by the total number of imports made.

For all three ranges, the top three imports are numpy, pandas and matplotlib. numpy provides linear algebra capabilities and multi-dimensional array objects useful for scientific calculations [16]. matplotlib is used for visualizing data with both 2D and 3D charts [16]. pandas provides powerful data structures, called data frames that are tightly intertwined with libraries such as PyPlot. These three libraries are all part of what is referred to by some [5] as the scientific Python stack and are frequently used together [5].

For notebooks smaller than 50kB, numpy appears in 39.7% of the notebooks, pandas in 27.3% and matplotlib in 25.2%. Many of the top 30 libraries for this range show a high complementarity with numpy and to a lesser degree pandas and matplotlib, as seen in Figure 4.5. A clear outlier of this range is the mirrored
complementarity between bs4, providing web-scraping functionality, and requests, the de facto standard for making HTTP requests in Python. seaborn is a Python data visualization library based on matplotlib and shows high complementarity to numpy, matplotlib and pandas.

When analyzing notebooks between 50–100kB, numpy, matplotlib and pandas are still the three most common libraries but with much higher frequencies, occurring in 71.03%, 60.57% and 49.62% of notebooks, respectively. A similar complementarity pattern as for the previous range is seen, with most libraries showing complementary functionality with numpy, matplotlib and pandas although to a higher degree. seaborn is more common, jumping from 12th to 5th most common from 4.17% to 12.67% of notebooks.

Notebooks ranging between 1–30MBs display a significantly higher ratio of several libraries. numpy is found in 85.08% of notebooks, matplotlib in 79.27%, pandas in 52.52%. os occurs in 32.87% and seaborn in 25.39% of notebooks. Most of the top 30 libraries show a high complementarity with numpy and matplotlib, and to a lesser degree with pandas and os, as seen in Figure 4.7. Standing out is cufflinks, a library binding plotly to numpy dataframes, with a high complementarity with said libraries and by far the library with highest complementary functionality towards plotly.

These subsets also seem to show differences in the number of imports being used by a single notebook. Table 4.2 shows imports found in at least 10% of Python notebooks over the three aforementioned subsets. The top imports in all three subsets are numpy, matplotlib and pandas, with their respective frequency increasing with larger notebooks. Furthermore, notebooks under 50kB have only these three libraries occurring in at least 10% of them, whereas the 50–100kB and 1–30MB ranges have 6 and 19 libraries cracking this threshold respectively.

4.4 Concluding Remarks

This section summarizes the data found in Section 4.2–4.3.
Figure (4.5) Library combinations in 100,000 Python notebooks of sizes under 50kB. Cell values indicate the probability that a library from the Compatible with axis is imported in a notebook given a library from the Primary library axis is imported. Probabilities are normalized to account for the number of times a Primary library is imported.
Figure (4.6) Library combinations in 100 000 Python notebooks of sizes between 50–100kB. Cell values indicate the probability that a library from the Compatible with axis is imported in a notebook given a library from the Primary library axis is imported. Probabilities are normalized to account for the number of times a Primary library is imported.
Figure (4.7) Library combinations in 54,025 Python notebooks of sizes between 1–30MB. Cell values indicate the probability that a library from the Compatible with axis is imported in a notebook given a library from the Primary library axis is imported. Probabilities are normalized to account for the number of times a Primary library is imported.
4.4.1 Amount of Clones

As shown in Section 4.2, notebooks completely comprised of non-original snippets are rare, with 12.6% of all notebooks being original. Approximately 75% of all snippets in the corpus appear in at least two distinct notebooks, and 53.9% of all notebooks are comprised of snippets also found in other notebooks. Note that this analysis also includes forked repositories, and these numbers are most likely somewhat skewed.

4.4.2 Library Usage

The sample subsets display a higher number of libraries that are found in more than 10% of notebooks as the sizes of notebooks increase, as seen in Table 4.2. Many of the top Python libraries are rather general, such as numpy, time, pandas and pyplot, but several specialized packages such as torch and tensorflow can be observed, both focusing on neural networks and machine learning [10]. Larger notebooks have higher overall dependencies between libraries with a clear concentration towards numpy, matplotlib, pandas and os.

Differences in usage between notebooks of different sizes are observed. For example: notebooks of sizes 1–30MB import os 32.87% of the times. This is more than twice the rate of notebooks between 50–100kB, and three times the ratio of notebooks of sizes 0–50kB. Similarly, notebooks between 1–30MB import seaborn at a rate double to that of notebooks between 50–100kB and more than five times the rate of notebooks between 0–50kB.

Some negative correlations are observed, but are insignificant. The lowest such correlation for any sample subset is -0.025 for tensorflow given bs4. Thus, the 30 most common libraries are void of any strong negative correlations.
<table>
<thead>
<tr>
<th>Library</th>
<th>0–50kB % of NBs</th>
<th>50–100kB % of NBs</th>
<th>1–30MB % of NBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>39.72%</td>
<td>71.03%</td>
<td>85.08%</td>
</tr>
<tr>
<td>pandas</td>
<td>27.36%</td>
<td>49.62%</td>
<td>52.52%</td>
</tr>
<tr>
<td>matplotlib</td>
<td>25.28%</td>
<td>60.57%</td>
<td>79.27%</td>
</tr>
<tr>
<td>seaborn</td>
<td>12.67%</td>
<td>15.10%</td>
<td>25.39%</td>
</tr>
<tr>
<td>sklearn</td>
<td>10.67%</td>
<td>12.67%</td>
<td>25.39%</td>
</tr>
<tr>
<td>os</td>
<td>15.10%</td>
<td>25.39%</td>
<td>32.87%</td>
</tr>
<tr>
<td>scipy</td>
<td>17.45%</td>
<td>12.30%</td>
<td>17.45%</td>
</tr>
<tr>
<td>time</td>
<td>16.23%</td>
<td>15.29%</td>
<td>16.23%</td>
</tr>
<tr>
<td>sys</td>
<td>15.29%</td>
<td>13.18%</td>
<td>15.29%</td>
</tr>
<tr>
<td>math</td>
<td>13.39%</td>
<td>13.18%</td>
<td>13.39%</td>
</tr>
<tr>
<td>random</td>
<td>13.18%</td>
<td>12.30%</td>
<td>13.18%</td>
</tr>
<tr>
<td>collections</td>
<td>12.30%</td>
<td>12.30%</td>
<td>12.30%</td>
</tr>
<tr>
<td>cv2</td>
<td>12.30%</td>
<td>11.90%</td>
<td>12.30%</td>
</tr>
<tr>
<td>glob</td>
<td>11.90%</td>
<td>11.94%</td>
<td>11.90%</td>
</tr>
<tr>
<td>plotly</td>
<td>11.94%</td>
<td>11.87%</td>
<td>11.94%</td>
</tr>
<tr>
<td>sklearn</td>
<td>11.87%</td>
<td>11.91%</td>
<td>11.87%</td>
</tr>
<tr>
<td>tensorflow</td>
<td>11.91%</td>
<td>10.15%</td>
<td>11.91%</td>
</tr>
<tr>
<td>datetime</td>
<td>10.15%</td>
<td>10.44%</td>
<td>10.15%</td>
</tr>
<tr>
<td>re</td>
<td>10.44%</td>
<td>10.64%</td>
<td>10.44%</td>
</tr>
<tr>
<td>warnings</td>
<td>10.64%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table (4.2) Libraries found in more than 10% of notebooks (NBs) for different notebooks sizes.
5. Criticism

Snippets of chronologically differing but otherwise semantically identical code will generate different hashes, although they are for all intents and purposes identical. This coarseness of this approach to clone analysis potentially obscures a significant amount of clones. The average duplication ratio is most likely higher than what is found in this paper. An example of two semantically equivalent segments of code producing different hashes is shown in Figure 5.1.

Conversely, although this analysis present statistics of the rate of duplicated snippets, it does so without taking in to account the sizes of the snippets. For example; consider two notebooks containing two snippets each and sharing one of them between each other. If their respective non-duplicated snippets differ in size they still have an equal duplication ratio of 0.5.

\[
\begin{align*}
\text{int } i &= 0 & \text{int } j &= 1 \\
\text{int } j &= 1 & \text{int } i &= 0
\end{align*}
\]

Figure (5.1)  Semantically equivalent code, producing different MD5 hashes

5.1 Threats to Validity

Any conclusions drawn from duplication rates depend on distinct snippets not producing identical hashes. For hashes of \( n \) bits and \( m \) outputs, the expected number of collisions is
The reasoning for this is that there are \( \binom{m}{2} \) separate pairs of hashes, each with a probability of colliding of \( 2^{-n} \) and hence an expected number of collisions of \( 2^{-n} \). The expected sum of a set of probabilistic values is the sum of each value’s expectation. Given 128 bits used for hashing and 32 288 102 snippets to hash, the expected number of collisions are approximately \( 1.53 \cdot 10^{-24} \). The author argues that this is an acceptable margin of error.

Notebooks of unknown languages make up 1.431% of the corpus and may potentially affect analysis of the less frequent languages such as Julia, R and Scala. However, this study has made notebooks written in Python its primary focus and includes statistics for other languages strictly for comparison.

When analyzing import frequency, every library is expected to coincide with itself at a rate of precisely 1.0. However, slightly lower figures are sometimes observed suggesting that the scripts parsing the notebooks are somewhat faulty, although these deviations are very minor. For example; in the notebooks under 50kB, 38,479 imports of numpy were observed. The number of combinations of numpy with itself for this range is 38 472, a deviation of 0.018%.

The `NoOfChars` attributes of `Snippet` and `Notebook` entities are computed once and present an update anomaly should a `Cluster` entity be modified.

This analysis does not factor in forks of repositories, potentially skewing the results towards a greater duplication rate.
6. Conclusions

The work behind this paper resulted in a corpus of 2,603,321 Jupyter Notebooks, as well as an MySQL database containing information for said notebooks to serve as a foundation for future research. Notebooks produced by the Jupyter Notebook community contain a significant amount of duplicated code, with the average notebook being comprised of 69.8% cloned snippets, 53.9% of notebooks only containing snippets found in other notebooks and only 12.6% of notebooks being completely original. Python notebooks make up 94.93% of the corpus and have an average duplicate rate of 70.6%, dominating the statistics for the entire corpus. The three most common languages excluding Python are Julia, R and Scala, making up 0.79%, 0.79% and 0.16% of all notebooks respectively.

The 30 most used libraries have a strong positive correlation with libraries numpy, matplotlib and pandas. During library correlation analysis, very small negative correlations were found, the smallest such being −0.025. Stronger negative correlation coefficients would indicate that a library is less likely to be observed given that another is already being used. This indicates that no use of a certain library excludes the use of another.

6.1 Discussion

The distribution of languages used in notebooks as well as the three most common Python libraries are mostly consistent with the findings of Adam Rule et al.. That study shows the frequency of these libraries only in notebooks actually using
external libraries whereas this paper makes no distinction between notebooks based on this criteria, making comparisons somewhat difficult.

A possible explanation for the absence of any strong negative correlations between library usage is the tightly intertwined functionality of many libraries of similar functionality. As an example, the plotting library seaborn makes heavy use of matplotlib functionality and while superficially appearing like a stand-alone plotting library, it should rather be considered a high-level API to matplotlib.

Analysis shows that 53.7% of Python notebooks are pure clones. This result is inconsistent with the results found by Petr Maj et al., claiming a 71% of Python files being pure clones, based on file-level hashes [12] similarly to the method used for this paper. It is important to note however, that the results of Petr Maj et al. refer to pure Python files, whereas this paper investigates Jupyter Notebooks written in Python, possibly explaining this discrepancy.

6.2 Future Work

With the data set acquired by this thesis as a foundation, future work should attempt to identify erroneous code, possibly based on common pitfalls when using the common libraries identified in this paper. The pollution of result from forked repositories would also have to be addressed.

The metric of character count in a snippet or notebook may be better converted to a count of rows as is more intuitive.

While some notion of the most frequently used libraries is achieved by this paper, this can be done more accurately by converting .ipynb files into their core language, e.g. Python, Scala, Julia, etc., and constructing abstract syntax trees from which further analysis may originate. One tool for such conversion is the command-line tool ipython nbconvert [15].

The degree of code duplication can be better determined using existing algorithms for the longest common sub-sequence problem, taking into account similar code appearing in differing order.

The analysis of frequent library imports only consider imports made within certain subsets and covers approximately 9.8% of all notebooks. A more thorough analysis would naturally examine the entire data set.
An initial goal of this study was to construct a graph with notebooks as vertices and shared snippets as edges between them. This would allow to perform analysis using established graph theory concepts such as clustering and centrality. This was abandoned due to computational limits and time constraints but is left as a suggestion for future work.
Bibliography


[14] Large-Scale Code Reuse in Open Source Software. First International Workshop on emerging trends in floss research and development. 2007. DOI: 10.1109/FLOSS.2007.10


Appendices
A. MySQL Database

This database was constructed with MySQL 8.0.15 for OSX 10.13 on a Intel Core i5 processor. This section covers the installation of a MySQL server and importing and using the database on MacOS.

A.1 Installing MySQL and Importing Database

To install Homebrew, from the command line type

```
$ ruby -e "\$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
$ brew install mysql
$ brew tap homebrew/services
$ brew services start mysql
```

This will install Homebrew and MySQL with brew services and start the MySQL server.

To set the root password for Your server and start MySQL, type

```
$ mysqladmin -u root password 'yourpassword'
$ mysql -u root -p
Enter password: **********
```
To import and make the Jupyter database the default one, from the command line, enter

```
$ mysql -u root -p jupyter < path\to\file\jupyter.sql
```

To verify, in MySQL server

```
> SHOW DATABASES
```

```
+----------------------+
| Database             |
+----------------------+
| jupyter.             |
| information_schema   |
| mysql                |
| performance_schema   |
| sys                  |
+----------------------+
5 rows in set (0.00 sec)
```

If jupyter is not the default database, this can be done with `use jupyter` in MySQL.

### A.2 Example Usage

Every Notebooks entry contains its language ID, number of snippets, duplication ratio. The following SQL query obtains the average duplication ratio of all non-empty notebooks, optionally with a language constraint:

```sql
SELECT AVG(DuplicationRatio)
FROM Notebooks N
INNER JOIN Languages L
ON N.Language=L.LanguageId
WHERE N.NoOfChars>0
[AND Name LIKE "%some language%"];
```
The following query finds the N largest clusters

```sql
SELECT C.ClusterId, COUNT(SnippetId)
FROM Snippets S
    INNER JOIN Clusters ON S.ClusterId=C.ClusterId
GROUP BY C.ClusterId
ORDER BY COUNT(SnippetId) DESC
LIMIT [N];
```

A.3 Database Schema

![Database Schema Diagram]
## B. Tables

### B.1 Various statistics

<table>
<thead>
<tr>
<th>Language</th>
<th>Q1</th>
<th>Mean</th>
<th>Median</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>0.333</td>
<td>0.698</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Python</td>
<td>0.333</td>
<td>0.698</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Julia</td>
<td>0.167</td>
<td>0.636</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>R</td>
<td>0.143</td>
<td>0.622</td>
<td>0.938</td>
<td>1.000</td>
</tr>
<tr>
<td>Scala</td>
<td>0.143</td>
<td>0.640</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table (B.1)  First quartile, mean, median and third quartile for duplication ratio over various language constraints.
B.2 Common Libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Used by % of NBs</th>
<th>Library</th>
<th>Used by % of NBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>39.72%</td>
<td><strong>future</strong></td>
<td>3.02%</td>
</tr>
<tr>
<td>pandas</td>
<td>27.36%</td>
<td>json</td>
<td>2.66%</td>
</tr>
<tr>
<td>matplotlib</td>
<td>25.28%</td>
<td>requests</td>
<td>2.21%</td>
</tr>
<tr>
<td>os</td>
<td>9.39%</td>
<td>pickle</td>
<td>2.19%</td>
</tr>
<tr>
<td>tensorflow</td>
<td>7.04%</td>
<td>warnings</td>
<td>1.99%</td>
</tr>
<tr>
<td>time</td>
<td>5.25%</td>
<td>csv</td>
<td>1.90%</td>
</tr>
<tr>
<td>sys</td>
<td>4.99%</td>
<td>keras</td>
<td>1.86%</td>
</tr>
<tr>
<td>sklearn</td>
<td>4.86%</td>
<td>torch</td>
<td>1.71%</td>
</tr>
<tr>
<td>re</td>
<td>4.48%</td>
<td>glob</td>
<td>1.68%</td>
</tr>
<tr>
<td>math</td>
<td>4.41%</td>
<td>nltk</td>
<td>1.58%</td>
</tr>
<tr>
<td>random</td>
<td>4.33%</td>
<td>itertools</td>
<td>1.48%</td>
</tr>
<tr>
<td>seaborn</td>
<td>4.17%</td>
<td>bs4</td>
<td>1.40%</td>
</tr>
<tr>
<td>scipy</td>
<td>4.10%</td>
<td>cv2</td>
<td>1.34%</td>
</tr>
<tr>
<td>collections</td>
<td>3.16%</td>
<td>PIL</td>
<td>1.10%</td>
</tr>
<tr>
<td>datetime</td>
<td>3.13%</td>
<td>string</td>
<td>1.02%</td>
</tr>
</tbody>
</table>

Table (B.2) The 30 most common libraries of Python notebooks under 50kB
<table>
<thead>
<tr>
<th>Library</th>
<th>Used by % of NBs</th>
<th>Library</th>
<th>Used by % of NBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>71.03%</td>
<td><strong>future</strong></td>
<td>4.99%</td>
</tr>
<tr>
<td>matplotlib</td>
<td>60.57%</td>
<td>json</td>
<td>4.15%</td>
</tr>
<tr>
<td>pandas</td>
<td>49.62%</td>
<td>pickle</td>
<td>4.15%</td>
</tr>
<tr>
<td>os</td>
<td>15.10%</td>
<td>itertools</td>
<td>3.09%</td>
</tr>
<tr>
<td>seaborn</td>
<td>12.67%</td>
<td>keras</td>
<td>3.83%</td>
</tr>
<tr>
<td>sklearn</td>
<td>10.67%</td>
<td>requests</td>
<td>3.23%</td>
</tr>
<tr>
<td>math</td>
<td>8.59%</td>
<td>warnings</td>
<td>3.91%</td>
</tr>
<tr>
<td>scipy</td>
<td>8.90%</td>
<td>csv</td>
<td>2.80%</td>
</tr>
<tr>
<td>tensorflow</td>
<td>8.30%</td>
<td>glob</td>
<td>2.28%</td>
</tr>
<tr>
<td>time</td>
<td>8.87%</td>
<td>h5py</td>
<td>2.16%</td>
</tr>
<tr>
<td>random</td>
<td>7.90%</td>
<td>nltk</td>
<td>2.67%</td>
</tr>
<tr>
<td>collections</td>
<td>6.51%</td>
<td>pylab</td>
<td>2.49%</td>
</tr>
<tr>
<td>sys</td>
<td>6.79%</td>
<td>string</td>
<td>2.85%</td>
</tr>
<tr>
<td>datetime</td>
<td>5.87%</td>
<td>torch</td>
<td>2.84%</td>
</tr>
<tr>
<td>re</td>
<td>5.81%</td>
<td>tqdm</td>
<td>2.11%</td>
</tr>
</tbody>
</table>

Table (B.3) The 30 most common libraries of Python notebooks between 50-100kB
<table>
<thead>
<tr>
<th>Library</th>
<th>Used by % of NBs</th>
<th>Library</th>
<th>Used by % of NBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy</td>
<td>85.08%</td>
<td>tensorflow</td>
<td>11.91%</td>
</tr>
<tr>
<td>matplotlib</td>
<td>79.27%</td>
<td>datetime</td>
<td>10.15%</td>
</tr>
<tr>
<td>pandas</td>
<td>52.52%</td>
<td>re</td>
<td>10.44%</td>
</tr>
<tr>
<td>os</td>
<td>32.87%</td>
<td>warnings</td>
<td>10.64%</td>
</tr>
<tr>
<td>seaborn</td>
<td>25.39%</td>
<td>pickle</td>
<td>9.25%</td>
</tr>
<tr>
<td>scipy</td>
<td>17.45%</td>
<td>PIL</td>
<td>8.07%</td>
</tr>
<tr>
<td>time</td>
<td>16.23%</td>
<td>json</td>
<td>8.25%</td>
</tr>
<tr>
<td>sys</td>
<td>15.29%</td>
<td>itertools</td>
<td>7.83%</td>
</tr>
<tr>
<td>math</td>
<td>13.39%</td>
<td><strong>future</strong></td>
<td>6.03%</td>
</tr>
<tr>
<td>random</td>
<td>13.18%</td>
<td>keras</td>
<td>5.38%</td>
</tr>
<tr>
<td>collections</td>
<td>12.30%</td>
<td>cufflinks</td>
<td>4.30%</td>
</tr>
<tr>
<td>cv2</td>
<td>12.05%</td>
<td>tqdm</td>
<td>4.46%</td>
</tr>
<tr>
<td>glob</td>
<td>11.90%</td>
<td>csv</td>
<td>3.79%</td>
</tr>
<tr>
<td>plotly</td>
<td>11.94%</td>
<td>skimage</td>
<td>3.66%</td>
</tr>
<tr>
<td>sklearn</td>
<td>11.87%</td>
<td>torch</td>
<td>3.82%</td>
</tr>
</tbody>
</table>

Table (B.4) The 30 most common libraries of Python notebooks between 1-30MB
C. Scripts

C.1 GitHub Mining

The scripts used to retrieve the raw URLs for the notebooks and remove any duplicate URLs can be found at http://bit.ly/gh_md_scrape.

C.2 Notebook Downloading Script

The script used to download notebooks can be found at http://bit.ly/nb_download.

C.3 Figures

This section provides the scripts used to generate data for and draw many of the graphs presented in this paper. Links to these scripts can be found in Table C.1. Most of the graphs were generated with the pattern of having one script first retrieve data from an SQL query and storing the unpolished results in a CSV file before letting another script read this data and perform various calculations from which the results can be used to generate the graphs.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Gist</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td><a href="http://bit.ly/violin_plot">http://bit.ly/violin_plot</a></td>
</tr>
<tr>
<td>4.5, 4.6, 4.7</td>
<td><a href="http://bit.ly/jupyter_lib_correlations">http://bit.ly/jupyter_lib_correlations</a></td>
</tr>
</tbody>
</table>

Table (C.1) Gist links to scripts used to generate several of the graphs in this paper.