Karlsruhe Institute of Technology

Is there a Correlation between the Use of Swearwords and Code Quality in Open Source Code?

Bachelor’s Thesis of

Jan Strehmel

at the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. Stamatakis, Alexandros
Second reviewer:	TT-Prof. Dr. Pascal Friederich
Advisor:	M.Sc. Dimitri Höhler
Second advisor:	M.Sc. Ben Bettisworth

01. October 2022 – 01. February 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . .

(Jan Strehmel)

Abstract

One of the most fundamental unanswered questions that has been bothering mankind during the Anthropocene is whether the use of swearwords in open source code is positively or negatively correlated with source code quality. To investigate this profound matter we crawled and analysed over 3800 C open source code containing English swearwords and over 7600 C open source code not containing swearwords from GitHub. Subsequently, we quantified the adherence of these two distinct sets of source code to coding standards, which we deploy as a proxy for source code quality via the SoftWipe tool developed in our group. We find that open source code containing swearwords exhibit significantly better code quality than those not containing swearwords under several statistical tests. We hypothesise that the use of swearwords constitutes an indicator of a profound emotional involvement of the programmer with the code and its inherent complexities, thus yielding better code based on a thorough, critical, and dialectic code analysis process.

Contents

List of Figures

List of Tables

1 Introduction and Preliminaries

1.1 Introduction

One of the most commonly known software failures caused by lack of software quality is the failed maiden flight of the Ariane 5 rocket on the 4th. June 1996 [28]. This disaster made it obvious that code quality is a very important often times overlooked part of software development.
There are different approaches to code quality, with one of the most popular being the guidelines that Robert Cecile "Uncle Bob" Martin popularised with his book Clean Code. In his book he has an entire chapter dedicated to the naming of variables, functions, classes etc.. His rules state that amongst other things, you should not be cute or use puns while naming [17]. This lead us to the question whether the use of swearwords in comments and in code affects the overall code quality. Our hypothesis going into this study was that there would not be a statistical difference between repositories that contain swearwords and the control group, which contains repositories that may or may not contain swearwords. To test this hypothesis we developed a program that gathers repositories from both groups on GitHub. This was done utilising the GitHub API, which allowed us to directly search for repositories containing swearwords. After identifying and downloading repositories we assess their code quality using the SoftWipe tool [30], which returns a score in the range from 0 (low) to 10 (high) that correlates to the code quality. Repositories containing swearwords have to pass an additional test, which confirms the presence of swearwords by using regular expressions.
After this data crawling and evaluation was done, we had a sample size of $\approx$ 3800 for repositories containing swearwords as well as $\approx$ 7600 repositories for our control group. We then used several visual and statistical tests to determine the difference between the two samples. These tests included normality tests, such as the Shapiro-Wilk test, as well as hypothesis tests that compare the two samples and two sample means with each other. We also investigated the quality of our result by using confidence intervals.

1.2 Preliminaries

In this section, we cover some basic GitHub notions and terminology and the Git-API. We also briefly introduce the SoftWipe tool, regular expressions, and automatons.

1.2.1 GitHub and Git-API

GitHub: GitHub is a service that provides version control and other useful features for software development. It comprises more than 28 million public repositories and is the source of the open source code repositories (datasets) for our analysis. [29] GitHub also allows users to keep track of projects that are of interest to the user by rating a project/repository with a *star*. These stars are essentially the "likes" of GitHub [8] and have the following two effects:

1. They allow the user to find the repository easily again in the future
2. It is (informally) a measure of popularity, which allows other users to find popular and possibly interesting repositories.

We will use *stars* as a measure of popularity later on.

Git-API: The Git-API allows the user to interact and automate processes with Git. The documentation is available at `https://docs.github.com/en/rest?apiVersion=2022-11-28`. For our use case, the most important function is the code search feature, which allows us to construct search queries to quickly search GitHub for repositories containing swearwords. We also use the repository search feature to obtain the general population, meaning repositories without any special traits. We needed to handle multiple restrictions of the Git-API in order to efficiently download the repositories. We list these restrictions below.

Restrictions of the Git-API

1. The **1000 results** per search-query are split into pages with a maximum of a 100 results per page. The page number and the number of results on a page can be passed as parameters via the API.
2. The **rate limit** is capped at 30 requests per minute if one is authenticated. To get authenticated, one needs to have a GitHub account and create a token. An explanation of how to create an authentication token can be found at: `https://docs.github.com/en/rest/guides/basics-of-authentication?apiVersion=2022-11-28`
3. The **timeout** can result in less then 100 results per page being returned if the query takes a long time to process. This can be avoided by lowering the results per page below 100 but this also increases the overall search time as the secondary rate limit appears to be hit more frequently.
4. The **secondary rate limit** is an important feature to keep the API from being abused by limiting aggressive polling and compute-intensive tasks [7]. In our case it is something we frequently encountered as the code-search constitutes a compute-intensive poll and is therefore constrained by the secondary rate limit.

A more detailed explanation of the inner workings of the Search-API is available at `https://docs.github.com/en/rest/search?apiVersion=2022-11-28`.

1.2.2 SoftWipe

SoftWipe [30] is an open source tool and benchmark to assess, rate, and review scientific software written in C or C++ with respect to coding standard adherence. The coding standard adherence is assessed using a set of static and dynamic code analysers such as Lizard (`https://github.com/terryyin/lizard`) or the Clang address sanitiser (`https://clang.llvm.org/`). It returns a score between 0 (low adherence) and 10 (good adherence). In order to simplify our experimental setup, we excluded the compilation warnings, which require a difficult to automate compilation of the assessed software, from the analysis using the `--exclude-compilation` option.

1.2.3 Regular Expressions and Automata

Definitions

- a finite *alphabet* Σ is a finite set of symbols
- a finite sequence of symbols $e \in \Sigma$ is defined as a *word* w
- two words w_1 and w_2 can be concatenated to $w_3 = w_1 \cdot w_2 = w_1 w_2$
- w^i is defined as concatenating w with itself i-times
- the empty word is ε and part of every alphabet Σ and $w^0 = \varepsilon$
- $*$ is called the *Kleene closure* is the set of all possible concatenations, e.g., $\{\varepsilon, w, w^2, ...\}$
- a *formal language* L over the alphabet Σ is defined as : $L \subseteq \Sigma^*$

Deterministic Finite Automaton: A Deterministic Finite Automaton (henceforth DFA) is defined as follows: For a finite alphabet Σ a DFA is a quintuple $(Q, \Sigma, q_0, F, \delta)$ where:

- Q is a finite set of states
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of end states
- δ is a function $\delta : Q \times \Sigma \longrightarrow Q$

which is usually represented as a directed graph where $q_i \in Q$ is a node and $\delta : Q \times \Sigma \longrightarrow Q$ are labeled edges between nodes. [20]

Nondeterministic Finite Automaton: A Nondeterministic Finite Automaton (henceforth NFA) is defined like a DFA with the only difference that δ is now defined as $\delta : Q \times \Sigma \longrightarrow 2^Q$, which means that one state and a letter can now transition into multiple different states (a subset of the Q power set) as well as the null/epsilon transition being allowed, meaning that we can jump from state a to state b without reading a character. [20]

Regular Language: A language $L \subseteq \Sigma^*$ is called *regular* if one of the following is true (inductive definition):

- Base case:
  - $L = \{a\}$ with $a \in \Sigma$
  - $L = \{\varepsilon\}$
  - $L = \emptyset$
- Induction: There are regular languages L_1, L_2 so that:
  - $L = L_1 \cdot L_2$
  - $L = L_1 \cup L_2$
  - $L = L_1^*$

[20] and [26].

Regular Expression: A regular expression (henceforth regex) is defined over an alphabet Σ as an expression e of the form: $e ::= \emptyset \mid a \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^*$ for any $a \in \Sigma$. Where brackets are allowed to denote groups e.g. $(a \cdot b)^*$. Where:

- $\emptyset$ is the empty set
- $e_1 \cdot e_2$ is the concatenation of two regular expressions, meaning a logical `and` as both e_1 and e_2 need to be true.
- $e_1 + e_2$ is equivalent to a logical `or` as either e_1 and e_2 need to be true.
- e^* is the *Kleene closure* with the same definition as given above, with the only difference being that instead of words you have another regex.

[20] In reality there are a lot more defined meta-characters, such as $[a - z]$ denotes the regex that accepts all characters of the alphabet. A list of such meta-characters in practise can be found at: `https://github.com/google/re2/wiki/Syntax`.

Equivalence of Regular Expressions and DFAs: The statement:
regular language $\Leftrightarrow$ regex $\Leftrightarrow$ NFA $\Leftrightarrow$ DFA,
is true and proven. The proof can be found in [20]. It is important to note that in order to transform a NFA to a DFA usually a power set construction is used, which increases the amount of states from $|Q|$ original states of the NFA to $2^{|Q|}$ states for the DFA.

1.2.4 General Setup

Our study only considers open source code written in the C programming language and found on GitHub. While it is technically feasible to extend this to C++ using the same crawling and data analysis scripts, we decided to disregard C++ code for the sake of simplicity and due to time constraints. However, we are agnostic as to whether C++ programmers have the same mindset as C programmers or are politically more correct. Therefore, we are cautious with respect to generalising our findings to the object-oriented programming community.

As outlined in the abstract, we only focused on English swearwords. For the swearwords we used the *swearword dictionary* (`https://www.noswearing.com/fulldict.xml`) as reference, which contains over 300 English swearwords.

For data crawling and analysis, we developed a Python program that runs on an external server with 64GB RAM and 12 cores and 500GB storage space using Linux (CentOS). The program itself is divided into a data crawling module and a data evaluation module. The data crawling module consists of two scripts, one for searching the repositories (`search_repos`) and the second for downloading them (`download_repos`). The data evaluation module comprises the `evaluator_parallel` script.

2 Data Gathering, Evaluation and Analysis Methods

2.1 Data Crawling and Evaluation Approach

This section covers the methods used to gather/crawl and evaluate the source code data as well as the restrictions and optimisations we deployed.

2.1.1 Why we chose the Git-API

Our initial idea was to deploy web scrapping using `beautiful soup`, which is a "library for pulling data out of HTML and XML files" [2]. With this library, we planned to build a web scrapper which was supposed to directly search for repositories with swearwords on the GitHub web page. This idea was quickly discarded as we took a closer look at the Git-API and realised that it provided all the functionality we needed. Web scraping has the advantage of imposing fewer restrictions than the Git-API, namely the maximum of 1000 results per search query (1), and the rate limit (2). However, the Git-API is faster and easier to use. To use this API, some form of URL library like the requests library is necessary to execute the URL queries.

2.1.2 Source Code Repository Crawling

This section covers the basic structure of the implementation of the aforementioned data crawling module of our Python program (i.e., the `search_repos` and `download_repos` scripts). More details are provided in the following sections.

Definitions: We distinguish between two fundamental types of repositories:

- *swear-repos* which are repositories obtained through the Git-API code search function and contain swearwords.
- *star-repos* which are repositories obtained through the Git-API repository search and are rated 4 stars or higher. Note that these repositories can also contain swearwords.

2.1.2.1 Repository Filtering

Before outlining the crawling details, we initially describe the two universal repository filtering criteria we deploy. First, we deploy a size limit of 625MB for repositories. Second, we limit the per-repository execution time of our coding standards adherence tool SoftWipe

[30] to one hour. Note that SoftWipe needs to execute several code analysis tools and hence execution times might become prohibitive for the more than 1000 repositories we finally analyzed. More details will be outlined in the evaluation Section 2.1.3 further below.

Rationale for size limit: During preliminary experiments we noticed that SoftWipe requires excessive execution times on big repositories. To address this issue, we explored the 95 percentile with respect to repository size of the first 800 downloaded repositories containing swearwords, which approximately amounts to 625MB. This was subsequently used as a size restriction for swear- and star-repos. This limit can be enforced directly via the URL query for star-repos but only indirectly, after the actual download for swear-repos since there is no repository size parameter in the code-search query.

Quality filtering: For the star-repos, we only consider repositories evaluated with four stars or more. This restriction was put into place to filter out barely known or used repositories and to only consider repositories of presumably higher quality as a reference for comparison. Note that stars, as mentioned in Section 1.2.1, do not represent a direct measure of the quality of a repository. Instead, they indicate that users found the repository helpful and/or interesting. However, one can assume that repositories which are of interest to people and are more widely used at least exhibit a decent level of code quality. The 4 star boundary was chosen arbitrarily. The rationale behind that was only based on the assumption that repositories are most likely pareto distributed according to stars and quality, meaning that even excluding repositories with only a small amount of stars will exclude most GitHub repositories and yield more high-quality repositories.

2.1.2.2 General crawling implementation

For a visual representation of the data crawling for swear-repos refer to Figure 2.1 which will be explained in this chapter.
For each Git-API search, a URL needs to be constructed, which is subsequently passed as a parameter to the aforementioned requests library. The requests library then returns either a requests object or an error code.
For every repository we store its name, number of stars, original URL, download URL and the authors' name. This is stored in a JSON file named after the search criterion and the programming language, meaning in the case of the swearword search after the swearword (e.g. `fuck_C.json`) or in the case of the repository-search after the star interval. Repository download works as a search by running the previously extracted download URL with the requests library. This returns a zip- or a tarball which is extracted and saved on the current machine, in our case the server we used. As we only search for repositories containing a single swearword at a time, a repository containing two swearwords such as `shit` and `fuck` might be found twice. For this reason, a check is required to verify that the repository has not already been downloaded previously. After the download is fished the "crawling" module terminates and the evaluation module is invoked (see Chapter 2.1.3).

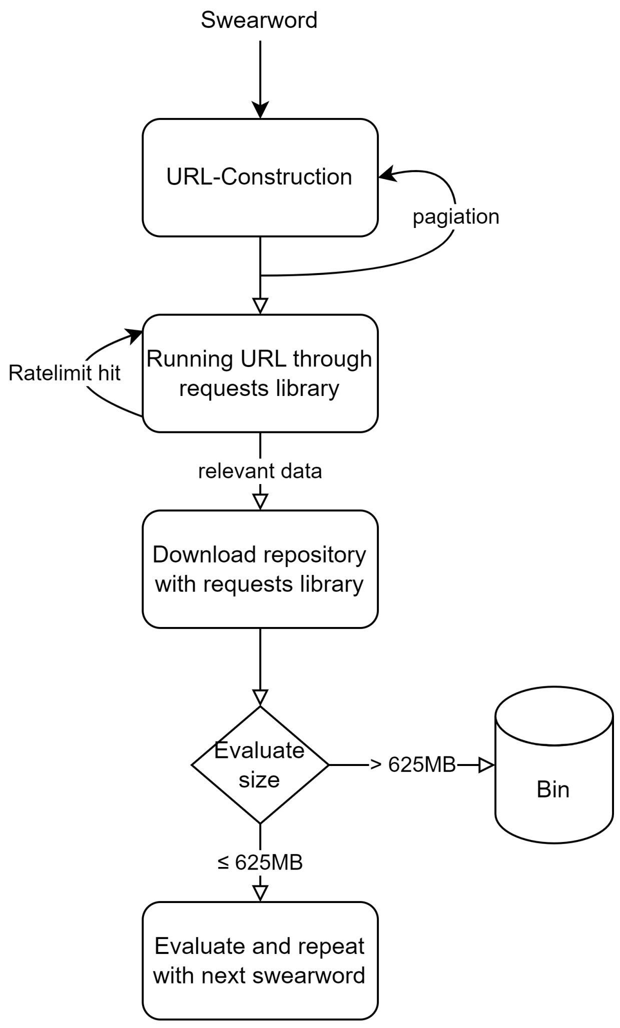

Swearword
URL-Construction
pagiation
Ratelimit hit
Running URL through requests library
relevant data
Download repository with requests library
Evaluate size
> 625MB
Bin
≤ 625MB
Evaluate and repeat with next swearword

Figure 2.1: Flowchart of the data crawling

2.1.2.3 API search

In general, the search is executed by running the constructed search URL with the requests library. The response of the Git-API will in general be structured like a python dictionary, which stores unordered key-value pairs [21]. This response contains information such as the name of the repository and author as well as the download URL. Some more meta information such as the number of stars are also passed as a URL and need to be queried for separately by passing the URL to the response library again. For the complete response scheme, please refer to `https://docs.github.com/de/rest/search?apiVersion=2022-11-28#search-code`. The possible search functions of the API are the following:

- Search-code
- Search-commits
- Search-issues-and-pull-requests
- Search-repositories
- Search-topics
- Search-users

[9]

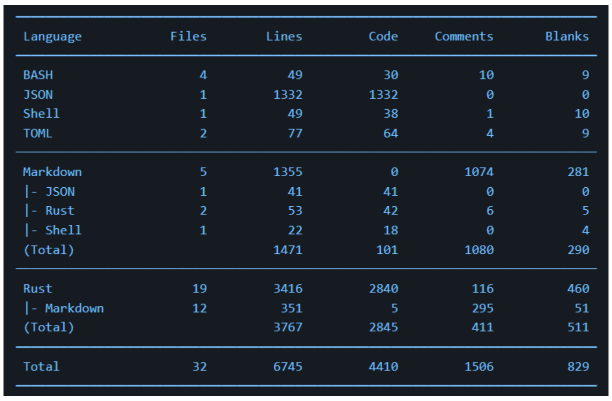
URL construction To identify swear-repos, we conduct a search for swearwords in the dictionary over the source code of the repository. Parameters that can be passed here are the following:

- A string (swearword) the code needs to contain. This is the only parameter that needs to be set, as the remaining parameters are optional.
- The programming language in which the file containing the string is written.
- The page and the amount of results per page.
- And the sort order (ascending, descending), in regards to how recently the GitHub search system has indexed a file. [9]

For star-repos we conduct a repository search without any key words. However, we do use a star interval to only select repositories rated with four or more stars. It is important to give a star interval and not just to search for repositories with the flag "stars > 3" because of the maximum of 1000 results per query. Other parameters in addition to the stars and the repository name that can be set are the repository size and the aforementioned parameters for page dimensions and sorting.

2.1.3 Repository Evaluation

This section will cover the functionality of the `evaluator_parallel` script of our program as well as the improvements we made to that process.



```
Language          Files     Lines     Code    Comments    Blanks

BASH                  4        49       30          10         9
JSON                  1      1332     1332           0         0
Shell                 1        49       38           1        10
TOML                  2        77       64           4         9

Markdown              5      1355        0        1074       281
|- JSON               1        41       41           0         0
|- Rust               2        53       42           6         5
|- Shell              1        22       18           0         4
(Total)                      1471      101        1080       290

Rust                 19      3416     2840         116       460
|- Markdown          12       351        5         295        51
(Total)                      3767     2845         411       511

Total                32      6745     4410        1506       829
```

Figure 2.2: Example output of `tokei` [6]

2.1.3.1 Repository Analysis

Our evaluation starts by counting the lines of code (LoC) in a repository using `tokei` which is a very fast tool to count the lines of code sorted by language. An example output of `tokei` is provided in Figure 2.2. In our case we are only interested in the value for C code. Then, we count and also verify again for the sake of correctness that the repository contains swearwords using a regular expression. If the presence of swearwords is not verified for a repo labelled as a swear-repo, it is discarded. The last step in our verification and evaluation pipeline is the execution of the code quality adherence test with SoftWipe to obtain a respective code quality score. This is the data analysis step which required the largest fraction of computing time. Finally, we calculate the *swear factor* as the number of swearwords divided by the lines of code and save all the relevant data in a file. That corresponds to one result file per repository, which has the benefit of avoiding potential concurrent file access issues with the parallelisation. After the evaluation is completed, the repository is compressed using `gzip` to save storage space. A visual representation of this process is provided in Figure 2.3.

Counting Swearwords A key question to be addressed is how to efficiently search millions of lines of code for over 300 swearwords without any false positives. A general rule for this is that we prefer to underestimate the number of swearwords in the repository rather then overestimating them since this process is used to confirm the presence of swearwords in the code. This might appear unnecessary since we already know that there swearwords should be present in a swear-repo due to the git code search. However, for the sake of verification, we check again for the presence of swearwords to rule out any programming mistakes. For this we initially tested the `regex` function of the standard `re` library which is relatively fast and under the condition that the regular expression is properly defined does not yield any false positives. This was later found to constitute a computational bottleneck and was speed up by using the `re2` library. For more information on this, refer to Section 2.1.3.2. This raises the general question as to what we should consider as being a swearword in the source code. We consider the following three cases specified by

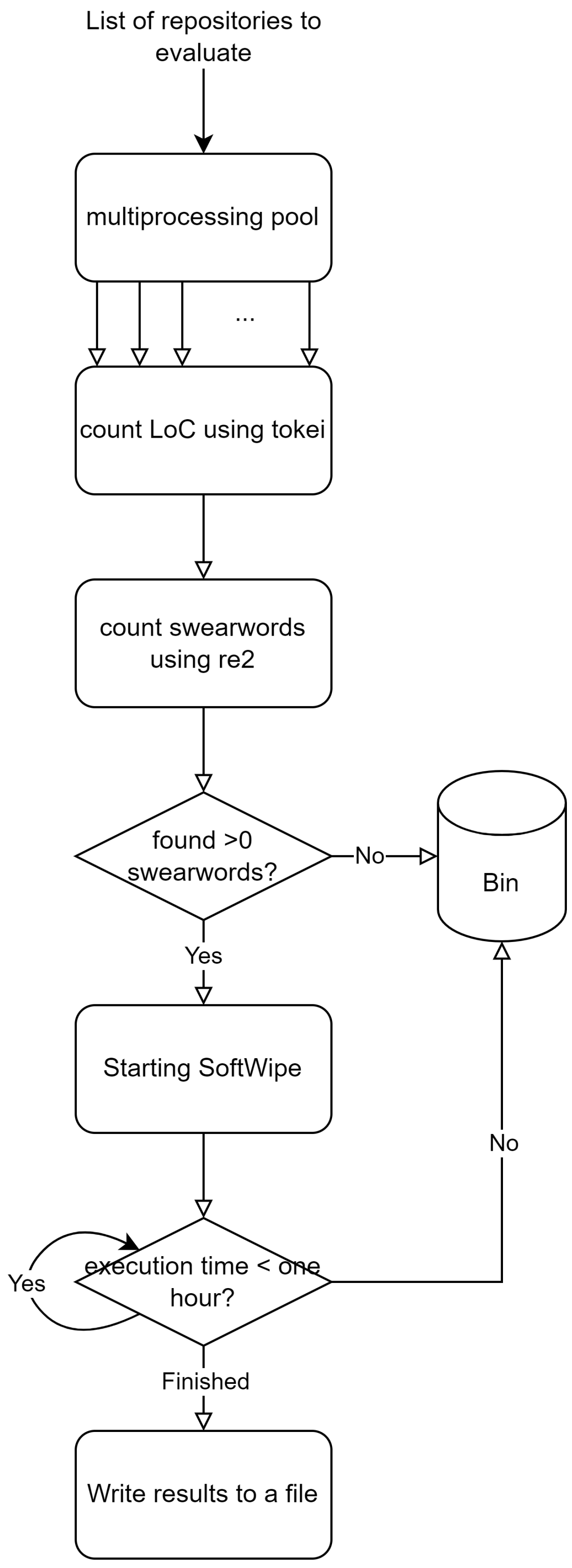


Figure 2.3: Flowchart of the evaluation process

regular expressions (regex) as swearwords. Below you will find the regex used to identify swearwords that is split into three parts. We use the following notation:

- `swearword` is a variable for the swearword in lower case
- `swearword_first_cap` is a variable for the swearword with the first letter capitalised (caps)
- `swearword_caps` is a variable for the swearword with all letters capitalised
- the white spaces are included here for the sake of readability but are not contained in the actual regular expression as this would obviously affect the results
- `S*` is an arbitrary string which can be empty
- `\b` denotes the start and, in the second occurrence, the end of the word
- `(-|_|[0-9])` this is the group of allowed separating characters sometimes supplemented by `[A-Z]`. One can argue that we should allow substantially more special characters, such as `*` or `+` as separating characters. However, these are not only uncommon, but they might dilute the context. With the current small set of separating characters, we can assume that the programmers intention was to swear.

The entire regex is as follows:

1. `\b\S*(-|_|[0-9]) swearword ((-|_|[A-Z]|[0-9])\S*)?\b|`
2. `\b\S*(-|_|[0-9]))? swearword_first_cap ((-|_|[A-Z]|[0-9])\S*)?\b|`
3. `\b(\S*(-|_|[0-9]))? swearword_caps ((-|_|[0-9])\S*)?\b`

The explanation for cases 1 to 3 is as follows:

1. This captures the swearword itself as well as `camelCase` with the swearword in the beginning and `snake_case`. Of course, there are also cases outside of these styles, such as `And_fuckThis1example`. While this is stylistically horrible this should obviously be a hit as well. More examples for the swearword `fuck` are:
   - `what_the_fuck`
   - `fuck10`
   - `fuckThis`
2. This case is mostly intended to capture the swearword itself and the `camelCase` where the swearword is in the middle. Of course this detects some stylistic oddities as well. Examples for the swearword `fuck` are:
   - `whatTheFuck`
   - `Fuck`
   - `this_Fuck-ingOddity`

3. This third and last case predominantly serves to capture the swearword in caps and the odd `snake_case` where the swearword is in caps. This is probably the least likely case to occur. Examples if the swearword is `fuck` are:
   - `WHAT_THE_FUCK`
   - `FUCK`
   - `FUCK_MY-badExamples`

All three cases are connected with an `or` expressed as | in the regex. Thus, we receive one regex per swearword. A debatable example regarding the definition of the above regular expression is that `bosshitpoints` will not be identified as swearword, but `bosShitPoints` will. However, our rationale is that the latter is intended by the programmer and should therefore count as swearing.
The disadvantage of using regular expressions is obviously that the broader swearing context is being ignored as are swearwords outside the ones defined in our dictionary. Also, without context or the inclusion of English grammar, swearwords such as `zeroFucksGiven` or `this_fucking_sucks` will not be found.

2.1.3.2 Runtime Bottlenecks and their Optimisation

As mentioned previously, the two largest performance bottlenecks are the execution times of SoftWipe and the identification and counting of swearwords using regular expressions. In the following, we will explain why they affect runtimes and how we alleviated the performance issues.

Parallelisation: We implemented two optimisations to accelerate the analyses. First, we parallelised the entire evaluation process in a straightforward way since the evaluations of individual repositories are independent from each other. In the first version of the program, the evaluation depended on a single result file, but this was changed so that the evaluation module produces one output file per repository. This was done to avoid concurrent writing file accesses.
The task-level parallelisation over repositories was implemented by deploying the standard python multiprocessing library, which allows the user to create and concurrently execute multiple processes. The documentation can be found at `https://docs.python.org/3/library/multiprocessing.html`.
The parallelisation yielded an approximately 6-fold run-time improvement on 6 cores compared to 1 core. As SoftWipe itself is also already parallelised, only half of the available cores on the server were used.

Re2 library: The second optimisation was to replace the standard `re` library for regular expressions by the `re2` library developed by Google. For more information regarding the `re2` library please refer to `https://github.com/google/re2`. The `re2` library guarantees execution in linear time, as it creates a DFA for the regular expression. As we have seen in the Section 1.2.3 every regular expression can be translated into a NFA which again can be translated into a DFA. However, this translation from NFA to DFA adds $\leq 2^Q$ new states if

Q is the set of transitions of the NFA. This is the reason why we could not combine the swearword regular expressions to one regex, since that led to an error. For a more detailed explanation of this and how it works in regards to `re2` specifically, please refer to [22]. As mentioned in the previous chapter 2.1.3.1 we currently have one regex per swearword. We could further concatenate those regex with an `or` and end up with a large regex for all swearwords. However, while this is theoretically possible, this regex is too large for the `re2` library and cannot be compiled into a DFA due to the previously mentioned $\leq 2^Q$ new states. Thus, we keep every swearword in a separate regex. To see the difference of `re` compared to `re2` we implemented a benchmark, which counts two repositories. The resulting time was 579 seconds for the `re` library compared to only 8 seconds for the `re2` library. One could try to further improve the performance by connecting some swearwords to one regex. However, the performance increase was substantial enough that further optimisations where deemed unnecessary as SoftWipe is still the biggest bottleneck.

2.1.4 Error Handling

This section will focus on possible errors that can occur during the execution of the program and how we dealt with them. It covers the re-entry mechanism which allows us to restart the program at the same point it was before the stop, as well as the errors that can occur during the data crawling or the evaluation.

2.1.4.1 Re-entry points of the program

To prevent the loss of progress if the program comes to a preemptive halt, a start-stop mechanism was implemented. There are 3 critical points in the program:

1. After the search
2. After the downloading
3. After the evaluation

After reaching each of those critical points we write the progress to a file. The re-entry works by reading the file and then skipping the steps (1 to 3) until we reach the last entry point and continue from there. The specifics of re-entry depend on the step we were on previously and are listed below:

1. We simply re-run the search query.
2. This is tied into our mechanism, which looks at whether a repository has been downloaded before. This way we ensure that the program continues exactly where it left of.
3. As each process writes a separate file after successfully executing SoftWipe we just check if for the repositories we need to evaluate a result file exists.

2.1.4.2 Data crawling errors

While data crawling we mainly need to focus on errors occurring while using the search function of the Git-API. Especially the primary and secondary rate limits that were mentioned before in Section 1.2.1 are things we need to watch out for. If either is hit, we wait for a few seconds and repeat the query. Errors that just limit the amount of possible repositories we get are ignored (e.g., the time limit of the Git-API or errors while downloading), as we just need data and do not need precision here.

2.1.4.3 Evaluation errors

As a general rule if an error occurs somewhere in the evaluation step the repository is discarded. When evaluating, there are three main types of errors we need to look for. The first are errors occurring while using *tokei*, where we only confirm whether the repository contains C code. The second possible error is specific to the *swear-repos*, as we need to confirm the presence of swearwords in the repository. The last type of error are errors that can occur while running SoftWipe. This can be SoftWipe terminating without returning a score or the imposed time limit of one hour terminated SoftWipe. In either case, the general rule applies and the repository is discarded.

2.2 Data Analysis

This section will go over and explain the tests we used, as well as discuss the results we have obtained. To implement statistical tests, the `scipy` library was used in conjunction with `numpy`. The documentation for the statistical functions of the `scipy` library can be found at `https://docs.scipy.org/doc/scipy/reference/stats.html`. We used `pandas` (`https://pandas.pydata.org`) for the data management and csv import. The `matplotlib` (`https://matplotlib.org`) library was used for the visualisation of the data in a `jupyter notebook` (`https://jupyter.org`).

2.2.1 Definitions

1. A *population* is a discrete group of things that can be identified by at least one common denominator. One usually only has a sample of said population which is used in a variety of tests to draw conclusions for the entire populations. [27]

2. The *general population* is in our context the set of all open-source repositories on GitHub and comprises repositories that are somewhat popular and do not exceed a certain size. Our set of star-repos is a sample of this general population.

3. The *target population* is the set of all open-source repositories on GitHub that contain swearwords where again our swear-repos are a sample of said population.

4. The *null hypothesis* usually denoted H_0 is the hypothesis which a statistical test is designed to test. [19]

5. The *alternative hypothesis* is the opposite hypothesis of H_0 and is accepted if the test rejects H_0.[19]
6. A *statistic* is a random variable, which can be calculated after observing the sample data from the sample data an example for such a statistic is the sample mean $\bar{X}$. [24]
7. A *point estimate* of a parameter θ of the population is a single number that is close to the true value of θ. An example for such a parameter θ is the population mean μ, which can be approximated by calculating the sample mean $\bar{X}$. This sample mean is then called a *point estimator* of θ. Such a point estimator can be any suitable statistic. [24]
8. An *empirical distribution function* of a sample simply takes the original sample and assigns each value $x_1, x_2, ..., x_n$ the probability $1/n$. [25]

2.2.2 Data Analysis Methods

To analyse the gathered data we deployed several different visual and statistical tests.

Defining our goal: We have obtained two large samples by running our program. With those two samples we want to achieve two main things:

1. We want to draw a conclusion from the sample to the underlying population. Meaning we consider both samples separately and draw conclusions about the population based on that single sample.
2. We want to draw a conclusion on the relationship between the target and the general population by comparing the two samples with each other.

This is done to determine if swear-repos do have a higher/lower code quality than the general population.

To justify and explain what we are doing, we give a brief introduction to the relevant statistical theory.

2.2.2.1 Central limit theorem

The central limit theorem states "that the sum of a sufficiently large number of independent identically distributed random variables approximately follows a normal distribution." [3]. This especially means that each statistic of the form $S_n = \sum_{i=0}^{n} \Psi(X_i)/n$ will asymptotically converge to a normal distribution if the mean and variance are finite [14]. If $\Psi(x) = x$ the expression S_n is the mean, which will be relevant later on.

2.2.2.2 Statistical tests based on a single sample

For a sample it is fairly easy to calculate a *point estimator* e.g. the sample mean $\bar{X}$. But there is no indication of the accuracy of this estimation. Something more sensible than a point estimator is a *confidence interval*, which returns an interval of plausible values for the statistic. Such a confidence interval is calculated with a confidence level, which is commonly set at 90%, 95% or 99%. Contrary to popular belief, this confidence value does not imply that the true value of, e.g., the population mean μ is in the 95% confidence interval with a probability of 95%. What it does mean is given an arbitrary not too small amount of samples, 95% of those samples give an interval that contains μ. The accuracy of the confidence interval can be seen by its width, which means that large confidence intervals are less meaningful compared to small confidence intervals. The problem with confidence intervals is that there are two requirements to calculate them:

1. The population has to be normally distributed
2. The true value of the population standard deviation is known.

Especially the second requirement is rather unrealistic in a real scenario. However given a large sample size one can ignore both requirements and substitute the population standard deviation for the sample standard deviation, a proof of which can be found in [24]. One method to use confidence intervals despite not having a large sample size is the bootstrap.

2.2.2.3 Bootstrapping

Bootstrapping is a re-sampling method that returns measures of accuracy for a given sample statistic. An example for these measures of accuracy is a confidence interval and a standard error, and an example for the test statistic could be the mean or the variance. It can also be used to approximate the sampling distribution. Two of the main advantages are that it is a rather straightforward process and that it does not assume any underlying distributions. Since a larger sample decreases the variance of an estimator (proof in [25]) and bootstrapping artificially increases the sample size, it generally helps with the estimation. For us, it is a great tool to approximate population parameters.

The Idea behind bootstrapping: It is generally done by re-sampling the original sample with replacement, meaning that it creates a new sample from the empirical distribution function [16]. More concretely for the given point estimator $\hat{\theta}$ we generate a new sample by drawing from the empirical distribution function of the original sample and calculate a point estimate of that newly generated sample. This processes is then repeated an arbitrary amount of times, commonly 999 or 9999 depending on computing power. This results in returning an estimated distribution function of $\hat{\theta}$ referred to as $F_{\hat{\theta}}$. We can then look at statistics for $F_{\hat{\theta}}$ to determine the accuracy of our point estimator $\hat{\theta}$. This method is generally known as *non-parametric bootstrap.* [25]

There are a lot of different ways as to how to actually do the bootstrapping that don't necessarily use the empirical distribution function to sample e.g. Monte Carlo algorithm, Bayesian Bootstrap, which will not be discussed here. For more information please refer to [16] or [5]. For the implementation of this method we used the `scipy` library (`https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bootstrap.html`).
[24]

2.2.2.4 Inferences based on two samples

One of the easiest ways to compare two populations with two respective means μ_1 and μ_2 is to compare the means by calculating the difference $\mu_1 - \mu_2$. Given a sample for each of the populations, we can now estimate this difference by replacing the population mean μ_i with the respective sample mean. This results in a new point estimator for which we theoretically could compute a confidence interval. However, a more common way to test this is to create a hypothesis test with H_0 being the assumption that $\mu_1 = \mu_2$. A test that does exactly that is Welch's t-test which will be covered in Section 2.2.2.9

2.2.2.5 Significance testing

Significance tests are used to test a hypothesis based on one or more samples. The main hypothesis we want to test are:

1. Does our *target population* have the same average SoftWipe score compared to the average SoftWipe score of the *general population*?
2. Does the *target population* and the *general population* follow the same distribution?

For a significance test, you need a test statistic $S(X)$, which is a real function of the sample data X, where large values mean that the null hypothesis is in doubt, and smaller values mean that there is no doubt about the null hypothesis. This function $S(X)$ is accompanied by a probability function $P(S(X))$ returns the probability of obtaining the test statistic $S(X)$.

p-value: This paragraph focuses on the p-value and its interpretation and possible misconceptions. The idea behind the p-value is that it gives a "probability of getting a test statistic $T(X)$ larger than or equal to the observed result."[10]. It is a random variable that is uniformly distributed if H_0 is true. It is generally used to reject H_0 if it is smaller than a given significance level α (commonly 0.05 or 0.01). There are many misconceptions about the p-value, the most common being that it is the probability that H_0 is true. Another incorrect assumption is that $1 - p$ is the probability that the alternative hypothesis is true. [18] The p-value has to be treated carefully as a small p-value can in some cases also be the fault of a large sample size and just small deviations from H_0 [24]. For this reason we give the resulting statistic and the p-value as well as a visual approach to the tests to conclude our results.

2.2.2.6 Jarque-Bera test

Our goal with this test is to assess if the data are normally distributed. This test only tests the skewness and kurtosis of the sample and compares them to the skewness and kurtosis of the normal distribution, which are 0 and 3. It works in analogy to the aforementioned statistical tests by calculating a statistic JB. If this test statistic is sufficiently large, the assumption of normality can be rejected. JB is defined as follows:

$$JB = \frac{N}{6}\left(W^2 + \frac{(K-3)^2}{4}\right) \tag{2.1}$$

where N is the sample size, W is the sample skewness and K the sample kurtosis.[11] For more details, please refer to [12]. Here, we use JB test implementation in `scipy` again, which requires a sample size > 2000 to work. If the sample size is below 2000, we need to use another test, such as the Shapiro-Wilk test which only works for up to 5000 samples.

2.2.2.7 Shapiro-Wilk test

Q-Q plot The Q-Q plot is an informal way to test the normality of a sample. It visualises the quantiles of the ordered observed sample $X_1, ..., X_n$ compared to the theoretical quantiles of the normal distribution [14]. Normality is concluded by how close the data points in the plot are to a linear function, which symbolises where the values of a normal distribution would accumulate.
The Q-Q plot is a subjective tool for assessing normality. However, the idea can be used to implement a more objective hypothesis test. This test is called the Shapiro-Wilk (SW) test, which assesses how well the observed quantiles fit the theoretical ones.[14]. A drawback of this test is that it only works for a sample size of up to 5000. [23] To implement this test the `shapiro` function of the `scipy` library was used. The documentation can be found at: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html`.

2.2.2.8 Kolmogorov-Smirnov test

The Kolmogorov-Smirnov test (KS-test) can be used as a goodness-of-fit test where a random sample is compared to a known distribution and it is possible to evaluate how likely the sample could have been obtained from that distribution. However, there is a second use case that is more relevant in our context. The KS-test can also be used to compare two samples with each other. These two samples, X and Y, can be of different size and from different populations with unknown distributions $F(x)$ and $G(x)$. The KS-test assesses the null hypothesis $H_0 := F(x) = G(x)$ for each x. Hypothesis H_0 is rejected for the significance level α, if $T > t_{n,m,1-\alpha}$, where T is the test result and $t_{n,m,1-\alpha}$ is the value of the Smirnov-table with the parameters n, m and $1 - \alpha$ where n is the sample size of X and m is the sample size of Y. The test T is defined as:

$$T = \sup_x |H_1(x) - H_2(x)|, \tag{2.2}$$

where $H_1(x)$ and $H_2(x)$ are the empirical distributions of the respective samples. [15]

To implement this test the `ks_2samp` function of the `scipy` library was used. The documentation can be found at: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_2samp.html`.

2.2.2.9 Welch's t-test

The Welch's t-test is an approximate solution for the Behrens-Fischer problem, which is defined as follows:

Behrens-Fischer Problem: We want to conclude whether the means of two independent normal populations given two independent samples X_1 and X_2 from these respective populations are equal *without* assuming equal variances. [4]
This problem is solved by assuming equal variance and tested for with the Students t-test. The test statistic of the Welch's t-test is given by:

$$T = \frac{(\bar{X}_1 - \bar{X}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}, \tag{2.3}$$

where n_i is the sample size, S_i is the standard deviation, μ_i is the expected value, and $\bar{X}_i$ is the mean of the arithmetic sample.

Normality assumption: The Welch's t-test assumes normality for the sample means, but not the underlying distribution. This leads to the question of whether this applies to our case as well. The assumption is that they indeed are normally distributed, as is proven by the central limit theorem. However, we did try to test this as well via bootstrapping and the Jarque-Bera test, by generating a high number of bootstrap replicates (e.g., 10,000) and calculating the mean of each of those samples. Then, one applies a goodness-of-fit test such as the Jarque-Bera test, to these bootstrap samples. These resulting distributions of the means where then visualised as a histogram, which you can find in Figure 3.5 and Figure 3.7. This shows that the assumption that the mean is normally distributed is obviously true. This is further proven by the two Q-Q plots of this distribution in Figure 3.6 and Figure 3.8, as well as the Jarque-Bera, which does not reject H_0. Our results of the Jarque-Bera test can be found in the table bellow.

	statistic	p-value
star-repos	0.77	0.68
swear-repos	1.25	0.53

Table 2.1: Table containing the statistic and the p-value of the Jarque-Bera test for the bootstrapped means

The advantages of this test are that different sample sizes for X_1 and X_2 are possible, as well as the only assumption being that the means of the underlying populations are normally distributed. Furthermore, the test is rather robust. [13]

3 Data Analysis Results and Conclusion

3.1 Test results

First, we did perform some visual tests on the data to get a sense of how the distribution looks like. In Figure 3.1 we can see a histogram for the star-repos with an overlayed normal distribution that is calculated from the sample mean and the sample standard deviation. It seems to closely resemble a normal distribution. To test that theory we deployed a Q-Q plot (Figure 3.2) combined with a Jarque-Bera test. The Jarque-Bera test strongly rejects the notion of normality with a p-value $\approx 2 * 10^{-}20$ and a statistic of ≈ 99.6. However, the Q-Q plot suggests that the data is almost normally distributed, especially in the center, which suggests that if we would need to assume normality for this distribution it would most likely not lead to bad results given the test procedure is robust. The plot deviates below 3 or above 9 from the theoretical normal distribution, which can be caused by a lack of information, since the information density on the tails of a distribution is rather low. Now the histogram for the swear-repos in Figure 3.3 and the Q-Q plot of that in Figure 3.4 suggest that the data is likely not normally distributed. A Jarque-Bera and Shapiro-Wilk tests further support this theory by giving p-values way below the 0.01 α significance level. The exact result of the statistic and p-value are: JB := statistic ≈ 90.07, p-value $\approx 2.77 * 10^{-20}$, SW := statistic ≈ 0.98, p-value $\approx 7.28^{-21}$. With the scatter plot of the star-repos (Figure 3.10), which maps the SoftWipe score to the LoC, no significant observation can be made except that there is a lot more data up to the 10.000 LoC mark. The plot itself does not contain data points beyond 70.000 LoC since there are so few of them and they dilute visibility. However, in the scatter plot of swear-repos in Figure 3.9, some data clusters around the SoftWipe score of 8 are observed, which is consistent with the spike in the histogram in Figure 3.3. To rule out that those clusters stem from the same author who writes good software, and also uses swearwords, we deployed a rule that only allows two data points per author. However, this did not affect the clusters and is therefore not the reason for their appearance.
Now we are interested in the mean and respective confidence intervals of the samples. To determine those, we deployed two main techniques per sample:

1. We calculated the arithmetic mean and confidence interval of the sample.

2. We calculate the bootstrap confidence interval of the sample, as well as its standard error.

The significance level was established at 99% and the results can be seen in the table below, rounded to two decimal places:
Both the bootstrapped and the arithmetic confidence intervals were the same and quite

	mean	confidence interval	bootstrapped confidence interval	standard error
star-repos	5.41	[5.38 - 5.45]	[5.38 - 5.45]	0.02
swear-repos	5.87	[5.81 - 5.93]	[5.81 - 5.93]	0.01

Table 3.1: Table containing the arithmetic mean and confidence interval, as well as the bootstrapped confidence interval and standard error of the two samples

narrow, suggesting good precision of our point estimate for the mean. Furthermore there is no overlap in them, which suggests a statistically significant difference between the two means. The reason for using bootstrapping was described in Section 2.2.2.2, which is that confidence intervals need an underlying normal distribution and known population variance to be computed. This should not be an issue for our sample size, since we can approximate the population variance with the sample variance and assume a normal distribution for the mean due to the central limit theorem, but we still used bootstrapping to confirm that our calculation is correct. In Figures 3.5 and 3.6 you can see that the mean, as the central limit theorem suggests, is indeed normally distributed. This is of course also true for the swear-repos, which you can see in Figure 3.7 and 3.8.
The last step was to deploy the hypothesis tests to further confirm our results from the visual tests. To test whether the two distributions are different, we used the KS-test, which resulted in statistic ≈ 0.20 and p-value $\approx 3.17 * 10^{-89}$. This leads us to reject H_0 in favour of the alternative Hypothesis, which is that the two distributions are not the same. Furthermore, we ran the Welch's t-test with H_0 being that the two means are the same. The result of the test where statistic ≈ 16.71 and p-value $\approx 2.04 * 10^{-61}$, which also led us to reject H_0. This concludes our tests and leads us to conclude that there is a correlation between swearing and an improvement in code quality. Obviously correlation does not imply causation, which means that swearing in code does not automatically improve your overall quality.

3.1.1 Interpretation

This study is also an observational study, as we do not control either group. This leads to the problem that although we have a statistically significant difference between the groups, it could be caused by other underlying factors [24].
It is very important to note that small p-values do not guarantee that the results are replicable or that statistical significance implies practical significance [18]. This means that swearing will not automatically improve the quality of your code. However, a study showed that swearing in the workplace acts as a form of stress relief [1], which in turn could then improve focus and therefore code quality. This might be a possible explanation for the findings.

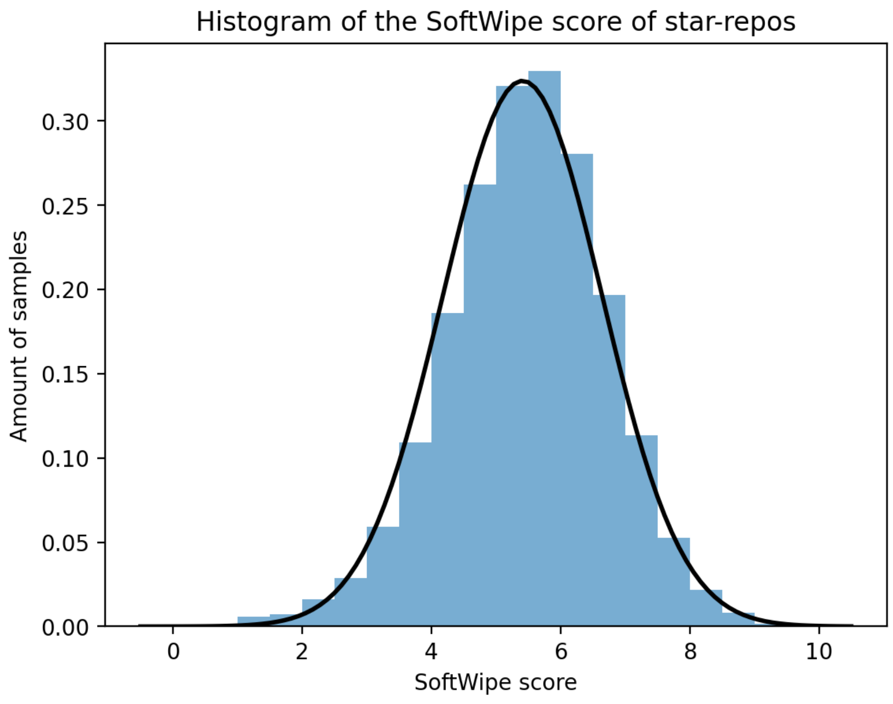


Figure 3.1: The histogram of the SoftWipe scores of star-repos compared to the theoretical normal distribution calculated from the sample mean and sample standard deviation.

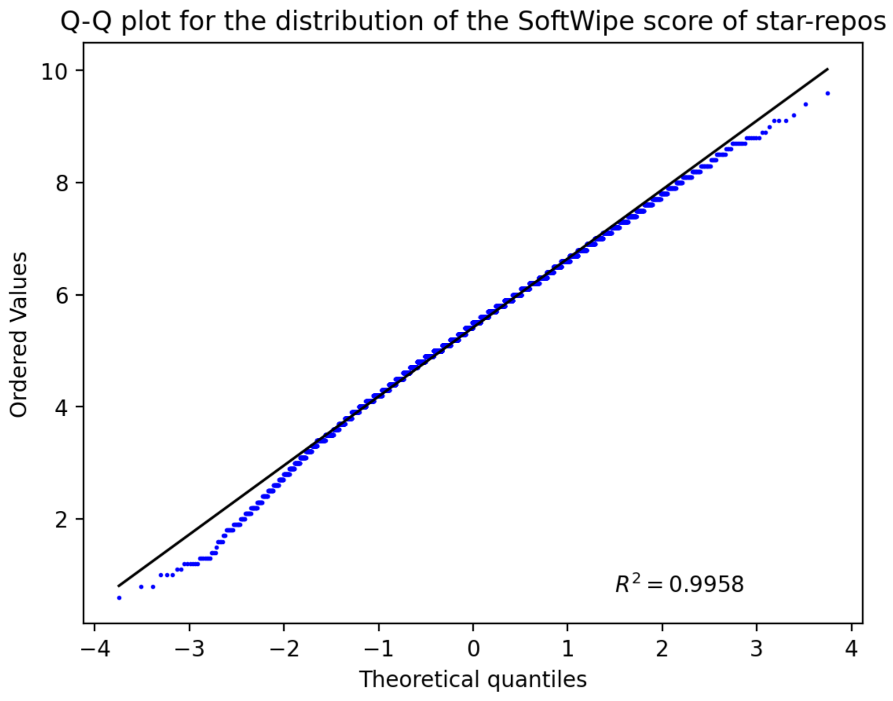


Figure 3.2: A Q-Q plot of the SoftWipe score of star-repos

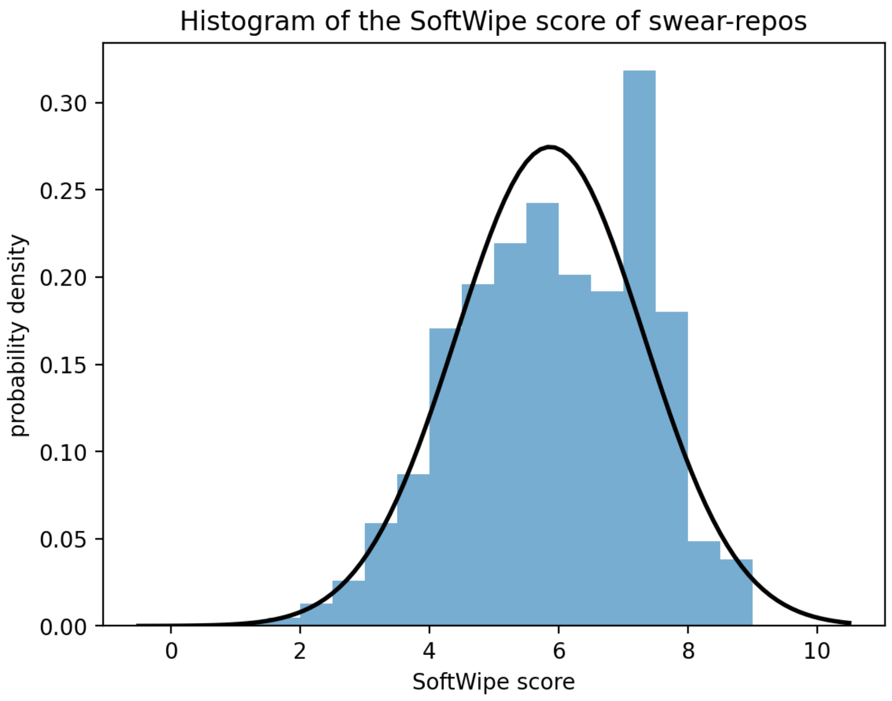


Figure 3.3: Histogram of SoftWipe scores for swear-repos compared to the theoretical normal distribution calculated from the sample mean and sample standard deviation.

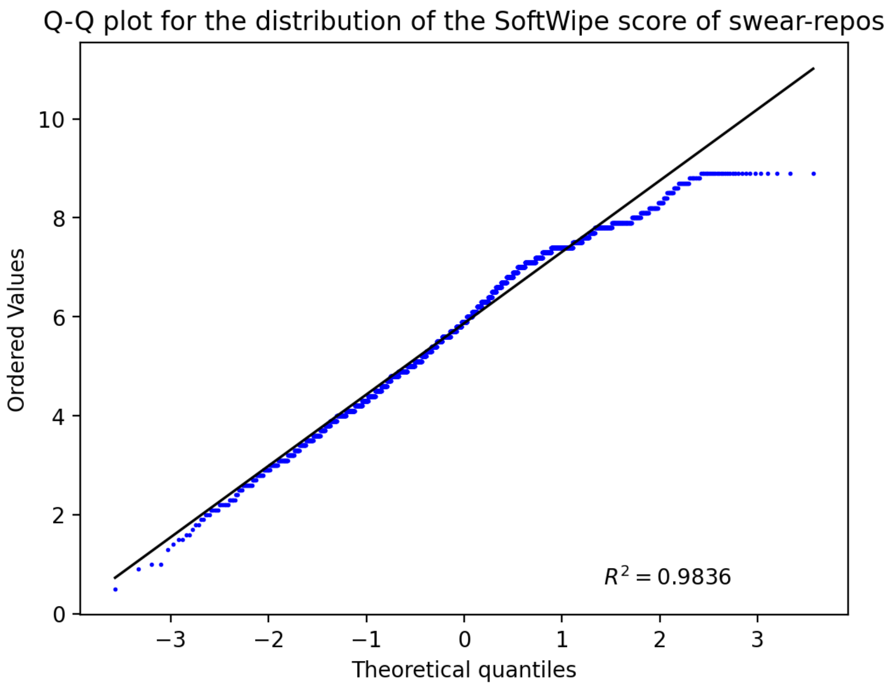


Figure 3.4: A Q-Q plot of the SoftWipe score of swear-repos

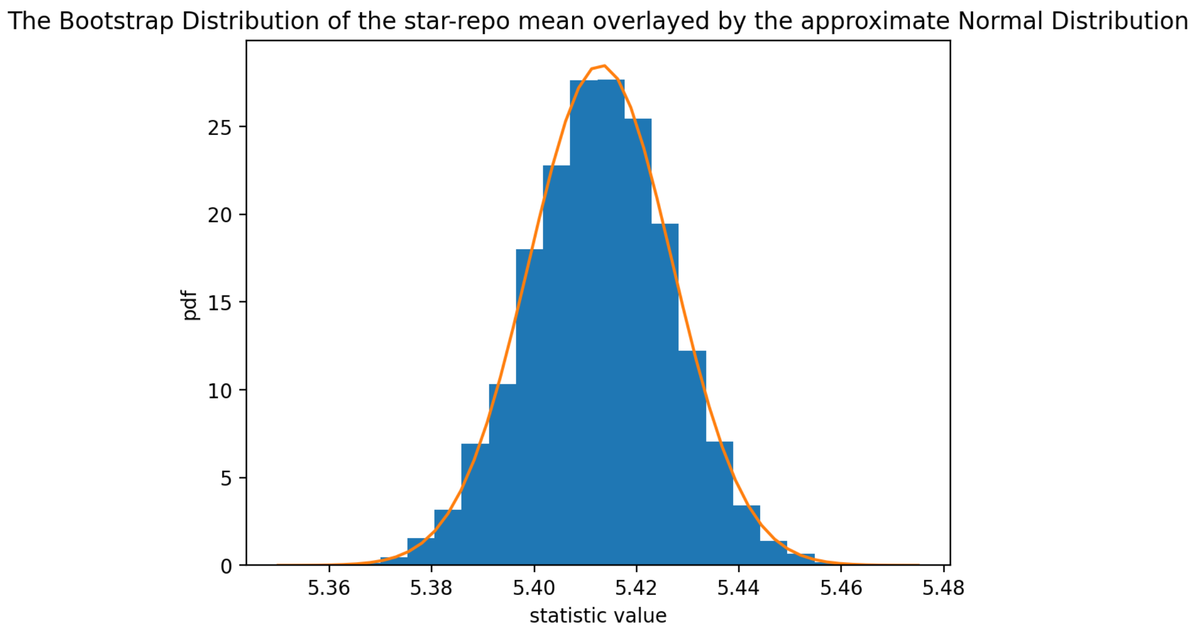


Figure 3.5: The histogram of the bootstrapped mean SoftWipe scores of star-repos compared to the theoretical normal distribution calculated from the bootstrapped sample mean and the bootstrapped sample standard deviation.

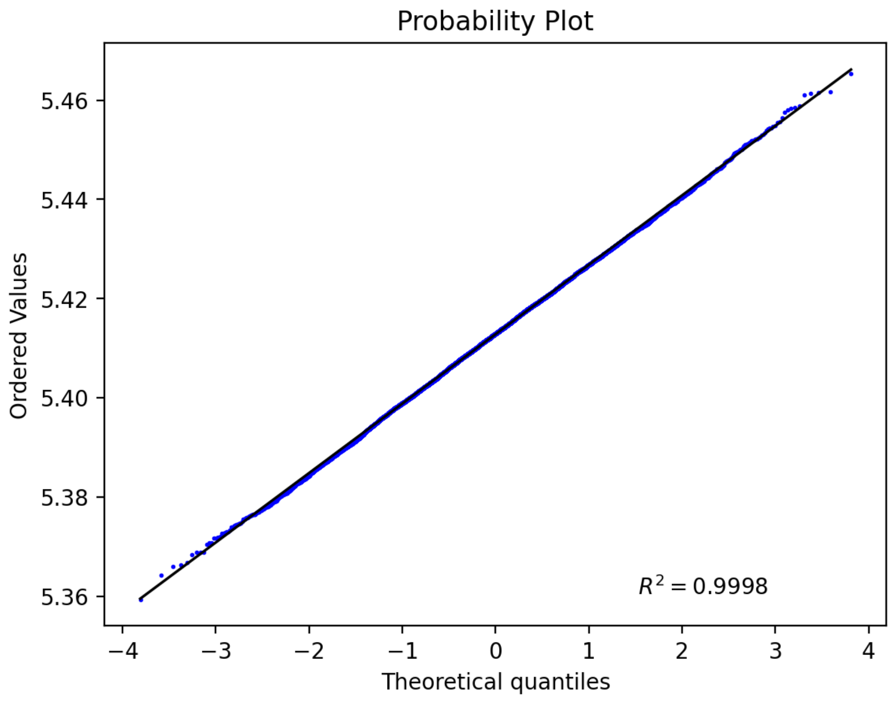


Figure 3.6: A Q-Q plot of the bootstrap sample of star-repos.

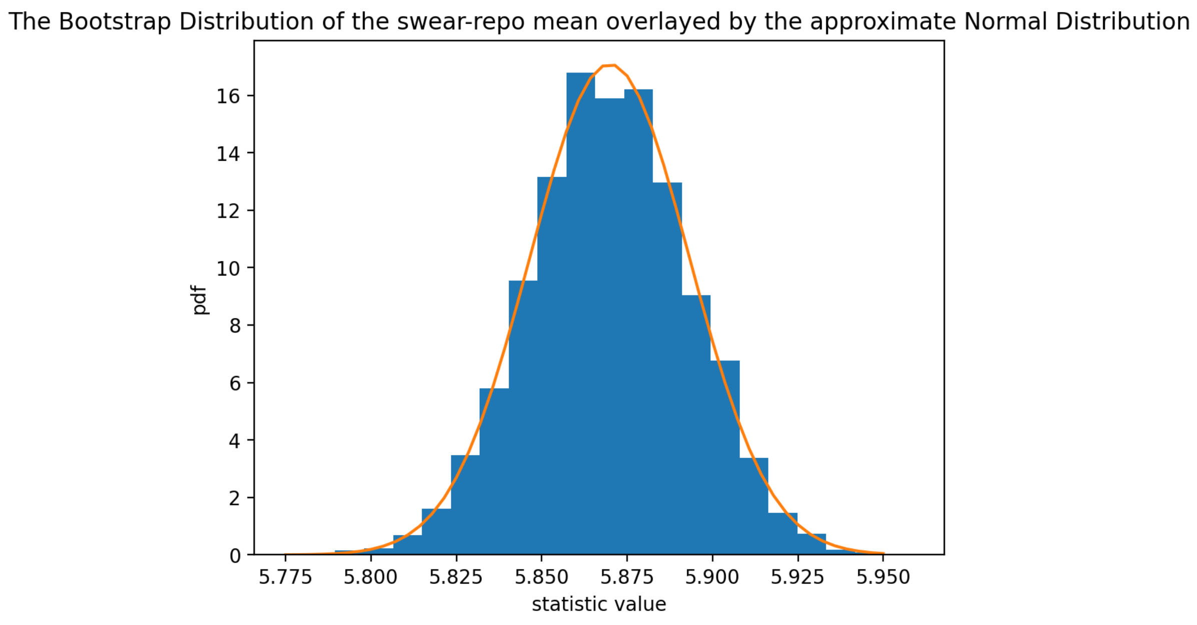


Figure 3.7: The histogram of the bootstrapped mean SoftWipe scores of swear-repos compared to the theoretical normal distribution calculated from the bootstrapped sample mean and the bootstrapped sample standard deviation.

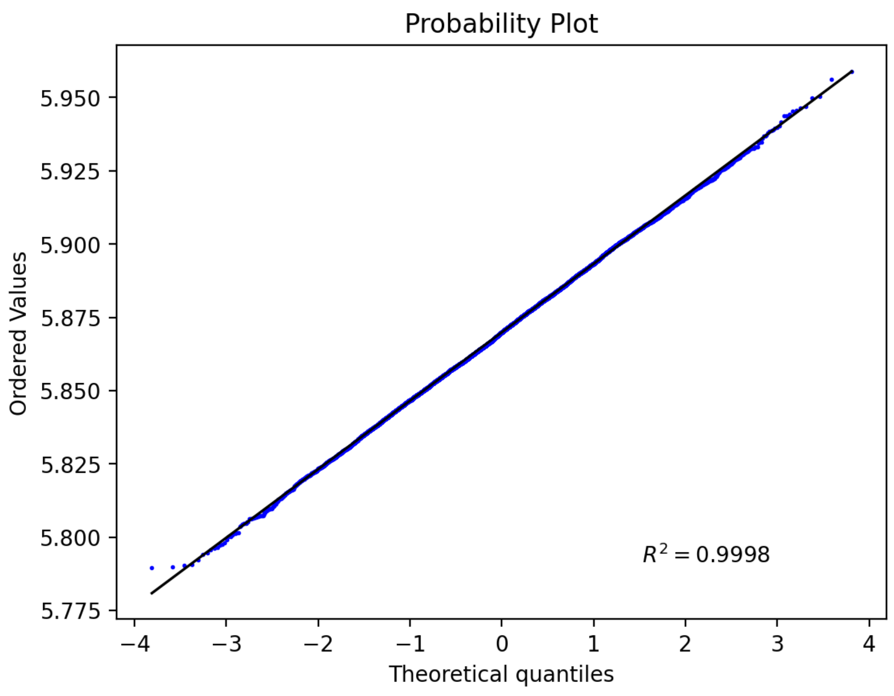


Figure 3.8: A Q-Q plot of the bootstrap sample of swear-repos.

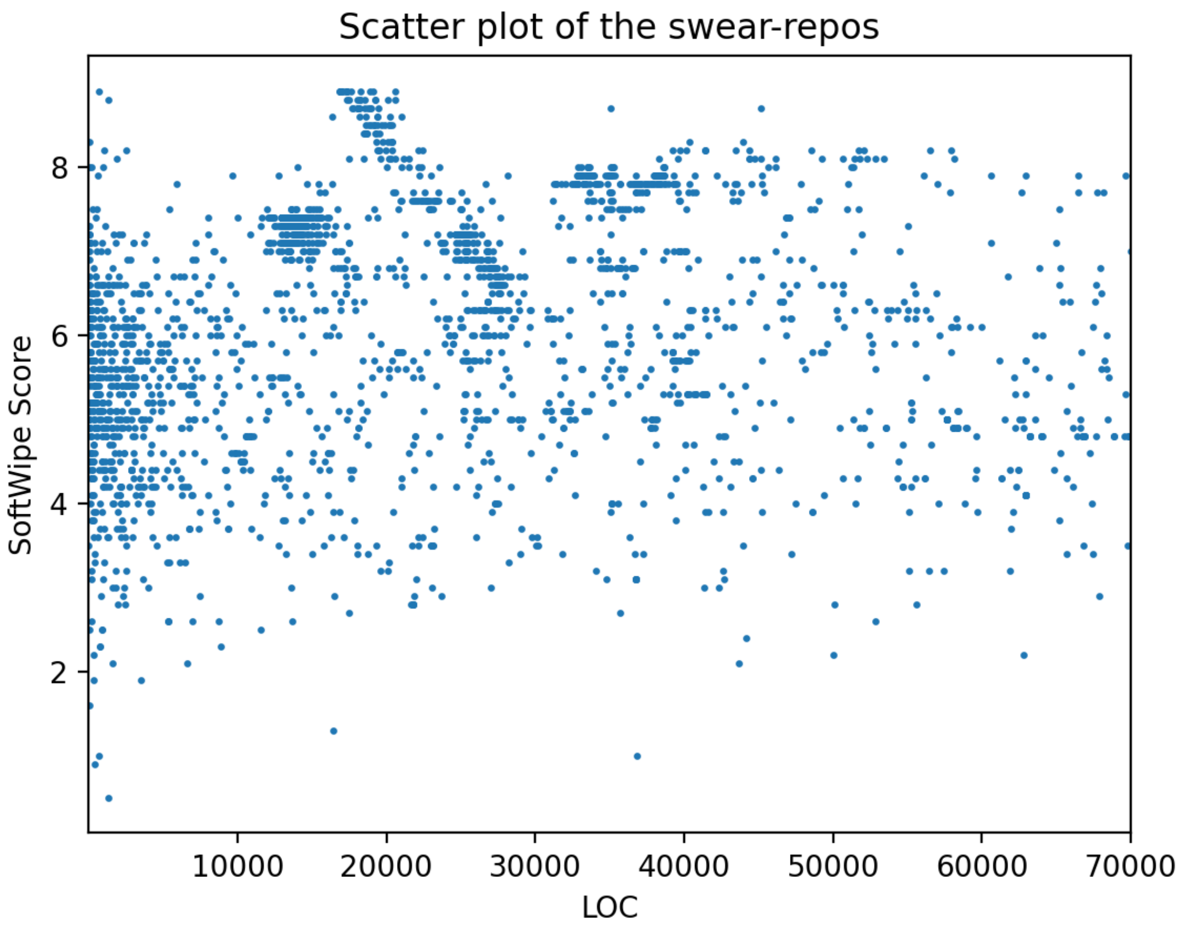


Figure 3.9: Softwipe scores over LoC plot for swear-repos cut of at 70.000 LoC

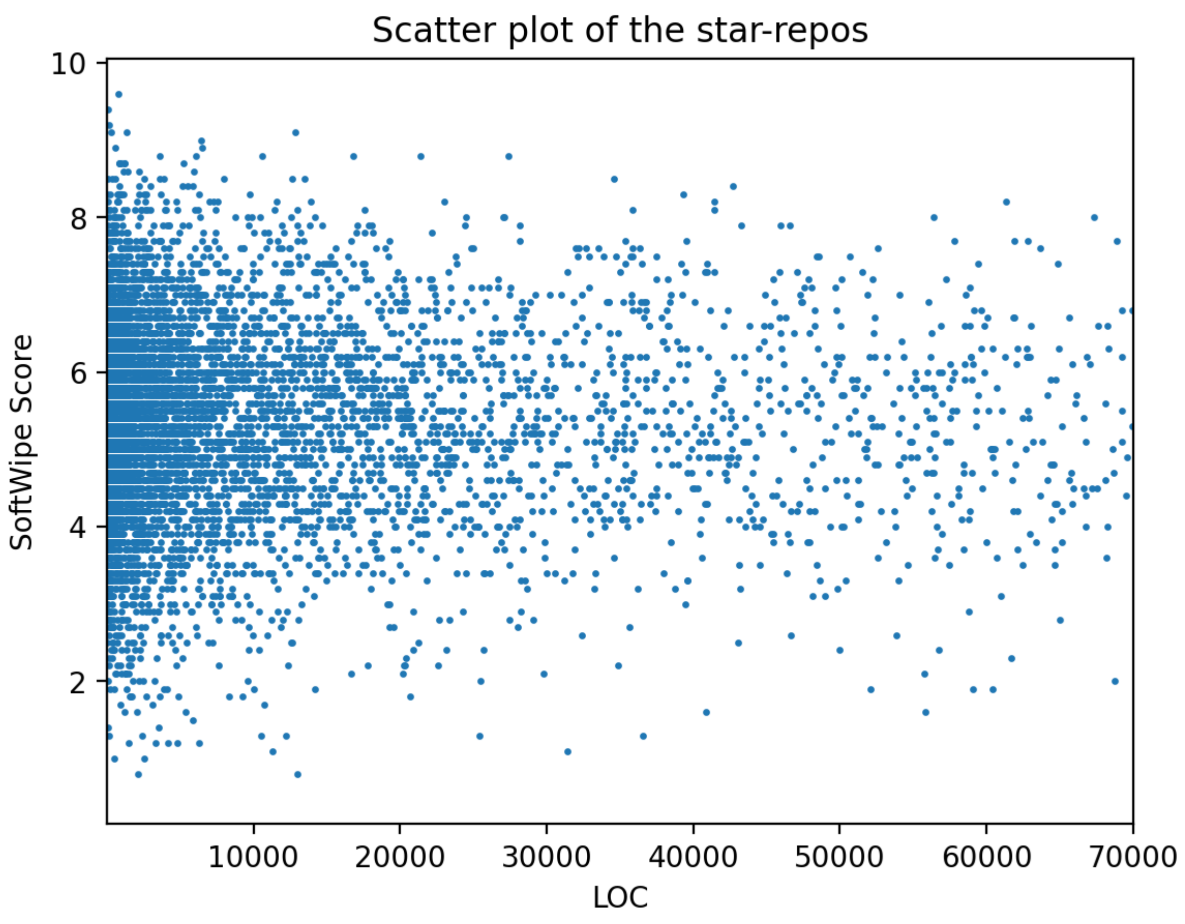


Figure 3.10: Softwipe scores over LoC for star-repos cut of at 70.000 LoC

3.2 Conclusion and Future Work

By using the Git-API we successfully collected more than 3,800 repositories containing swearwords. Those swear-repos were then evaluated with the SoftWipe tool to calculate a score, which represents the code quality. Those swear-repos were then compared to over 7600 repositories, which we selected to be our general population and also analysed with the SoftWipe tool. This comparison was done by running multiple hypothesis tests, such as the Kolmogorov-Smirnov test. These tests, combined with our visual analysis of the data yielded the result that repositories containing swearwords exhibit a statistically significant higher average code-quality (5.87) compared to our general population (5.41).
There are many things that you can consider to further add to this study. Of course "[...] there's no data like more data"-Kai-Fu Lee, thus obtaining more swearword samples is of course of interest to us, as well as the possibility to include C++ code as well. During the data crawling, one could of course also deploy natural language processing to more accurately identify swearwords.
One of the key unanswered questions are the clusters in Figure 3.9. For this a more thorough investigation into the similarities between these repositories could provide a better inside into their existence.
Also, a more detailed study on how the numbers of lines of code or star restrictions affect the data could be interesting. This can be further supplemented by investigating the existence of a correlation of the number of swearwords per lines of code to the code quality or by investigating the existence of a correlation between the number of stars to the code quality. Since the Git-API returns a lot of information on the repositories there are a lot more interesting investigations one could conduct, which would then drastically increase the dimensions of the data and therefore increase the complexity of the analysis.

Bibliography

[1] Yehuda Baruch et al. “Swearing at work: the mixed outcomes of profanity”. In: *Journal of Managerial Psychology* 32.2 (Jan. 2017), pp. 149–162. ISSN: 0268-3946. DOI: 10.1108/JMP-04-2016-0102. URL: https://doi.org/10.1108/JMP-04-2016-0102.

[2] *Beautiful Soup Documentation — Beautiful Soup 4.9.0 documentation.* URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (visited on 12/24/2022).

[3] “Central Limit Theorem”. In: *The Concise Encyclopedia of Statistics.* New York, NY: Springer New York, 2008, pp. 66–68. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_50. URL: https://doi.org/10.1007/978-0-387-32833-1_50.

[4] Allan S. Cohen and Seock-Ho Kim. “Behrens–Fisher Problem”. In: *International Encyclopedia of Statistical Science.* Springer, Berlin, Heidelberg, 2011, pp. 138–141. DOI: 10.1007/978-3-642-04898-2{\textunderscore}142. URL: https://link.springer.com/referenceworkentry/10.1007/978-3-642-04898-2_142.

[5] B. Efron. “Bootstrap Methods: Another Look at the Jackknife”. In: *The Annals of Statistics* 7.1 (1979), pp. 1–26. DOI: 10.1214/aos/1176344552. URL: https://doi.org/10.1214/aos/1176344552.

[6] GitHub. *GitHub - XAMPPRocky/tokei: Count your code, quickly.* URL: https://github.com/XAMPPRocky/tokei (visited on 01/29/2023).

[7] GitHub Docs. *Resources in the REST API - GitHub Docs.* URL: https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28#rate-limiting (visited on 12/27/2022).

[8] GitHub Docs. *Saving repositories with stars - GitHub Docs.* 2023. URL: https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars (visited on 01/09/2023).

[9] GitHub Docs. *Search, - GitHub-Dokumentation.* URL: https://docs.github.com/en/rest/search?apiVersion=2022-11-28#search-code (visited on 01/12/2023).

[10] Raymond Hubbard. “P–Values”. In: *International Encyclopedia of Statistical Science.* Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1144–1145. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_465. URL: https://doi.org/10.1007/978-3-642-04898-2_465.

[11] Carlos M. Jarque. “Jarque-Bera Test”. In: *International Encyclopedia of Statistical Science.* Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 701–702. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_319. URL: https://doi.org/10.1007/978-3-642-04898-2_319.

[12] Carlos M. Jarque and Anil K. Bera. “Efficient tests for normality, homoscedasticity and serial independence of regression residuals”. In: *Economics Letters* 6.3 (1980), pp. 255–259. ISSN: 0165-1765. DOI: https://doi.org/10.1016/0165-1765(80)90024-5. URL: https://www.sciencedirect.com/science/article/pii/0165176580900245.

[13] Damir Kalpić, Nikica Hlupić, and Miodrag Lovrić. “Student’s t-Tests”. In: *International Encyclopedia of Statistical Science*. Springer, Berlin, Heidelberg, 2011, pp. 1559–1563. DOI: 10.1007/978-3-642-04898-2{\textunderscore}641. URL: https://link.springer.com/referenceworkentry/10.1007/978-3-642-04898-2_641.

[14] Maurits Kaptein and Edwin van den Heuvel. *Statistics for Data Scientists: An Introduction to Probability, Statistics, and Data Analysis*. 1st ed. 2022. Undergraduate Topics in Computer Science. Cham: Springer International Publishing and Springer, 2022. ISBN: 9783030105310. DOI: 10.1007/978-3-030-10531-0. (Visited on 01/06/2023).

[15] “Kolmogorov–Smirnov Test”. In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 283–287. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_214. URL: https://doi.org/10.1007/978-0-387-32833-1_214.

[16] Venus Liew. “An overview on various ways of bootstrap methods”. In: *MPRA Paper* (Jan. 2008).

[17] Robert C. Martin and James O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009. ISBN: 978-0-13-235088-4. URL: https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00.

[18] Raymond S. Nickerson. “Null-Hypothesis Significance Testing: Misconceptons”. In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1003–1006. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_422. URL: https://doi.org/10.1007/978-3-642-04898-2_422.

[19] “Null Hypothesis”. In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 388–389. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_290. URL: https://doi.org/10.1007/978-0-387-32833-1_290.

[20] Alberto Pettorossi. *Automata Theory and Formal Languages*. 1st ed. Undergraduate Topics in Computer Science. Springer Cham. ISBN: 978-3-031-11965-1. DOI: 10.1007/978-3-031-11965-1. URL: https://link.springer.com/book/10.1007/978-3-031-11965-1 (visited on 01/09/2023).

[21] *Python Dictionaries*. URL: https://www.w3schools.com/python/python_dictionaries.asp (visited on 12/27/2022).

[22] Russ Cox. *Regular Expression Matching in the Wild*. 2015. URL: https://swtch.com/~rsc/regexp/regexp3.html (visited on 12/24/2022).

[23] *scipy.stats.shapiro — SciPy v1.10.0 Manual*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html (visited on 12/26/2022).

[24] SpringerLink. *Modern Mathematical Statistics with Applications*. 2st ed. Springer Cham, 2023. ISBN: 978-1-4939-4221-3. DOI: `https://doi.org/10.1007/978-1-4614-0391-3`. URL: `https://link.springer.com/book/10.1007/978-1-4614-0391-3#%20toc` (visited on 01/19/2023).

[25] SpringerLink. *Statistics for Data Scientists*. 1st ed. Springer Cham. ISBN: 978-3-030-10531-0. DOI: `https://doi.org/10.1007/978-3-030-10531-0`. URL: `https://link.springer.com/book/10.1007/978-3-030-10531-0` (visited on 01/23/2023).

[26] Dorothea Wagner. "Theoretische Grundlagen der Informatik Vorlesung am 15.10.2019". In: (). URL: `https://i11www.iti.kit.edu/_media/teaching/winter2019/tgi/tgi-2019-20-vorlesung-01.pdf` (visited on 01/30/2023).

[27] WhatIs.com. *What is population? | Definition from TechTarget*. URL: `https://www.techtarget.com/whatis/definition/population` (visited on 01/19/2023).

[28] Wikipedia, ed. *Ariane flight V88*. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Ariane_flight_V88&oldid=1131897741` (visited on 01/28/2023).

[29] Wikipedia, ed. *GitHub*. 2022. URL: `https://en.wikipedia.org/w/index.php?title=GitHub&oldid=1130382758` (visited on 12/31/2022).

[30] Adrian Zapletal et al. "The SoftWipe tool and benchmark for assessing coding standards adherence of scientific software". In: *Scientific Reports* 11.1 (2021), p. 10015. ISSN: 2045-2322. DOI: `10.1038/s41598-021-89495-8`. URL: `https://www.nature.com/articles/s41598-021-89495-8`.