Mining Graph Patterns in Software Development from Code Repositories

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science,
Data Science

By:

Alexander MacDonald
Ethan Vaz Falcao
Jakob Simmons
Nur Fateemah

Project Advisors:

Fabricio Murai
Frank Zou

Date: March 1st 2024

Abstract

GitHub is a cloud-based version control system built on the Git platform and is used by over 100 million users globally. When viewing a repository on GitHub two important features are notably missing on a repository page. The first is a complexity metric, indicating how difficult it would be for a new user to pick up and start working on the project. The second is whether or not a repository will be active in the future, sometimes this information can be found in a project's ReadMe or by analyzing recent commits but there is no single indicator. These two factors are very important when determining whether or not to use the code in a GitHub Repository or when importing a library from GitHub into a project. This project aims to use machine learning models to estimate complexity metrics and determine a repository's status based on data retrieved from the GitHub API without having to download the repository to your local computer. This was done by generating graphs of repositories with files being nodes, edges connecting files committed together and the weights of those edges being the number of times two files were committed together. Encodings of the features of these nodes were created with a Graph Auto Encoder to be used as features. It was found that with the models used graph encodings provided no benefit to predictive models.

Acknowledgements

First, we would like to thank Professor Carlos Henrique Gomes Ferreira of the Universidade Federal de Ouro Preto in Brazil. Hadn't he proposed the idea of modeling GitHub repository information through graphs of source files as input to machine learning models, this project would never have existed.

Additionally, we would like to thank our advisors Professor Fabricio Murai and Professor Frank Zou. The knowledge that Professor Murai provided us on what Machine Learning models to use, how to incorporate our graph data into our models, and how to best generate our training data was invaluable. Additionally, the concept of using graphs based on GitHub repositories as additional methods for training these models was a great opportunity for learning. Without the insight that Professor Frank Zou provided us for our EDA section, we would not have had the same understanding of our data that we have now.

Finally we would like to thank Lou Nelson Jr. of the GitHub support staff for going above and beyond in helping us to set up the GitHub enterprise token. Without his help, the collection of our data using the GitHub API would have been a much more complicated and time-consuming process.

Contents

List of Tables

List of Figures

1 Introduction

Michigan Technical University defines software engineering as "the branch of computer science that deals with the design, development, testing, and maintenance of software applications." [1] Software engineers write the software that people interact with daily to solve a problem. The term "software engineering" was first used around 50 years ago and has since become one of the most rapidly growing engineering disciplines. When Margaret Hamilton, one of the programmers who wrote the code for the command and lunar modules on the Apollo missions [2], first used the term software engineering [3], there were no widely disseminated computer programming principles or patterns. You could learn the basic commands for how to interact with the computer but not how to create "good" programs which, in essence, are easy to read, edit, and maintain, are well thought out, and do not reinvent the wheel. To make it easier to write good code, coding methodologies such as Waterfall and Agile were developed so that large teams could build and fix software in a structured manner [4]. However, as programs and software engineering teams grew, people needed a way to share the code and fixes they had written quickly and more efficiently.

1.1 Code Organization

One source of code complexity comes from the dependencies between different sections of code. According to Sonatype "a software dependency is a relationship between software components where one component relies on another to work properly" [5]. As the size of coding projects grew and they required more complex functionality to do their jobs, the number of classes and thus the number of dependencies increased. In addition to the number of dependencies increasing the size of the file systems also grew. The 3.5-inch floppy disk on which many programs were written could only store 256 KB and at their latest stages could store up to 2 MB [6]. Modern SSDs can have data storage in the order of terabytes and applications can take hundreds of megabytes while video games can take multiple gigabytes. These large-scale programs often comprise a much larger number of programming files consequently requiring a way of storing and sharing programs on and between computers. As programs got larger programmers needed ways to determine how complex their code was and devised several coding metrics. Although there is no one "all-encompassing" code metric, by using a comprehensive collection of them, code quality can be estimated relatively well. The solution that programmers devised to solve this problem was Version Control.

1.2 Version Control

According to Atlassian, one of the leaders in version control systems, version control "is the practice of tracking and managing changes to software code." [7] Version control has gone through several different iterations starting with the IEBUPDTE system used with IBM's OS/360 [8]. Although it has been a matter of debate whether or not this was truly the first iteration of version control since it was a punch card system [8], it did provide the means to create and update code libraries similar to today's version control systems. Before the first collaborative version control system came out, there were a few systems called SCCS (Source Code Control System) and RCS (Revision Control System) which allowed a sole developer to track changes they made on files but did not allow more than one developer to work on a file at once. It wasn't until CVS (Concurrent Versions Systems) was developed by Dick Grune in 1986 that allowed more than one developer to work on a file at the same time [8].

Currently, Git is the most well-known Version Control System [9]. Git is a free open-source version control system that was created by Linus Torvalds for the development of the Linux kernel [10]. One of the major features that makes Git different from any other version control system is how Git stores data. Other version control systems such as CVS, Subversion, and Perforce store information as a set of files and the changes made to each one over time. Git instead stores a snapshot of the file system on commit. For any files that were not changed, Git stores a link back to the previous snapshot. For files that were changed, Git stores a new version of that file for the snapshot while still retaining the old ones [11]. On its own Git is a very helpful tool, it allows a developer to store backups and view a log of changes made to their project which can be stored on a server for others to access [12]. However, this raises a new issue: creating a server to store Git projects can be expensive in terms of price, time, and knowledge required. Once again, developers had to develop a solution for this; Cloud-Based version control systems.

1.3 Cloud-Based Version Control Systems and GitHub

In the age of the modern internet, developers have created many cloud-based version control systems so that people can work together on projects remotely using Git without needing a centralized server. Some examples include GitLab [1], Bitbucket [2], and of particular relevance, GitHub [3]. GitHub is used by people from all over the globe to create large and small-scale projects, many of which are built upon and used by other developers to improve their projects or to avoid writing code that has already been written by someone

[1] https://about.gitlab.com/
[2] https://bitbucket.org/product
[3] https://GitHub.com/

else [13]. Currently, there are around 100 million users on GitHub, most of which come from the US, India, and China [14]. In 2022, there were over 413 million open source contributions on GitHub, and over 90 percent of the Fortune 100 companies used GitHub to host their projects [15].

Synopsys defines Open source software as "software that is distributed with its source code, making it available for use, modification, and distribution." [16] One of the first open-source programs was the A2 system, an object-oriented operating system developed at UNIVAC, which was shipped with its source code included [17]. Since the A2 system the rise of open source software, especially within the last few years has been astronomical. In the modern day, people can interact with and edit the code for the programs they are using more often and companies built on open source software like MongoDB are selling for billions of dollars [18]. With the rise in open source code has also come the desire and opportunity to study the code written from a research lens and determine what is truly considered "good code." Zimmerman et al. [[19]] used machine learning models to predict if a file will contain bugs on new releases of Eclipse and our hope is to extend these ideas to repository maintainability and quality.

GitHub has many built-in features that allow developers to see who is working on their repository. Moreover, the GitHub REST API allows users to view information such as commit history and issue data. However, GitHub does not provide metrics about code quality or maintainability built into their website or REST API. This has left a gap in the market for companies to provide SaaS products to better review code and analyze maintainability.

Many competing companies have been able to capitalize on this niche but the two that have carved out their spot in the market are Qodana by Jetbrains [4] and SonarQube by Sonar Source [5]. Qodana and SonarQube allow developers to inspect static code across many languages, 60+ for Qodana and 29+ for SonarQube. They are a sort of all-purpose repository tool as it provides information on bugs, code smells, coverage, complexity, and security. The common issues from these platforms include limited support from their free versions coupled with expensive commercial licenses for access to more in-depth analysis and support. Additionally, tools like these often require code repositories to be downloaded to a computer and require access through GitHub Enterprise which is an additional expenditure.

Goal

The goal of this project is to develop an application that can be used to analyze a GitHub repository without downloading the entire repository by using the GitHub API.

[4]https://www.jetbrains.com/qodana/
[5]https://www.sonarsource.com/

Objectives

Our team will achieve the goal through the following four objectives.

1. Collect data using pre-existing tools, frameworks, and APIs
2. Establish general repository features that could be be associated with maintainability and activity status.
3. Establish if/what network graphs are good predictors of maintainability and activity status.
4. Use data networks to determine the maintainability of a code repository and whether or not that repository is "dead" or "alive".

Outline

In Section 2, we present the literature review, where we discuss some of the code metrics and repository analysis tools that have been developed and used in the past. Next, in Section 3, we explain the methods we propose to use in this project, discussing other alternatives and existing trade-offs. In Section 4 we cover our findings from our exploratory data analysis. Section 5 we cover the architecture and functionality of our data collection tool. In Section 6 we go over the results of the different models we trained for our desired tasks. Section 7 covers the limitations that we faced over the course of the project. In Section 8, we found that the models we used could not explain the complexity between our graph encoding features and targets and we discuss the final product that we created which can gather the data from the a repository, and determine Maintainability Index and the active status of a repository. Finally, Section 9 discusses future work in this field to be done.

2 Literature Review

Many collaborative coding projects on GitHub generate vast amounts of data, including repository metrics, which are crucial for assessing code maintainability and efficiency. However, accessing and analyzing this data effectively is a challenge. This literature review aims to explore existing tools and solutions that can assist in training a model to analyze the maintainability and efficiency of code repositories. This report will look at code quality measurements, the different tools to extract and analyze data, and ways to visualize and network the data.

The research articles that it covers are primarily focused on tools and methodologies designed for (i) extracting commit metrics from GitHub repositories and generating comprehensive reports and (ii) computing code quality metrics that have been used in the past. These studies delved into the ever-expanding landscape of software development and sought to address the growing need for efficient and insightful metrics analysis.

2.1 Code Quality Measurement

It is important to write code that will be easy to read by others, easy to identify and fix bugs in, and easy to extend in the future. To do this, software engineers have to know how to write good quality code. Although it is important to know how to write quality code, software engineers also need a way to measure the quality of the code and thus several metrics have been created over the years to measure code quality including the Halstead Measurements, Cyclomatic Complexity, and Maintainability Index, which are described in detail below.

2.1.1 Halstead /Measurements

The Halstead Measurements were developed by Maurice Howard Halstead [20]. They can be used to calculate how complex code is based on the total number of unique operators (n_1), the total number of unique operands (n_2), the total number of occurrences of operators (N_1), and the total number of occurrences of operands (N_2). The equation for Halstead Volume is $V = (N_1 + N_2)\log_2(n_1 + n_2)$ and the equation for Halstead difficulty is $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$. A single metric that accounts for both of the factors above is the Halstead Effort H_E, which is defined as the product between Halstead Volume and Halstead Difficulty: $E + V \times D$.

Halstead's metrics were one of the first attempts at describing code complexity but left much to be desired. According to a group from Bell Labs, the estimates that are derived from Halstead's metrics to determine the number of errors to be expected were not true on a small scale but perhaps were accurate on a very large scale [21].

2.1.2 Cyclomatic Complexity

The Cyclomatic Complexity metric was developed by Thomas McCabe for procedural languages, although it is still a common code metric in the day of object-oriented languages [22]. Cyclomatic complexity is calculated using $M = E - N + 2P$ where N is the count of every statement in the code, E is the connections between the statements and P is the number of outcomes that the program can produce. Cyclomatic

complexity's usefulness in academia is questionable due to it being a slightly more sophisticated variant of lines of code count. However, in the industry, it is widely used to get a quick and easy measure of how complex a program is [22].

2.1.3 Maintainability Index

Oman and Hagemeister [23] created two different functions for the Maintainability Index, the 3-metric and the 4-metric calculations, which differ on whether the number of comments is taken into account. The 4-metric Maintainability Index MI_4 used Halstead Effort (E), Cyclomatic Complexity (C), Lines of Code (LoC), and Number of Comments (CM) and is computed using the equation

$$\mathrm{MI}_4 = 171 - 3.42\log(\mathrm{avg}(E)) - .23\mathrm{avg}(V) - 16.2\log(\mathrm{avg}(\mathrm{LoC})) + .99ave(\mathrm{CM}),$$

Where the averages are taken across each module in a program. Some success has been found using the Maintainability Index to quantify and improve software maintainability but it is still only a piece of the puzzle [24].

2.2 Existing Tools

Tools that collect necessary data from GitHub repositories play a crucial role alongside code quality metrics. Many research articles emphasize GitHub's significance in fostering open-source communities and stress the importance of extracting meaningful metrics to understand project health, developer contributions, and code quality. Studies such as "Tooling for Time- and Space-Efficient Git Repository Mining" by Heseding, Scheibel, and Döllner, and "Three Trillion Lines: Infrastructure for Mining GitHub in the Classroom" by Mattis, Rein, and Hirschfeld exemplify this recognition.

One such tool, GIT2NET, uncovers detailed collaboration networks within Git repositories and provides insights into code changes and ownership by employing text mining techniques and constructing time-stamped networks. Additionally, GIT2NET has the potential to provide in-depth insights into code maintainability and efficiency[25]. The tool creates a comprehensive database capturing details of code co-edits at various levels by using advanced text mining techniques and metrics such as Levenshtein edit distance and text-based entropy.

Additionally, GIT2NET generates timestamped collaboration networks, supports multiple export formats, and efficiently handles massive repositories with its parallel processing capability. In a case study

involving two software projects, the researchers demonstrated GIT2NET'S ability to analyze file modifications at a fine-grained level, producing significantly different network structures compared to coarse-grained methods.

A second tool is PYREPOSITORYMINER [26], written in Python, combines various optimization approaches to streamline repository traversal and data extraction, supporting both Python-based metrics and external metrics for software analysis. The paper presents extensive performance evaluations, showcasing speed improvements and flexibility for integrating custom metrics and external analysis tools, particularly suitable for Python-based code repositories. Lastly, PYDRILLER, a versatile Python framework, simplifies the complex process of mining Git repositories by efficiently extracting commits, modifications, and source code, offering extensive APIs for data retrieval and manipulation, making it a promising solution for research project.

Finally, PYDRILLER is a versatile Python framework that simplifies the intricate process of mining Git repositories[27]. Spefically, this framework's agility in extracting information, encompassing commits, modifications, and source code, holds promise for research projects. PYDRILLER'Sfeatures are presented as a solution to challenges such as slow and complicated extractions of different repositories. It also offers extensive APIs that facilitate data retrieval and manipulation.

2.2.1 Other Methods

GHTorrent. Other research articles suggested another route of directly using the GitHub API along with GHTorrent. Gousios and Spinellis [28] delve into the challenge of analyzing GitHub repositories using an API while circumventing the inherent limitations imposed by GitHub's API. GitHub, as the premier platform for global software development collaboration, accumulates a vast reservoir of data crucial for understanding software engineering practices, collaboration dynamics, and project evolution. However, GitHub's API constraints, including rate limits and historical data access restrictions, pose a formidable obstacle to comprehensive and large-scale analysis. To address this issue, the article introduces a solution in the form of the "GHTorrent project".

Another project [28] serves as an offline mirror of GitHub's extensive data, offering a comprehensive dataset that comprises raw data and their metadata and dependencies. The data is processed and stored in both relational and MongoDB databases. With GHTorrent's open-source framework and over 15 TB of data, it has garnered attention within the research community, enabling more than 200 researchers to harness its online access points for empirical research. In summary, the GHTorrent project emerges as a potential

resource, encouraging researchers to unlock insights regarding GitHub's collaborative software development landscape while navigating API limitations.

Analysis of repository forks. An alternative approach involves the utilization of umbrella repositories which is seen in Biazzini and Baudry [29]. It involves constructing a single Git repository that serves as a comprehensive container for all forks of the same project, which encapusaltes the entire collaborative ecosystem and comparative analysis. The process begins by creating an empty Git repository and systematically adding each fork of the project as a remote, each with a unique identifier. Subsequently, the content of each branch from the forks is fetched into this "umbrella repository.". The resulting "umbrella repository" serves as a powerhouse of collaboration insights. It reveals the common commit history shared among all forks and also optimizes memory utilization by consolidating data from various repositories.

Through the use of directed acyclic graphs, this method captures, matches, compares, and mines complex development histories. The tool GITWORKS is employed to organize and extract metrics from Git repositories, enabling the characterization of the "official" repository history concerning contributions from the forks. It tracks how commit histories diverge and converge over time due to code evolution, measuring divergence by monitoring non-trivial commits post-fork. A pivotal aspect of this approach is the identification of "Interesting Commits" (iCommits) unique to specific forks. These iCommits unveil relevant content not present in the mainline code, shedding light on shared code among subgroups of developers. In essence, this methodology offers insights into unique code contributions and collaboration patterns, facilitating a comprehensive understanding of code evolution and developer dynamics within a multi-forked GitHub project. This method is relevant because it shows the intricate cases of forking GitHub repositories and why it would be complicated to create metrics for these type of repositories.

File-by-file approach. Another method found in the literature review took a file-by-file approach. Muthukumaran et al. [30] used some simple predictors such as the number of commits, lines added, and lines deleted, as well as some more complicated code churn predictors such as Mean Period of Change, Maximum Burst, and Maximum Change Set to determine whether a file would have bugs on release. Using Gaussian NB, CART Decision Trees, Logistics Regressions, and NB Tree algorithms, Muthukumaran et al. were able to achieve an accuracy of over 80 percent in determining if a file would contain bugs post-release with every method and achieved an accuracy of 90.4 percent with the logistic regression.

2.3 Comprehensive Tool Comparison

The literature review indicated that three tools were considered and evaluated using various GitHub repositories of different file sizes and commit counts. Among these tools, PYREPOSITORYMINER presented limitations, as it required specific code technologies like Ansible and Tosca to be integrated into the repositories to function effectively. Ansible is an open-source automation tool utilized by system administrators and DevOps engineers to automate software provisioning, configuration management, and application deployment tasks in IT environments. TOCSA (Telecommunications Operational Security Assessment) is a framework developed by the European Union Agency for Cybersecurity (ENISA) specifically designed for assessing the security of telecommunication networks, commonly used by telecommunications companies and regulatory bodies to evaluate the security posture of telecommunication infrastructures. From the above defiinitons, this limitation would restrict our analysis to a small subset of repositories. Despite the tool's relatively faster thread performance, this constraint did not align with the project's core objective of analyzing all publicly available GitHub repositories comprehensively. Consequently, PYREPOSITORYMINER was deemed unsuitable for providing the necessary data to advance the project into its machine-learning phases.

The second tool, GIT2NET [31], emerged as a promising option based on research findings. Git2Net exhibited several strengths, notably its ability to calculate metrics such as Cyclomatic Complexity and various other important code quality or complexity metrics while collecting information from Git repositories. Additionally, Git2Net provided valuable visual representations illustrating the connections between files and collaborators within the repositories. These visualizations offered insights into how files were interrelated and how contributors were connected, enhancing the tool's utility for comprehensive repository analysis. However, in large repositories with many files, commits, or contributors. The graphs often become too complex to gain information from and it takes very long to run when there are more than 1,000 commits.

The third tool under consideration, PYDRILLER yielded a generally positive experience as suggested by the research [27]. PYDRILLER's performance is influenced by the number of commits within a repository, resulting in varying durations for data extraction and analysis depending on the repository's size and commit history. In repositories with extensive commit histories, this could potentially lead to workflow slowdowns. Furthermore, PYDRILLER provides a high-level overview of repository changes but cannot delve into intricate details such as halstead effort and Maintainability Index. While this simplicity is advantageous, it may be perceived as a limitation when detailed insights into code alterations or commit-specific information are required. For users seeking in-depth insights into individual commits or code modifications, a specialized tool or a more profound exploration of Git's command line interface may necessary.

In conclusion, the next steps were to evaluate both PYDRILL and GIT2NET with their performance on different repositories, then either the whole tool will be used or only an aspect of the tool, this is done in Section 3.1.2.

3 Methodology

This section explains the methods that we are using to create a suite of tools and machine-learning models to complete the project objectives. The overarching procedure of this MQP involved gathering general data from repositories, acquiring code quality data, collecting graphical data, training models, and conducting exploratory data analysis (EDA). Subsequent sections will delve into the methodology with greater detail.

3.1 Data Collection and Interfacing with the GitHub API

An initial exploration of existing datasets was conducted to find training and test data for this project but this exploration led to many dead-ends. However, there were two major discoveries. One was a Kaggle dataset that included data from the top 1,000 starred repositories on GitHub in 2017. As of the writing of this document, this dataset has been removed from Kaggle for an unknown reason. We were not able to maintain an active mirror of this dataset as it was not used beyond the first week or so of data collection. The other was a Google-hosted SQL interface provided by GitHub is an interface built around a snapshot of GitHub's public data from 2018 using Google Cloud to enable easier interactions using SQL [32].

Upon further investigation, we found that the Google Cloud SQL was limited in scope and would have been difficult to integrate into our further data processing steps. The Kaggle dataset was determined to have more promise as a data source. We discovered that the data was extremely outdated due to it being from 2017. Furthermore, the scope of the Kaggle dataset was limited in number of metrics provided for each repository such as not included repository files or issues. Although the data stored in the Kaggle dataset was limited, the repositories listed within it were still useful for later data collection, as we discuss below.

We determined that an in-house data collection tool that interfaced directly with the GitHub API would need to be developed to allow for custom functionality and more control over the queries to collect the metrics for training models. There was an initial blocking issue using the GitHub API due to rate limits implemented by GitHub. The free token provided with every GitHub account has a rate limit of only 5,000 API calls per hour thus significantly reducing the speed of data collection.

To circumvent this strict limitation, we considered two alternative solutions. The first solution that we discussed would have taken all of our team's free API tokens to use in a round-robin fashion where a new token would be used once a token had reached the limit. This is a gray-area of the terms of service at GitHub if not entirely against them. As the team would have used their personal GitHub accounts we decided to embark on a different approach by using an Enterprise token from GitHub. The enterprise token provides an increased limit of 15,000 API calls per hour. Obtaining an Enterprise token from GitHub required installing a GitHub application to a GitHub enterprise to obtain a .pem public key file which was then posted to the GitHub API through a JSON web token alongside the account information for the enterprise [33]. This post request returned an Enterprise token with a maximum lifetime of one hour. At the expiration of the token, another token can be requested using the same process. There is no request for free tokens when exceeding their rate limit as they are static tokens. This token with a higher rate limit was satisfactory to our team during data collection.

A query was created using the GitHub search API that provided the top 1,000 repositories by their number of stars from GitHub users as of October 2023. This list of repositories from the search query was then merged with the repositories provided by the Kaggle dataset to have a mix of modern popular repositories alongside the popular repositories from 2017. After removing duplicates repositories and repositories with over 7,500 commits from this list, a final set of data was collected containing $\tilde{1}$,700 repositories. We removed repositories with more than 7,500 commits that were found through exploration because we included repositories with a larger amount of commits through the Kaggle set and we needed to lower the total time for data collection to finish this project on time.

During data collection It could not be determined which metrics were going to be the most useful for model training, so the "cast a wide net" approach was used for data collection on the repositories we wanted to explore. The data definitions (Appendix A) included general repository data, commit data, issue data, and file data for each repository. To collect and format this data in a manageable way, functionality was created to ensure that data would be collected consistently, stored in easily processed CSV files, and have complete up time despite the GitHub API rate limit. This data was stored within a sub-directory of the program and was easily archivable for further processing.

A test repository was also generated for the testing and development of this tool to better understand the GitHub API and its limitations [34].

3.2 Evaluating Tools for Collecting Code Quality Metrics

Through the literature review, we directed our attention towards PYDRILLER and GIT2NET due to their similar features and aspects to PYREPOSITORYMINER without the limitation of repositories requiring Tosca and Ansible. We created a test spanning an assortment of repositories of varying sizes and commit counts to determine efficiency. The range of commit counts in these repositories was between 1,000 and 4,000 commits, with corresponding variations in file sizes. Interestingly, analyses revealed that PYDRILLER'S processing time was closely tied to the number of commits rather than the overall size of the repository. For instance, SINDRESORHUS/AWESOME, which comprised 1,125 commits and a size of 1 MB, took approximately 26 minutes to process fully. In contrast, IMPRESS/IMPRESS.JS, with a much larger file size of 4 MB but a smaller commit count of 400, finished processing much faster in just 14 minutes. (Note: hardware that conducted this has 8GB RAM and 2.60 GHZ processor).

Nonetheless, one particular finding from experimentation with PYDRILLER is crucial to highlight: To collect data from the Facebook REACT GitHub repository, which can be considered an outlier for its size and 15,000 commit count, surpassed the 12-hour mark during analysis with PYDRILLER. This observation underscores the impact of commit volume on PYDRILLER performance, especially when confronted with repositories characterized by extensive and interwoven commit histories .

We explored GIT2NET further. This tool actively facilitated data collection and introduced an additional layer of visual analysis that other tools lacked, according to our research. GIT2NET actively implements the calculation of metrics such as Cyclomatic Complexity. Its capability to generate visual representations of the relationships between files and collaborators within repositories was a notable asset, providing an overview of the intricate relationships within repositories. We conducted the same test on GIT2NET, where it took 43 minutes with SINDRESORHUS/AWESOME and 2 hours with IMPRESS/IMPRESS.JS.

The percentage increase in processing time of GIT2NET compared to PYDRILLER for the sindresorhus/awesome repository was approximately 65%, and for the impress/impress.js repository, it was around 260%. We then decided to profile each of the tools, notably concluding that the longest part of the process for both tools was extracting and mining the data. These findings suggest that Pydriller may be the tool of choice when working with larger repositories and emphasize the importance of optimizing data mining processes in both tools to enhance their overall efficiency. However, another problem arose due to the fact that all the tools covered in the literature review only focused on collecting code quality metrics from each commit rather than each file, which is what we needed in order to analyze and collect data from files. We established the base tool for collecting code quality metrics as PYDRILLER; the next step was to

establish what specific component makes PYDRILLER calculate the metrics and apply it to a file-by-file basis.

3.3 Collecting Code Quality Metrics with a File by File Basis

We proceeded to enhance the collection of code quality metrics that indicate the maintainability of a repository. These metrics included Cyclomatic Complexity, lines of code, Halstead Volume, and Maintainability Index (see definitions in Section 2). As previously stated, PYDRILLER, GIT2NET, and GitHub API were sufficient for collecting general details for the repository but were not sustainable for collecting the code quality metrics due to their time-consuming nature (around 80 minutes for every 50 repositories). Additionally, the tools would collect the metrics for every single commit instead of every file. For instance, for five commits in one file, we needed the code quality metrics for that one file and not for each individual commit. To address those issues, we needed to develop an efficient method for collecting metrics for the entire repository on a file-by-file basis. Since, according to our last Section, PYDRILLER was the most time efficient, we examined how PYDRILLER gathered metrics for individual commits and expanded this approach to cover the entire repository.

Through exploring PYDRILLER'S documentation, it is stated that it wouldn't have been possible to make PYDRILLER without LIZARD. LIZARD an open-source tool easily installable via pip in Python, repository metrics could be collected via the command-line, simplifying the extraction of file-level metrics. LIZARD provides the following data for repositories: nloc (lines of code without comments), AvgCCN (average Cyclomatic Complexity), token count of functions, and the parameter count of functions. The primary limitation of LIZARD was that it only supports 13 languages which includes C/C++ (works with C++14), Java, C# (C Sharp), JavaScript (With ES6 and JSX), TypeScript, Objective-C, Swift, Python, Ruby, TTCN-3, PHP, Scala, GDScript, Golang, Lua, Rust, Fortran, Kotlin, Solidity, and Erlang. LIZARD did not provide the calculations for the rest of the important metrics like the Halstead Volume and the Maintainability Index yet they could be calculated using the formulas and the data collected.After combining the output those two programs, the final metrics of around 1600 repositories was collected with the average Cyclomatic Complexity, average Halstead Volume, and average Maintainability Index.

3.4 Modeling the retrieved data as graphs

There are a number of Python Graph Libraries but the main four that are recommended by the creators of Python are:

- python-igraph [6] - a set of Python bindings for the library igraph[7] which provides a set of portable, easy to use and efficient network tools which benefits from being used across multiple languages, thus receiving regular updates
- NetworkX [8] - focuses on the creation, manipulation and study of complex networks and is based on the NumPy and SciPy packages
- graph-tool [9] - Based upon the C++ Boost Graph library and benefits from parallelization with focuses on graph manipulation and statistical analysis
- rustworkX [10] - a general purpose graph library which benefits from the performance and safety that comes with being written in Rust

It was determined that NetworkX would be best suited for the purposes of the project because of the team's previous experience using the library and its focus on complex network structures which would be leveraged heavily. [35] During research, three different graphs were defined that can help visualize repositories for developers, two of which can provide some intuition regarding code quality or maintainability.

The first graph that was determined could be useful as a pure visualization of a repository is a graph of the repository's file tree which we simply call **repository structure graphs**, can be formally defined as $G_1 = (V, E_1)$, where V is the set of nodes, each representing either a file or directory, and E is the set of directed edges (u, v) representing the fact that u is a parent directory of v, for $u, v \in V$. To distinguish different node types in the rendering of these graphs, leaf nodes, branch nodes, and the root are colored differently and the size of nodes decreases the deeper in a repository they are. An example of a repository structure graph can be seen in Figure 1.

Due to the high number of commits that occur in many repositories, it was determined that the number of commits would have to be limited. This could be implemented by either setting a number of commits n to use or by setting a date range for when the commits occurred. It was determined that setting a date range was a better solution as it provided a good way to view specific time periods, and the number of commits could be limited by looking where the last n commits started. The first of the two graphs that may provide insight into code quality or maintainability is the file commit graph which links files together based on the number of times they have been committed together. These graphs are called **co-committed files graphs** It can be formally defined by $G_2 = (V, E_2, W_2)$ where V is the set of nodes representing files in the

[6] https://python.igraph.org/en/stable/
[7] https://igraph.org/
[8] https://networkx.org/
[9] https://graph-tool.skewed.de/
[10] https://www.rustworkx.org/

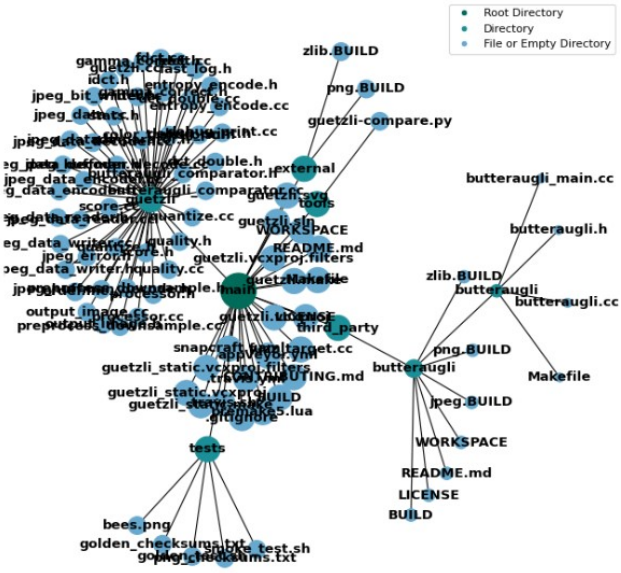

Figure 1: Repository structure graph of Google's repository "Guetzli".

repository, E_2 is a set of undirected weighted edges (u, v) where files u and v were committed together such that $u, v \in V$, and W_2 where w is the weight for edge (u, v) representing the number of times u and v were committed together. These graphs were rendered using a "Inverse Kamada-Kawai squared" layout which inverses the weights of the edges, squares them and then plots the nodes using the Kamada Kawai layout provided by NetworkX [11]. Figure **??** shows an example of Google's repository GUETZLI extracted based on the commits between dates 1/16/2017 and 7/5/2018.

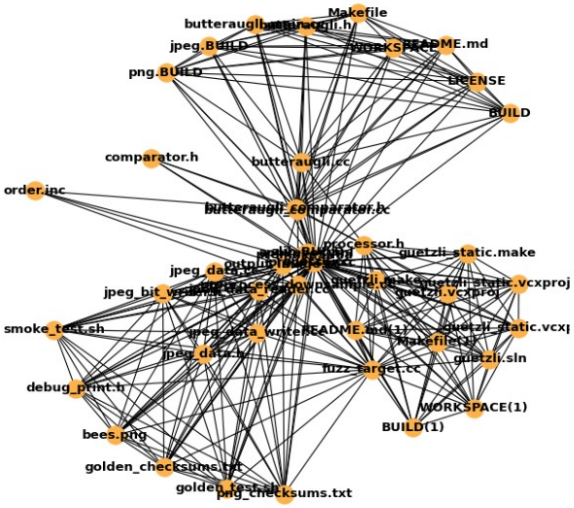

Figure 2: Co-committed file graph graph of Google's repository "Guetzli"

[11] https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.kamada_kawai_layout.html

The final graph is the **user-file commit graph**, a bipartite graph linking author nodes to file nodes when an author u commits to a file v and can be defined by $G_3 = (U, V, E_3, W_3)$ where U is the set of nodes representing authors who committed to the repository, V is the set of nodes representing files in the repository, E_3 is a set of undirected weighted edges (u, v) where u is an author who committed to file v such that $u \in U$ and $v \in V$, and W_2 where w is the weight for edge (u, v) representing the number of times author u committed to file v.

Similarly to G_2, users can limit the start and end dates for the commits being considered. Figure 3 shows the author-file commit graph for Google's repository Guetzli extracted based on the commits between dates 1/16/2017 and 7/5/2018. This graph was rendered using the bipartite layout provided by NetworkX [12] where author nodes are shown on the left and file nodes are on the right.

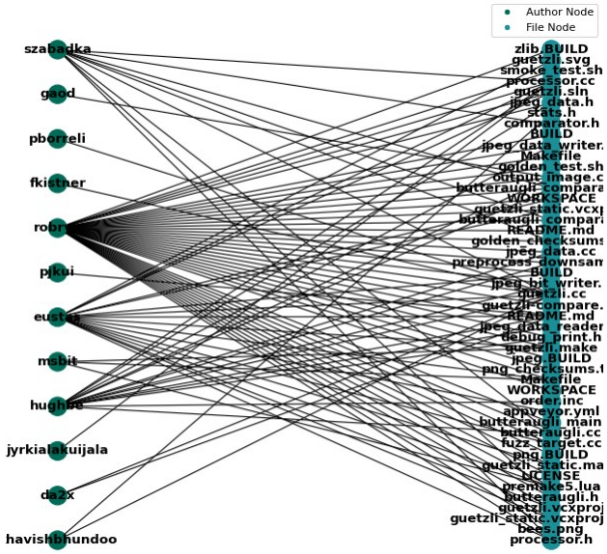

Figure 3: User-file commit graph of Google's repository "Guetzli"

3.5 Extracting graph features via Graph Autoencoders

In order to leverage the graph data collected with machine learning methods, it was determined that Graph Autoencoders (GAEs) could be used to create reliable graph embeddings. Other methodologies such as graph2vec creates embeddings through random walks and thus do not consistently generate the same encodings for graphs. GAEs are a subset of Graph Neural Networks (GNNs) that can be used to map graph data into lower dimensional spaces[36]. The inputs to a GAE are a graph $G = (V, F, E, W)$ where V is a list of nodes, F are a set of features for each node in G, E are a set of (u, v) such that $u, v \in V$ and W

[12] https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.bipartite_layout.html

are the weights for edges in E. The input for the GAE were those generated in the previous section of the form G_2 and the node features were file size, the last date the file was committed to, and a one hot encoding generated by the json file described in Appendix B

The GAE used was a variation on the implementation of Thomas N. Kipf and Max Welling's GAE [37] written by GitHub user shionhonda [13] in Deep Graph Library[38] (DGL). The primary difference between shionhonda's implementation and the created implementation was that DGL's built-in implementation of a Graph Convolutional Network (GCN) was used to encode the data instead of the custom layers shionhonda used for simplicity. The initial features for each node were file size, last date committed, and a one hot encoding of the file's language based on its file extension. A list of the languages and file types included can be found in Appendix B. The GAE has 3 hidden layers which reduce the initial 19 input features into 8 output features through hidden layers which encode 16, 14, and 10 hidden features, no optimization was done. This resulted in a 1.085 binary cross-entropy loss. The average for each output feature was then taken across every node for the graph and the resulting vector was used as input features to our machine learning models, as described in the next section.

3.6 Prediction Tasks

This project has the goal of training models to two different prediction tasks. These two tasks directly relate to objective 4 found in Section 1. In order to do this a regressor must be trained which can estimate complexity metrics of a repository and a classifier must be trained to predict the activity status of a repository.

For both the prediction tasks, the models that were chosen to be trained were Random Forest, K-nearest neighbors, Lasso Regression, and XGBoost for regression and classification tasks. The following list shows the reasons for each model:

- Random Forest was chosen due to its ability to handle high-dimensional data and capture complex relationships between features and target variables.
- K-nearest neighbors was selected for its simplicity and effectiveness in capturing local patterns in the data. It does not make strong assumptions about the underlying data distribution and can be particularly useful when dealing with non-linear relationships between features and target variables.
- Lasso regression was included because of its ability to perform feature selection by penalizing the

[13] https://GitHub.com/shionhonda/gae-dgl/tree/master

absolute size of the regression coefficients. This can help in identifying the most relevant features for predicting the target variables, leading to potentially simpler and more interpretable models.

- XGBoost, a gradient-boosting algorithm, was chosen for its high predictive accuracy and efficiency. It iteratively builds a sequence of weak learners, each focusing on the mistakes of the previous ones, leading to improved overall performance.

Additionally, multiple libraries such as SciPy and scikit-learn are utilized for model implementations, hyper-parameter tuning, and feature normalization functions.

The choice of these models was also influenced by the nature of the features used in the analysis. The graph features, such as stars, date created, date pushed, num commits, open issues, closed issues, total additions, total deletions, total issues, and file count, suggest the presence of complex relationships that these models are well-equipped to capture, the listing of these features can be found in Appendix A. These models were also trained on only the graph encodings generated by our GAE and the concatenation of the initial features and the graph encodings.

Training and evaluation metrics

Training Metrics:

- Mean squared error.
- Log-loss

Evaluation Metrics:

- Mean absolute percentage error
- Accuracy percentage
- For qualitative assessment: confusion matrices

Dataset cleaning, and training, validation and test splits. The dataset will be split into three parts, training, testing, and validation. The dataset which is in the form of a CSV file is pre-processed and cleaned before being used for training the models. Specifically, all rows with null complexity values, rows that threw an error when generating the DGL graphs, or rows that threw an error when generating the encoding for the graphs were removed totaling a loss of 296 or 18% of the initial 1601 rows. Additionally, for the classification task, we removed the rows which had an activity status of "unknown" and trained on binary classification.

4 Exploratory Data Analysis Insights

This section will cover the process while conducting Exploratory Data Analysis (EDA) and derive insights, to create heuristic approaches in order to predict future activity and determine the current status of repositories.

4.1 Characterization of Commits and Developer Activity

In this section of the EDA, we delve into individual GitHub repositories and then conduct an overall view of multiple repositories. Throughout the EDA, some of our main goals were to gain insights into development dynamics and to investigate the relationship between commit messages, code changes, and the overall status of a repository.

To gain intuition about the properties of GitHub repositories, we performed an initial case study on Facebook's REACT repository using the dataset of every commit in the repository, which contained key metrics such as authors, date, commit message, code additions, code deletions, number of files, and the newly introduced 'net_code_change'.

Initially, we looked at the numerical data and the shape and distribution of the code changes in the REACT repository, shown in Figure 4. Each change involves approximately 35 additions, 15 deletions, and thus, a net code change of 20 lines added. The presence of outliers, indicated by the high standard deviations and extreme maximum values, suggests that some changes are substantially larger than the typical ones. Notably, the distribution of all the features skewed left along the x-axis, indicating a vast majority of small code changes.

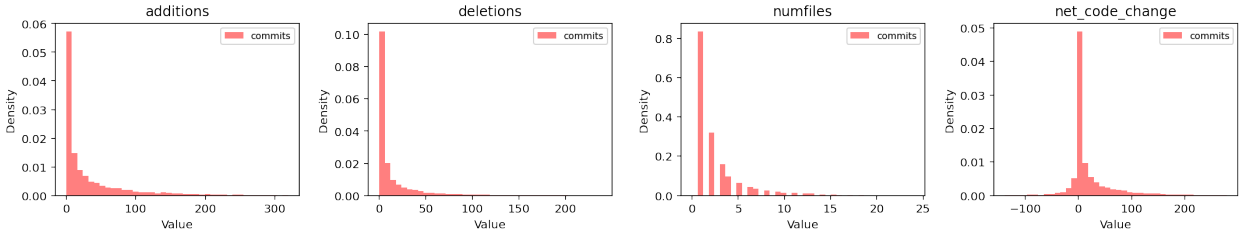

Figure 4: Distributions of Facebook's "react" commit statistics

Next, we go into the length of the commit messages authored by contributors of REACT. As seen in Figure 4, the distribution was right-skewed, indicating that a majority of the messages were brief, consisting of only a few words. Further exploration involved investigating the correlation between the net code change

per commit and the commit message length which revealed a small correlation of 0.231, suggesting that the length of commit messages had little to no impact on the/carries very little information about net code changes.

Following our examination of commit message lengths in the REACT repository, we look at the activity patterns of the repository's three most active authors by commit count. The scatter plot depicted in Figure 5 illustrates the commit frequency grouped by week for the three most active contributors from 2013 to 2023, each represented by a unique color: 'zpao' in blue, 'gaearon' in orange, and 'sophiebits' in green.

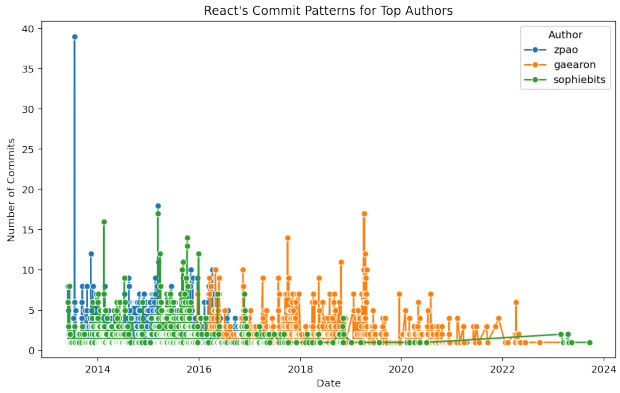

Figure 5: React's Author Commit patterns

Historical data is shown through the plot, with "zpao" and "sophiebits" contributing to the early phase of the project with a substantial number of commits. This initial surge of contributions gradually fades off, which could reflect a shift in project leadership or a change in the contributor's focus within the project's lifecycle. Additionally, "gaearon" presents a portrait of steady and sustained contributions over the years, which has occasional bursts that could relate to critical development periods or version releases. "Sophiebits" reemerged later in the timeline but displayed bursts of activity, which could be indicative of significant project milestones, the undertaking of substantial features, or focuses being split between projects.

Building on our examination of the commit message lengths within the REACT repository, we now shift our focus to the actual code contributions. As shown in Figure 6, this scatter plot provides a visual representation of the net code changes associated with the commits from these top contributors. These data points, when interpreted alongside the earlier discussed commit message lengths/number of commits, deepen our understanding of the collaboration within the REACT repository. The visualization of net code changes adds depth to our prior insights, reinforcing the idea that the length of commit messages is not a reliable indicator of the extent of code contributions.

Figure 6 traces the magnitude of each author's code changes over time, with the net effect of their

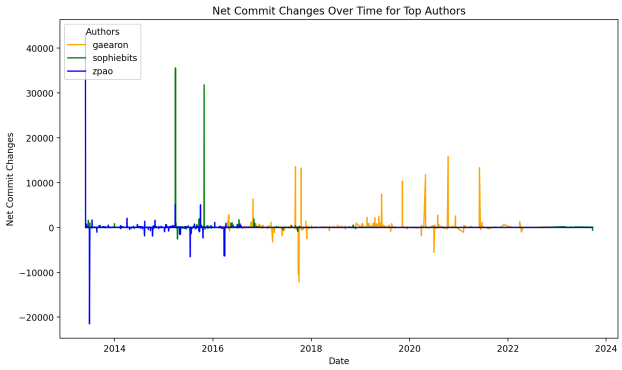

Figure 6: Net Code Changes by React's Top Authors

contributions (additions subtracted by the total deletions) plotted along the y-axis against time on the x-axis. Each author is distinguished by color, with "zpao" represented in blue, "gaearon" in orange, and "sophiebits" in green. Based on the history, we can discern "zpao" and "sophiebits" as key early contributors, with their intense initial activity decreasing over time, hinting at a possible transition in their roles, a pivot in the focus areas within the project or focuses on other projects. "Gaearon", represented in orange, shows a pattern of consistent contribution throughout their journey on the project, with spikes that may align with key development sprints or feature implementations. "Sophiebits", despite a later start, displays periods of high activity, which could align with significant responsibilities or major project phases.

4.2 Top Repository Patterns

Examining the top 980 repositories on GitHub, based on file count, featuring their attributes, to obtain a comprehensive overview of each repository. Based on file size, the dataset contains attributes of the top 2.2 million files on GitHub. It contains the "Owner", "repo", "filePath", and "fileSize". The top repositories include Apple (25,449), Apache (21,953), and Alibaba (16,275).

Based on the previous paragraph and the plot, you could talk about the distribution of the top 980 repositories on GitHub in terms of file count by their owners. Figure 7, reveals the percentage share of the total file count attributed to each of top 10 repositories. Apple has the largest share with 30.1%, followed by Apache at 25.9%, and Alibaba at 19.2%. This reveals that the organizations of the repositories have substantial resources and capabilities to maintain such extensive repositories. On the other hand, the rest of the file count is spread out among seven other owners.

Following the file count distribution pie chart, Figure 8 shows the distribution of repositories by file size sum, offering a different perspective of the top repositories. In this view, the font repository holds

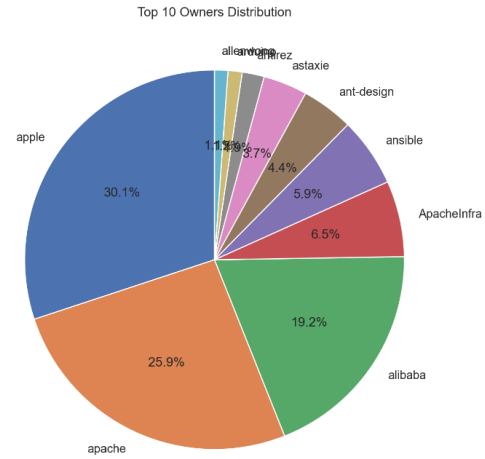

Figure 7: Top repositories based on file count

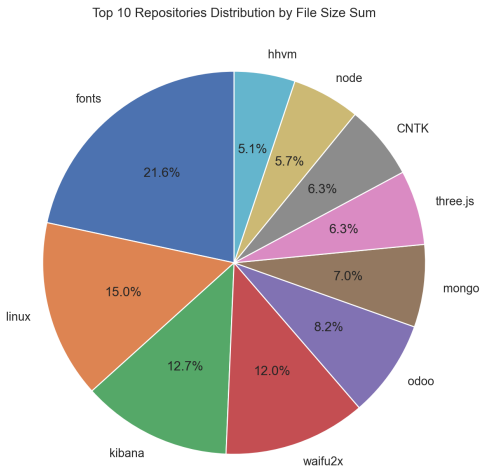

Figure 8: Top repositories based on File Size Sum

the largest share with 21.6% of the total file size, followed by LINUX at 15%, and KIBANA at 12.7%. This distribution indicates that while some repositories may have a higher number of files, others may have fewer but larger files, contributing significantly to the repository's total size. 'hhvm', 'node', and 'waifu2x' also have notable shares, suggesting they contain substantial file sizes within their repositories.

4.3 Repository File Structure

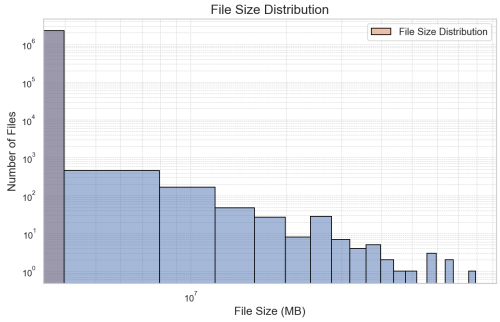

Figure 9: File Size vs. Number of Files

We initially looked at the distribution of the filesize, in figure 9 and found it was rightly skewed, indicating that a majority of the files, were small sizes, as evidenced by the tall bar at the far left of the histogram. As file size increases, the number of files decreases sharply. Overall figure 9 provides a clear representation of a inverse relationship between file size and file count.

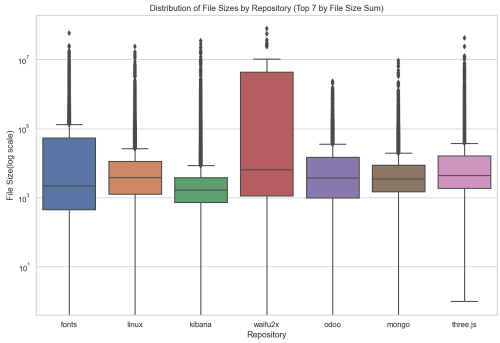

Figure 10: File Size Distributions of Top Repositories

Figure 14 shows the file size distribution across seven different repositories, highlighting the variability and range of file sizes within each repository. For instance, the 'waifu2x' repository has a notably larger median file size and greater spread of file sizes compared to others like 'fonts' and 'linux'. The presence of outliers, indicated by the diamond-shaped points above the whiskers, suggests that exceptionally large files could influence the repositories' average file size.

To help gain insights into the owner's collaboration methods, we examine the file extensions to determine the file types. Analyzing file extensions within software repositories can provide insights into the collaboration methods and development focus. The ratio of code files (.java, .py, etc.) to documentation files (.md, .txt, etc.) can reveal the importance of documentation and code readability within the project. Additionally, a high number of test files (.spec, .test, etc.) could indicate a commitment to testing and quality assurance practices like Test-Driven Development (TDD), a technique where developers write tests for new features before writing the code that implements those features. File extensions associated with version control and project management highlight the collaborative tools, suggesting a strategic approach to team coordination.

We started by categorizing files within the dataset based on their paths, distinguishing between the following file types: coding, documentation, or test files. By iterating through all 2.2 million files and extracting the file type, then depending on the file path structure and the file type, it labels it either as coding, documentation, or a test file. We then tally the number of coding versus documentation files, thereby quantifying the ratio of code to accompanying documentation. It can also reveal patterns specific to each repo owner, as some owners may prioritize extensive documentation, while others may focus more on code development.

Figure 11: File Category Distribution

There were 1.823 million coding files and 322k documentation, showing a significant majority of code files over documentation files, with a code-to-documentation ratio of 5.66. The insights gained highlight a potential emphasis on coding efforts over comprehensive documentation, aligning with common software development practices. Further exploration into specific repositories of documentation files could provide a better understanding of the dataset, guiding considerations for project management, and collaboration strategies.

Looking at the distribution of the top 20 file types, it reveals that .js (JavaScript files) are the most numerous, followed by .txt, .ts (TypeScript files), and others. This distribution supports the earlier obser-

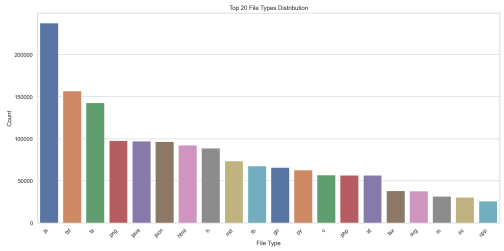

Figure 12: File Type Distribution

vation that coding efforts are extensive, as code-related file types like .js, .ts, and .java appear prominently. The presence of .txt, and .md (Markdown files), which are often used for documentation, is also visible but significantly less frequent in comparison to code files, which strengthens the idea that the repositories focus more on writing code than on creating documentation.

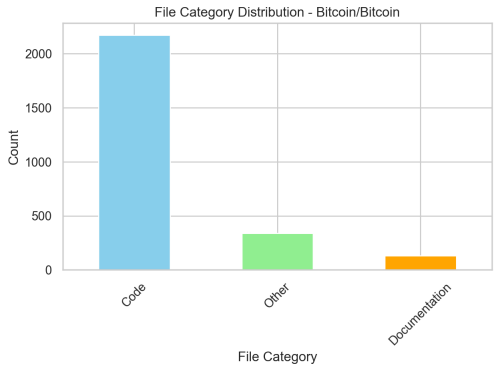

Figure 13: File Extension Distribution - Bitcoin Repository

To look into the file type distribution of a specific repository, we randomly selected the BITCOIN repository. The resulting barplot reveals the distribution portrays a repository that is heavily development-oriented, with a strategic approach to documentation and supplementary resources, reflecting the project's focus on innovation while ensuring its maintainability and collaborative accessibility.

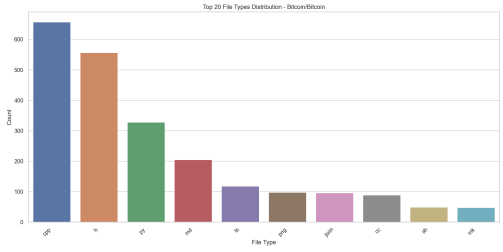

Figure 14: File Size Distributions - Bitcoin Repository

To investigate the file extension distribution of the BITCOIN repository. The displayed plot categorizes the top 20 file types, where .cpp (C++ source files) and .h (C/C++ header files) are the most prevalent, suggesting a heavy focus on C++ development. The presence of .py suggests testing or automation within the repository. Other extensions like .ts, .png .json, .cc .sh, and .mk indicate a diverse development that incorporates various technologies and scripting to support the repository's infrastructure and build system.

4.4 Power BI Dashboard - Predicting Repository Status

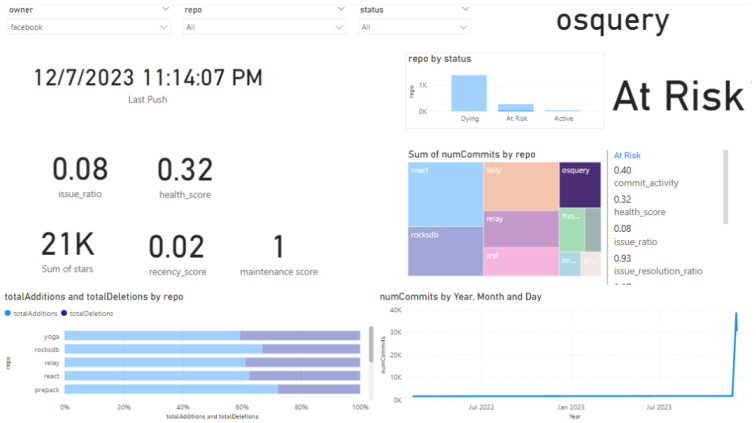

Figure 15: Facebook/Osquery - Power BI Dashboard

To help us visualize and monitor the metrics of a specific repository, we turned to creating a Power BI dashboard. Utilizing Power BI for this purpose brings several advantages, primarily its capacity to query through large datasets efficiently and its analytical features which allow us to visually identify trends and patterns in the data. The interactive capability of Power BI can allow users to focus on any specific repository, or to drill down into specific areas of interest, such as a predicted repository status. A visual and dynamic approach will allow for a more intuitive understanding of a repository's health and activity levels.

This dashboard showcases a suite of informative metrics and visualizations: the 'Last Push' metric provides the timestamp of the latest update; 'issue_ratio', 'health_score', 'recency_score', and 'maintenance score' offer insights into the engagement and maintenance levels of each repository. The 'Sum of stars' reflects the community's interest and popularity. The 'repo by status' visualization categorizes repositories into statuses such as 'At Risk' or 'Active', and the 'Sum of numCommits by repo' graphically breaks down commit volumes by color. A horizontal bar chart of 'totalAdditions and totalDeletions by repo' clearly contrasts code contributions against removals, indicating the development activity. Lastly, the 'numCommits by Year, Month, and Day' trend line tracks commit frequency, allowing for the identification of engagement patterns at a glance.

To help us test out Power BI's machine learning capabilities for our purposes, we calculated a metric called "Health_Score." The development of this metric primarily served as a test case to explore the application of machine learning within Power BI. "Health_Score" was calculated using a weighted formula, where different aspects of a repository's activity and maintenance are taken into account to assess its overall health. The weights were defined as follows: alpha at 0.3 for normalized stars, which could indicate the

popularity or approval of the project within the community; beta also at 0.3 for commit activity, to represent the ongoing development efforts; gamma at 0.2 for issue resolution ratio, as a measure of the project's responsiveness to issues; and zeta at 0.2 for the maintenance status score, reflecting the upkeep of the repository.

The results from the machine learning model within Power BI, using a LightGBM classifier, indicate a strong performance in predicting repository statuses. The model boasts a high area under the curve (AUC) score of 82%, reflecting a robust capability to differentiate between the classes of repository status, which are 'Dying', 'At Risk', and 'Active'. The predicted results show exceptional precision at 100%, meaning that all the repositories the model predicted as 'Dying' were indeed in that category. Additionally, the recall is at 90%, indicating that 90% of the repositories that are actually 'Dying' were correctly identified by the model.

Even if the health score was created as a test metric, its application provides a proof of concept that machine learning, particularly LightGBM classifiers, can be integrated effectively within Power BI to produce meaningful, actionable insights. The Power BI dashboard, while useful for visualizing and monitoring repository metrics, was ultimately not adopted because the "HealthScore" metric derived from the machine learning model did not yield reliable predictions to guide our decision-making processes.

4.5 Time Series Analysis - Forecasting Repository Status

To help us, predict the status of a repository, we turned to testing a time series plot since it can capture the dynamics of the past commit activity, reflecting both regular patterns and the outliers. We choose to predict the "time_between_commits", to try to look into the development cycle's regularity, detect future periods of inactivity, and potentially identify signs of a declining project.

In creating the time series forecast for the ALIBABA/WEEX repository, I first checked the data for stationarity using the Augmented Dickey-Fuller test, as non-stationary data can lead to unreliable predictions. Upon finding the data to be non-stationary, I applied differencing to stabilize the mean of the time series.

Using an ARIMA (AutoRegressive Integrated Moving Average) model to forecast future values, I chose the model's parameters (p=1, d=1, q=0) to fit the differenced data, which specifies a first-order autoregressive model with one differencing step and no moving average component.

Once the model was fit to the historical data, I forecasted the "time_between_commits" for the next 12 months. Figure 16 shows the historical data as a solid line and the forecast as an orange dashed line, providing a clear visual prediction of future commit activity patterns for the ALIBABA/WEEX repository.

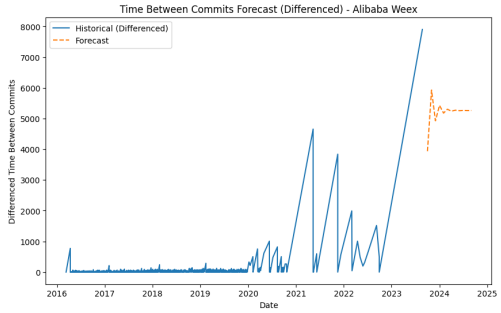

Figure 16: Forecasted Time Between Commits Forecast Alibaba/Weex

The time series plot displays that the time between commits is expected to remain at relatively high levels. This could be s strong indication that the repository is experiencing a decline in activity, as the model predicts a continuation of the trend of infrequent commits.

4.6 "Dead" or "Alive" Repositories

Determining the status of a repository as "dead" or "alive" can be crucial for developers and organizations to understand which projects are actively maintained and which have become stagnant. This status can affect decisions on dependencies, investment of time, and the overall trust in the codebase. In the process of determining which attributes would best represent the status of repositories, we employed a heuristic approach, taking into account several key indicators of activity.

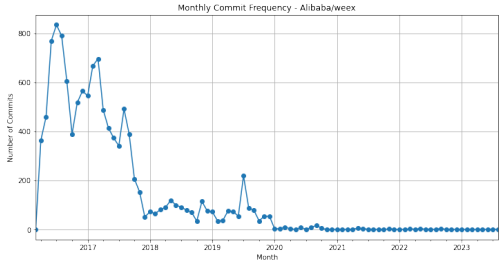

Figure 17: Monthly Commit Frequency Alibaba - Weex

To help build a intuition of how to define a strong heuristic, we choose to look at Alibaba's WEEX repository. Firstly, looking at the monthly commit Frequency and the time since the last commit of WEEX in Figure 17. The plot shows a declining trend in the number of commits over time, with a peak in activity occurring in early 2017. After this peak, there is a noticeable drop, with commit activity reaching a low, stable state after 2019. This pattern of decline and leveling off suggests that the repository has transitioned from a state of active development to one of minimal activity, which could be indicative of a "dead" or inactive repository.

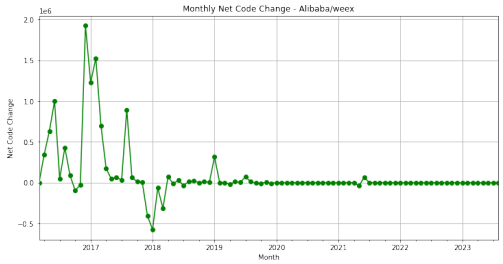

Figure 18: Net Code Change "Alibaba - Weex"

Following the discussion of the monthly commit frequency, we can look into the 'Net Code Change' and 'Time Difference Between Commits' for the WEEX repository to further predict its activity status. The 'Net Code Change' plot indicates the volume of changes in the code base over time. A downward trend or consistently low levels of code change, as shown in the graph, can be a strong indicator of reduced development activity, which may suggest the repository is becoming inactive or "dead."

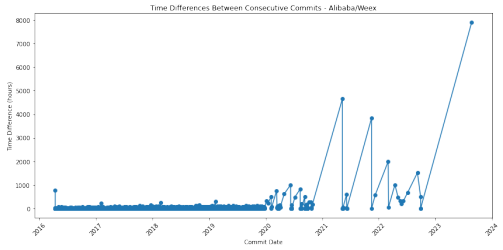

Figure 19: Time Difference between Commits "Alibaba - Weex"

Additionally, the 'Time Difference Between Commits' graph illustrates the intervals between consecutive commits. Increasing time gaps between commits, especially if they are significant compared to the project's historical data, could signal a decline in active maintenance. In the case of WEEX, the visible upward trend in time differences in recent years aligns with the prediction of a decreasingly active or "dead" repository. Together, these metrics paint a comprehensive picture of the repository's health and can be critical for stakeholders assessing the vitality and potential longevity of the project.

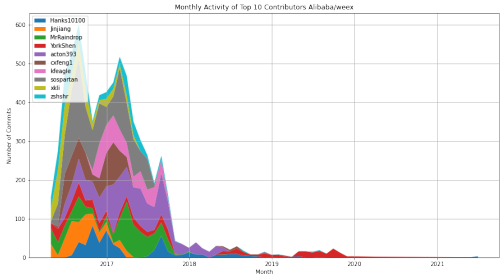

Figure 20: Weex's Author Contribution

Building on the analysis of the 'Net Code Change' and 'Time Difference Between Commits' plots, the 'Monthly Activity of Top 10 Contributors' chart further explains the activity levels within the WEEX repository. This stacked area chart details the monthly committed contributions of the ten most active contributors over time. Initially, the chart reveals a vibrant and collaborative environment with a high volume of contributions peaking around 2017. However, there is a noticeable decline in activity over the subsequent years. By 2019, the contributions have significantly tapered off, with minimal input from the previously active contributors. This decline in contribution activity, particularly from key developers, underscores the earlier observations suggesting a reduction in active development and possibly indicating a transition of the repository towards inactivity.

4.7 Heuristic for Labeling Repository Status

The analysis of the Alibaba Weex repository played a crucial role in creating a heuristic based on commit timestamps for labeling repository status as "Alive" or "Dead". By examining the various indicators of repository activity, including monthly commit frequency, net code changes, intervals between commits, and the contribution levels of the top contributors. The patterns of each indicator highlighted a clear shift from an actively developed state to one an inactive repository, thereby revealing the critical role of commit timing and frequency as indicators of a repository's health. Consequently, this led me to adopt a heuristic centered around commit timestamps, specifically analyzing the intervals between successive commits, the maximum historical interval, and the duration since the last commit against a reference date. This approach provides a nuanced and dynamic method for classifying repositories, leveraging their unique activity patterns to accurately assess their current state.

We utilize the commit timestamps for each repository to label the status of a repository as either "Alive" or "Dead". Let's denote the sequence of commit timestamps as $\{X_i\}$. Then, we define the following terms:

$$\begin{aligned} \text{Interval between commits:} &\quad Y_i = X_{i+1} - X_i \\ \text{Maximum interval:} &\quad Z = \max(Y_i) \\ \text{Last commit to collection date:} &\quad T_{X_n} = T - X_n \\ \text{Reference Date:} &\quad T \text{ (ex. November 1, 2023).} \end{aligned}$$

This heuristic method for determining the status of a repository as "Alive" or "Dead" focuses

on the patterns within the commit timestamps. By calculating intervals between successive commits and identifying the maximum historical interval, we can establish a baseline for the repository's activity. A repository is classified as "dead" if the time since the last commit is more the maximum interval, suggesting an abnormal drop in activity. Conversely, if the last commit occurred within the 95th percentile of all intervals, the repository is likely still active. This percentile threshold adapts to the repository's unique rhythm, accommodating for both consistently active projects and those with naturally sporadic update cycles.

Repository Status Categorized as:

$$\text{"dead" if } T_{X_n} > Z$$

$$\text{"alive" if } T_{X_n} < \text{percentile}_{95} \text{ of commit intervals}$$

"Threshold" refers to a value that distinguishes between an "alive" and a "dead" repository based on the timing of commits. Specifically, it is the 95th percentile of all commit intervals for a given repository. If the time since the last commit is less than this value, the repository is considered "alive"; if it's more, the repository is considered "dead." This percentile value is key because it is dynamically set according to the commit history of each repository, allowing for a tailored assessment that accounts for the different activity patterns across projects. We performed two tests, one where we looked at all of the repository's commits, "Overall Percentile Threshold" and another one where we only looked at the most recent k commits, "High Percentile Threshold":

$$\begin{aligned} \text{Overall Percentile Threshold (test1):} &\quad \text{All commit history} \\ \text{High Percentile Threshold (test2):} &\quad \text{Most recent } k \text{ commits} \end{aligned}$$

4.8 Heuristic Results

The use of overall and high percentile thresholds provides a two-pronged approach to understanding activity trends, considering the entire commit history as well as placing emphasis on recent activity. This dual analysis helps in discerning whether a dip in commit frequency is a recent trend or part of a longer-term decline. It's a strategy that leverages historical commit behavior to predict current status. However, it's most effective when repositories have a regular pattern of commits, and may not fully account for projects with erratic update histories or those that are complete and require minimal updates.

The "Unknown" category in this classification method captures repositories that don't fall into "Alive" or "Dead" based on commit activity. A repository is "Alive" if recent activity is within the 95-th percentile of intervals, and "Dead" if the gap since the last commit is more than the longest past interval. "Unknown" covers repositories that don't meet these criteria, acknowledging those in less active maintenance or with irregular updates.

We decided to choose November 1, 2023, as the reference date T for evaluating repository activity is strategic, even though the collection date of repository data concludes around January 1, 2024, primarily due to the seasonal work patterns that typically occur around the end of the year. During the Christmas and New Year's period, many companies experience a slowdown or a complete halt in committed activity as employees take time off, which can introduce a bias in the data analysis, and throw off our predictions. This period can often coincide with project planning cycles and fiscal year-end reviews, leading to a natural dip in commits as ongoing projects are concluded and new ones have yet to begin. By only using data up to November 1 2023, we sidestep the irregularities caused by the holiday season, thereby ensuring a more accurate reflection of a repository's activity.

4.8.1 Heuristic Overall Percentile Results

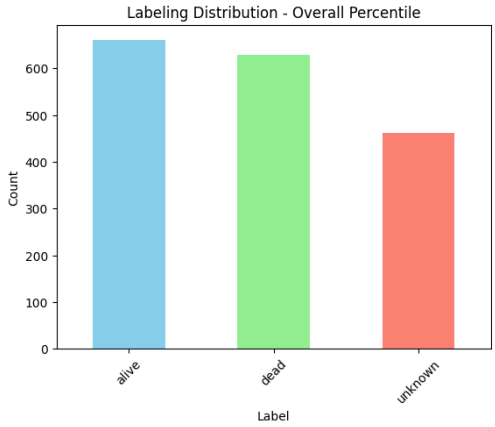

Figure 21: Repository Status Distribution Count - Overall Percentile

The bar plot illustrates the results of applying the heuristic method to the commit history of the 980 top repositories, quantifying the status of each as "Alive," "Dead," or "Unknown." The counts represented by the bars convey the distribution of these predicted statuses across the entire dataset. Shows how many repositories are actively maintained, which have fallen into disuse, and which could not be conclusively categorized. The presence of a substantial "Unknown" category underscores the complexity of such classifications and indicates that a significant number of repositories exhibit patterns that do not neatly conform

to the defined “Alive” or “Dead” criteria.

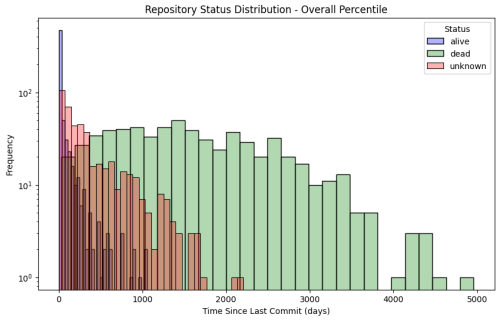

Figure 22: Predicted Repository Status Distribution - Overall Percentile

The histogram in figure 22 depicts the distribution of repository statuses across the set of 980 top repositories. Each bar corresponds to the number of repositories with the specified time since the last commit, categorized as “Alive,” “Dead,” or “Unknown,” labeled in blue, green, and red, respectively. The x-axis represents the days since the last commit, and the y-axis shows the frequency of repositories on a log scale as well to better visualize repositories with lower counts. This plot reveals not only the prevalence of each status but also how the time since the last commit correlates with the likelihood of a repository being labeled as active or inactive. A large number of repositories are classified as “Alive” with their last commit occurring relatively recently, as indicated by the cluster of bars on the left side of the plot, suggesting a healthy level of ongoing activity within a substantial portion of the dataset. Repositories labeled as “Dead”, tend to have a longer time since the last commit, as shown by the bars shifting rightward, signifying a decline in activity. There is a noticeable count of repositories with an “Unknown” status, particularly in the middle range of the time axis, which indicates that these repositories did not fit neatly into the “Alive” or “Dead” categories.

4.8.2 Heuristic High Percentile Results

Figure 23 illustrates the results of labeling the repositories using a high percentile threshold in our heuristic test, looking at the most recent 30 commits. The ’dead’ category, shown in green, has the highest count, suggesting that a significant number of repositories have not had recent activity within the 95th percentile of commit intervals, and thus are considered inactive. The ’alive’ category, in blue, represents repositories with recent commit activity falling within the 95th percentile of commit intervals, indicating they are actively maintained. The ’unknown’ category, in red, accounts for repositories that do not fit neatly into the ’dead’ or ’alive’ categories, potentially due to irregular or infrequent commit patterns.

The histogram in this figure 24 illustrates the distribution of repository statuses across a set of

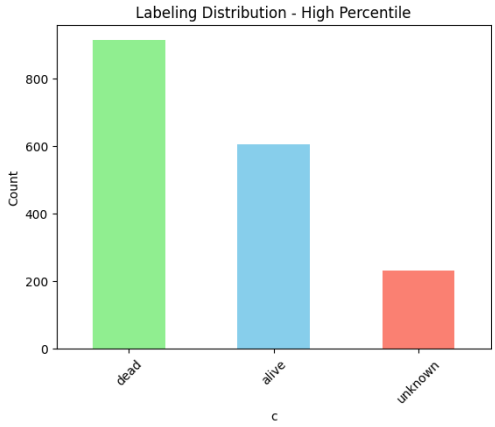

Figure 23: Labeling Distribution - High Percentile

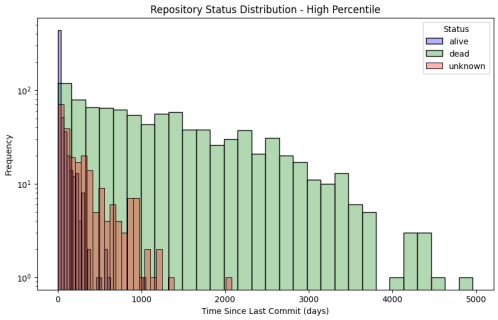

Figure 24: Repository Status Distribution - High Percentile

repositories, based on the last 30 commits. This visualization highlights the relationship between the recency of commits and the current activity status of the repositories. The cluster of blue/brown bars on the left side of the plot indicates a substantial number of repositories calculated as "Alive," suggesting an active and maintained state. A notable observation is the presence of green bars, labeled as "Dead" repositories, found on the left side of the plot where we would expect to see "Alive" repositories. This suggests that several repositories with recent commits are being incorrectly labeled as "Dead". The unexpected placement of green bars among the recent commits suggests that the heuristic may have limitations to avoid misclassification.

Upon reviewing both tests, we decided to proceed solely with the overall percentile heuristic. This decision was informed by the observation that the overall percentile heuristic resulted in a more accurate classification, as seen in its corresponding histogram. The key difference noted was the reduced presence of green bars, indicating "dead" repositories, towards the lower end of the histogram, which signifies repositories with recent commits. This suggests a more precise alignment between the actual repository activity and its classification.

4.8.3 Heuristic Validation

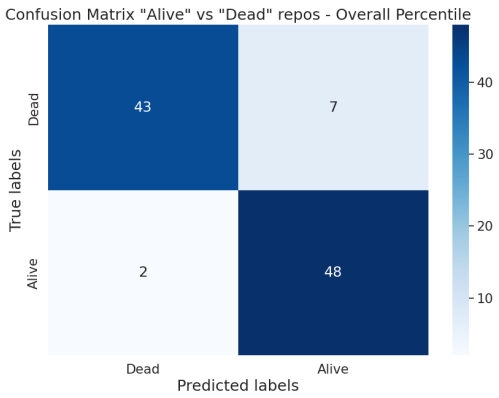

Figure 25: Repository Status Validation Confusion Matrix - overall percentile

To help validate the prediction of the status, 100 repositories were selected at random—50 predicted as "Alive" and 50 as "Dead." Each repository underwent a thorough review of commit history, active issues, and discussions within the repository to establish a consensus on its actual status.

The confusion matrix plot serves as a validation tool, displaying the accuracy of the predictive model against the manually reviewed sample. The matrix shows the number of repositories correctly predicted to be "Alive" (true positives) and "Dead" (true negatives), as well as those incorrectly predicted (false positives and false negatives). In our sample, 43 out of 50 repositories predicted as "Alive" were confirmed to be active,

while 48 out of 50 predicted as “Dead” were accurately predicted, indicating a high level of precision in the model’s predictions. The low numbers of false positives and false negatives suggest that the model is reliable. Some repositories classified as “Alive” were incorrectly predicted as “Dead” due to their irregular commit patterns and intervals. However, it’s worth noting that in certain instances, there were over 10 repositories that had been inactive for over 300 days, presumed to be fully inactive, experienced a revival, and were recently updated in 2024. For example, the repository “pythondict-quant,” maintained by “Ckend,” as of November 1, 2024, hadn’t made a commit in 654 days, but it unpredictably recently came back to life and made a commit on January 7, 2024. Additionally, the repository ’isucon9-qualify’, owned by ’isucon’, where was inactive since Oct 1, 2020, for 1125 days, and came back to life on Jan 13, 2024, and has been highly active ever since.

4.8.4 Resurrected Repositories

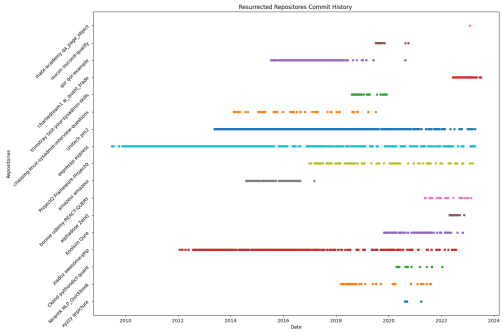

Figure 26: Resurrected Repositories Commit History

Figure 26 illustrates the varied lifespans and development rhythms across repositories, reflecting the unique circumstances and motivations driving each project’s contributors. When looking at the recommit history of all “resurrected” repositories, it’s crucial to discern some of the circumstances, on the cause of a revival, whether they are attributable to random occurrences or common trends. A repository can spring back into action for various reasons, such as a renewed need due to technological advancements, funding, essential updates for compatibility, or even new interests within the community.

5 Data Collection Tool

This creation of the data collection tool resulted in a program that can be easily configured to perform GitHub API exploration and collection, manual entry data collection, and small quality-of-life functionalities to improve the user’s experience. The tool has a set control flow for collecting relevant data

for manually entered repositories and discovered repositories from a preset number of queries. The program follows the control flow described by Figure 27:

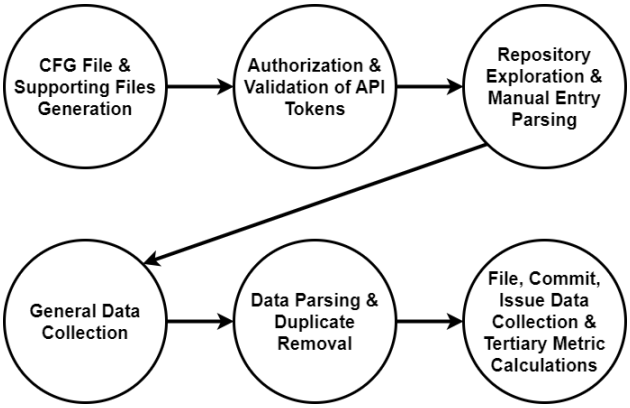

Figure 27: Control Flow of the Data Collection Tool

A readme was created and stored on our GitHub repository for this tool that explains the functionality of this tool, notes for future developers, and future plans if this tool were to be further developed with more functionality implemented.

Prior to launching the program the user writes the necessary authentication information to "Auth.py" which will allow them to interface with the GitHub API. This information can either be an free API token from GitHub or the path to an enterprise token's .pem file alongside the GitHub application ID. The enterprise token takes higher priority than the free token when the program operates.

The user is greeted with the following interface seen in Figure 28. This interface allows the user to configure what data is collected by the program. It offers languages that match the availability of the other complexity calculation tools, pydriller and lizard that we used for this project. If no languages are selected, there is no exploration with GitHub's search API and only manually entered repositories are processed. It also allows the user to limit the number of commits from explored repositories. *Example.* If a repository were to have 10,000 commits and the limit was 5,000 commits, the repository would not be noted. Finally, the user can specify if the repositories they want to explore through GitHub's search API should be sorted by any combination of stars, forks, or help-wanted-issues with stars being the default. At the closing or the use of the "generate" button, a file named "cfg.json" will be generated with the current selections. This file will hold the configuration of the program as it executes. It must be noted that if the user has specific repositories to collect data on, the link to the repository on GitHub should be manually entered to the "cfg.json" under "priorityRepos" in the JSON file generated after running the program for the first time.

After the above is completed, the user should re-run the program and this time will be greeted by

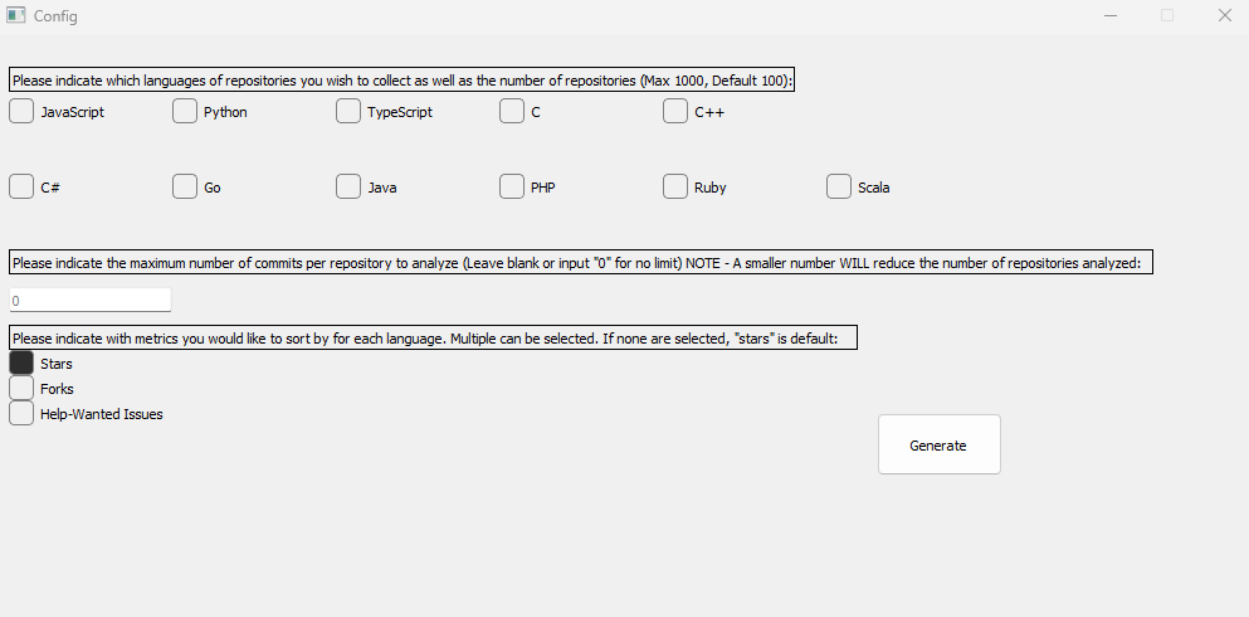

Figure 28: The UI for the configuration of the data collection tool

an acknowledgement of the limitations of the program and the GitHub API which are described in Section 7 and the readme. Upon the second running of the program, the necessary supporting data folder and files will be generated. These files and all future data references follow the same data definitions described in Appendix A. The authentication information provided by the user is then ran against the GitHub API with a general query. The response headers of this general query are analyzed to confirm the validity and type of token provided. The program then parses the configuration file into the program state to determine which queries are used as well as other limitations including commit count. The manually entered repositories in the configuration file are then validated to exist publicly on the GitHub API. In the case that they do not exist, they are voided from the program.

At this point the program begins data collection by collecting the "general data" for each manually entered repository and begins executing the GitHub API search queries to collect more repositories based on the criteria from the configuration file. The "general data" is then collected for the discovered repositories as well. This collected data is then parsed to remove duplicate repositories as well as repositories that violate any restrictions from the configuration file. It is important that we state that the GitHub API does not have any metric that provides the number of commits under a repository. We engineered a solution using a modified binary search to find the first commit under a repository which could tell us the number of commits under a repository based on the page that commit was found on.

The program then takes all the parsed repositories and iteratively collects and calculates various

metrics across all relevant data from a repository. A repository's files are collected by traversing their master branch git tree, a data structure implemented by GitHub. This structure allowed us to recursively traverse the tree and build our own structure of all the files where there are directories (trees) and files (blobs). Using the data provided a tertiary statistic of file extension occurrences and sizes for each repository. Finally, the commit data and issue data is collected for each repository by iterating through the pagination provided by the GitHub API.

Additional functionality was added to this program as it collected data and we ran into issues. One of the first major issues we encountered was HTTPS errors that our requests encountered. Several catches were implemented to ensure proper collection. One of the most successful implementations was having 10 retries per query upon HTTPS error. However, this was not always a sufficient solution so error logging was implemented to inform us of API errors in the data collection. We could not implement error handling due to time restrictions. Another issue encountered regularly during testing was the possibility of the program being interrupted during data collection. We implemented a save state and auto-saving for the program so there would be no interruptions if the program crashed. This save state is stored in several cache files that save and load the program state so there is minimal redundancy in the data collection.

Finally, extra commands were implemented as quality-of-life functionality for any future user of this program. A list of the commands and their explanations can be found below.

- **-help** - This command explains all the possible functions that are present in the program
- **-rm** - This command resets the program by removing all the data in the Data folder and includes an additional option to reset the "cfg.json" file.
- **-zip** - This command allows users to easily generate a compressed archive of all the data they have collected using the tool.
- **-acc (arg)** - This command allows for the express collection of data for an individual repository. The argument is the GitHub link to the repository to be collected.

6 Results

This section will cover the results of the data collection process, the data analysis, and the results of the training models.

6.1 Dataset

We used the Data Collection Tool for over 30 days and retrieved over 7GB of text data for over 1,700 repositories from the GitHub API regarding relevant repositories and stored the data in CSV files using a repository owner's name and the repository's name as the primary key.

6.2 Regression Task: Complexity Metrics Prediction

Our first machine learning task was to train regressors on the Maintainability Index, Cyclomatic Complexity, and Halstead Volume using repository features, graph encodings, and both repository feature and graph encodings, the results of which are displayed below.

6.2.1 Repository features Regressors

The models that were used for the regression task were Lasso Regression, Random Forest, k-nearest neighbors, and XGBoost. Hyperparameters for all models were fine-tuned utilizing random numbers generated from a tool that sklearn uses called Randit to achieve optimal performance, employing a best-fit selection approach and using the Random Search CV from sklearn [39]. The results for the Maintainability Index can be seen in Table 1. The presented results indicate that Random Forest performs relatively better in predicting the Maintainability Index compared to the other models, with an average MAPE of 54.45%. However, KNN, Lasso Regression, and XGBoost also demonstrate their effectiveness with average MAPE values of 60.28%, 63.55%, and 59.50% respectively.

Table 1: Model Performance on Repository Features: Maintainability Index

Model	**Mean Squared Error**	**R^2 Score**	**MAPE (%)**
Random Forest	**54.45**	**0.14**	**10.56**
KNN	60.28	0.04	11.17
Lasso Regression	63.55	-0.01	11.40
XGBoost	59.50	0.06	11.06

Table 2 contains the accuracy information for regression models trained on repository features with the target of Cyclomatic Complexity. The models, including Random Forest, KNN, Lasso Regression, and XGBoost, struggle to accurately predict Cyclomatic Complexity, as evidenced by high mean squared error and MAPE values, coupled with low R^2 scores. Despite variations in performance metrics, all models demonstrate limited ability to capture the underlying patterns of Cyclomatic Complexity, suggesting potential challenges in the feature set or modeling approach.

Table 2: Model Performance on Repository Features: Cyclomatic Complexity

Model	Mean Squared Error	R^2 Score	MAPE (%)
Random Forest	2.61	0.05	**42.76**
KNN	2.54	**0.08**	43.15
Lasso Regression	2.75	-0.00	46.03
XGBoost	**2.46**	0.04	45.49

We have omitted the results for our predictions on Halstead Volume as many of the models failed to converge. Despite Random Forest showing relatively better performance compared to other models, the overall predictive capability remains unsatisfactory for Cyclomatic Complexity.

6.2.2 Graph Features Regressors

The next step was to train our models on the graph encodings generated from the GAE and on the concatenation of the graph features with the encodings of the node information. Tables 3 and 4 below show the performance on only the encodings for both Maintainability Index and Cyclomatic Complexity.

Table 3: Model Performance on Graph Encodings: Maintainability Index

Model	Mean Squared Error	R^2 Score	MAPE (%)
Random Forest	63.50	-0.01	11.39
KNN	63.07	-0.00	11.28
Lasso Regression	63.05	-0.01	11.40
XGBoost	**61.32**	**0.03**	**11.12**

Table 4: Model Performance on Graph Encodings: Cyclomatic Complexity

Model	Mean Squared Error	R^2 Score	MAPE (%)
Random Forest	**2.75**	**0.00**	**45.98**
KNN	2.82	-0.03	46.52
Lasso Regression	**2.75**	**0.00**	46.03
XGBoost	3.02	-0.10	51.33

Comparing the performance of the models which use only the graph encodings as features, we find that the performance of the Random Forest and Lasso models were best for estimating the Maintainability Index while Random Forest showed the best performance for Cyclomatic Complexity. Next, we incorporated the original features with the graph encodings for training the models, the results of which can be found in Tables 5 and 6.

Once again, the performance of all models on Cyclomatic Complexity was subpar. With the graph encodings, we saw a slight increase in every model but Lasso for Maintainability Index and Cyclomatic Complexity.

Table 5: Model Performance on Test Set and Graph Encodings: Maintainability Index

Model	Mean Squared Error	R^2 Score	MAPE (%)
Random Forest	**54.53**	**0.13**	**10.62**
KNN	60.28	0.04	11.17
Lasso Regression	63.55	-0.01	11.40
XGBoost	63.27	0.00	11.37

Table 6: Model Performance on Test Set and Graph Encodings: Cyclomatic Complexity

Model	Mean Squared Error	R^2 Score	MAPE (%)
Random Forest	2.62	0.05	43.41
KNN	**2.54**	**0.08**	**43.15**
Lasso Regression	2.75	0.00	46.03
XGBoost	2.57	0.06	43.96

6.3 Repository Status Classifiers

Our second task was to train classifiers on whether a repository was dead or alive once again using repository features, graph encodings, and both repository feature and graph encodings. The log loss values are displayed in Table 7 and confusion matrices for the best model with each dataset are displayed in Figure 29.

Table 7: Model Performance on Classification (log loss)

Model	Repository Features	Graph Encodings	Features and Encodings
Random Forest	4.68	16.37	4.68
KNN	8.57	15.98	8.75
Lasso Classification	16.36	16.36	16.36
XGBoost	**1.98**	**7.76**	**2.07**

The confusion matrix on the left is for the Random Forest classifier trained on only repository features, the confusion matrix in the center was for the XGBoost trained on only the graph encodings and the confusion matrix on the right was for the Random Forest Regressor trained on bot the repository features and graph encodings. As can be seen in the confusion matrices, XGBoost had a tendency to make false positive labels with the model trained on only repository features performed the best.

7 Limitations

This section will cover the limitations we encountered across the several branches of this project.

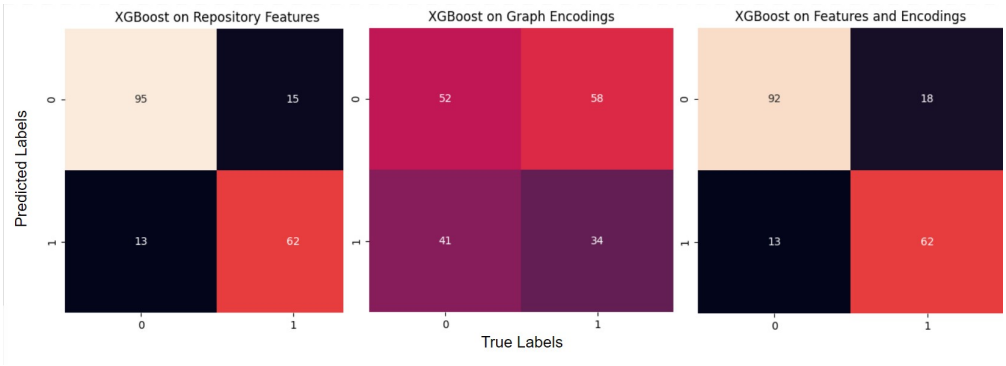

Figure 29: Confusion Matrices for the best models

7.1 GitHub API

Several limitations were encountered regarding the GitHub API during data collection. Firstly, there were hard caps to limit the data transfers over GitHub's API in many dynamically sized metrics. For example, files associated with individual commits were limited to a count of 3,000. The API would provide the first 3,000 files in alphabetical order for individual commits and the remaining files were truncated from the request. Another limitation of the API was the number of files in a directory under a git tree. There was a maximum of 100,000 files and a 7MB data limit for tree queries where the rest was truncated [40]. There was also hesitancy to multithread the API requests through the API as enterprise tokens were quite expensive relative to the funding of this project, which resulted in the data collection taking longer than it would have taken with more funding.

7.2 NetworkX

We faced two primary issues with NetworkX. When rendering the graphs, many of them were very dense making it so that the labels were unreadable, and distinguishing the nodes from each other was difficult. This was solved by reducing the number of nodes by applying restrictions on the commit dates, as well as by removing hidden files (files whose name follows the format ".<filename>.*") from the graphs. Additionally using the Kamada-Kawai and Bipartite layouts helped make graphs more readable.

The second issue we faced with NetworkX was converting our graphs from NetworkX to the format that DGL uses for storing graphs. DGL's API provides the function from_networkx(), however, it is warned that "Creating a DGLGraph from a NetworkX graph is not fast, especially for large scales"[38]. Since our graphs often had thousands of nodes, we altered the code to instead generate a CSV from which we could use the DGL function data.fromCSV() to construct our graphs rather than creating a NetworkX graph.

7.3 DGL

The primary issue that we faced when dealing with DGL was the scale on which our data was stored. Initially, we were storing graphs with every commit from every repository. However, this created CSVs $\tilde{6}$0GB in size which would not fit into memory on our computers. Eventually to decrease the size of the CSVs we only used the first 100 commits for any repository larger than 100 commits.

7.4 Repository Status

Due to the variability in repository activity patterns and the limitations of relying on commit timestamps, accurately labeling the life status of a repository as "Alive" or "Dead" poses significant challenges. There are challenges due to irregular commit patterns, where projects may not follow a predictable activity rhythm. The irregularities can stem from various factors, including the project's lifecycle phase, external dependencies, developer availability, and other external influences, which may not directly reflect commit activity. Moreover, the heuristic's reliance on specific metrics like the 95th percentile for commit intervals may not be suitable for all repositories. To overcome these limitations, a more meticulous approach may involve considering additional data sources, and the quality of contributions, looking into the developer forums, and understanding the broader context in which a repository exists, aiming for a more accurate and comprehensive assessment of a repository's status.

7.5 Code Quality Metrics Tools

One limit on the code quality metrics that were collected was that it was limited to 13 languages, thus it is not adjustable to every language that is found on GitHub. Due to that limitation, the models won't be able to capture the whole story of GitHub repositories and accurately predict the target variable. Additionally, many repositories on GitHub are used for simply storing text files so some repositories produced a null value for this reason as well.

7.6 Training Models

The models were not able to fully converge when running Halstead Volume on both the graph and repository features, which led to an incomplete full analysis of the code quality metrics. The failure of the models to converge when running Halstead Volume as the target variable could be caused by several factors such as inadequate ability to model the complexity between features and Halstead Volume or weak

correlation. Thus the tables have been omitted from the results section as they did not seem to benefit or explain any other behaviors that were tested.

8 Conclusion

In this study, we have explored the performance of various machine learning models in predicting Maintainability Index and determining repository status based on different sets of features. Our investigation has provided valuable insights into the strengths and limitations of these models, shedding light on their effectiveness in capturing underlying patterns within software repositories. Among that, we have also discovered many ways to use the API to collect data and ways to analyze different repositories through EDA. The next sections will summarize our findings and discuss their implications for software metrics prediction and repository management.

8.1 Analysis of Regressors

When analyzing the performance of different machine learning models for predicting the Maintainability Index based on repository features, we observe distinct behaviors. Random Forest stands out as the top performer, showing the lowest prediction errors and a reasonable ability to explain some variance in the Maintainability Index. On the other hand, XGM struggles to provide accurate predictions, indicating its limitations in capturing the underlying patterns in the data. Lasso Regression and KNN fall in between, with Lasso Regression showing similar results to Random Forest but with less ability to explain the variance. Overall, Random Forest emerges as the most reliable choice for this task, offering valuable insights into the maintainability of software repositories.

Across all models, the predictive performance for Cyclomatic Complexity remains suboptimal. The MAPE values ranging from 36.58% to 48.48% indicate significant discrepancies between predicted and actual Cyclomatic Complexity values. These results underscore the complexity of predicting software metrics and highlights the need for further research and refinement in feature selection, model development, and evaluation methodologies to enhance predictive accuracy and usefulness in practical software development contexts.

8.1.1 Comparison of Feature Sets for Regression Task

The comparison between the results obtained from the training models, trained with repository features and the models that are trained with graph features, showed distinct patterns. Regression, Random Forest, K-nearest Neighbors (KNN), and XGBoost were trained to predict the Maintainability Index based on 1300 rows of repository metrics. Notably, Random Forest emerged as the top performer, exhibiting the lowest mean squared error (MSE) and a reasonable ability to explain variance. Conversely, XGM struggled to provide accurate predictions, which could be due to its limitations in capturing underlying data patterns. Lasso Regression and KNN fell in between with MAPE values falling between those of the Lasso and KNN regressors.

For graph features, the performance remains consistent with those found when only using the found features, with Random Forest consistently outperforming other models for each of the target variables. In summary, the comparison underscores the robustness of Random Forest in leveraging both repository and graph features for software metrics prediction. Despite the incorporation of graph encodings, other models failed to outperform Random Forest, indicating its effectiveness in capturing complex relationships within the data. However, the consistent suboptimal performance of KNN suggests limitations in leveraging graph features for prediction tasks.

Additionally, we were also able to discern that the relationship between repository graph encodings and complexity metrics either does not exist or requires a far more complex model to have the power to describe the relationship between repository graph encodings and complexity metrics.

8.2 Analysis of Classifiers

In contrast to what we discovered in the analysis of our regressors, XGM Boost rose to the top of our classifiers for determining if a repository was dead or alive, whereas Lasso was consistently the worst. From inspecting the confusion matrices, it is clear that XGM tends to assign more false positives in detriment of false negatives.

8.2.1 Comparison of Feature Sets for Classification Task

Once again when comparing the different classifiers across the datasets, the models all performed best on only the repository features. with XGM doing very slightly better on only the repository features than when using both the repository features and graph encodings. One major difference from the regression

task can be seen in the models trained on only the graph encodings. During the regression task, the models trained on only the graph encodings showed a very small decrease in performance to the models trained on repository features and both repository features and graph encodings. In our classifiers however, the log loss for every classifier trained on only graph encodings other than Lasso showed either double or quadruple the logs loss compared to the classifiers trained on the other data. This has an even greater indication that the graph encodings either do not correlate with the classification task that we set out on, or we need to use a more in-depth machine learning algorithm to model the relationship between the encodings and the status of a repository.

9 Future Works

This section will cover the future works and improvements that we recommend for further research in similar fields or using our work.

9.1 Data Collection Tool

Many improvements could be created for our data collection tool as it was developed hastily to begin data collection as quickly as possible. We were able to outline some major improvements for this tool during the conclusion of this project.

Firstly, implementing a data updater would eliminate the current redundancy of acquiring new data. Once the program reaches a checkpoint in data collection, it cannot be redone or changed without resetting the tool. The updater should operate in the following manner.

1. Updating the number of commits and issues of a repository based on the general data.
2. Then, update the file structure data to the most updated files in the repository.
3. Finally, update the commit and issue data for the repository.

The main savings on processing time would be achieved by updating the commit and issue data as opposed to recollecting it all, by noting the last commit/issue collected (SHA, Date, or Number) and only collecting commit/issue data (stored chronologically on GitHub's databases) that hasn't been collected yet. This would also require sorting the issues and commits data by date after each successive collection.

Another improvement that could save hours is improving the save states of the program. Currently, the save states denote checkpoints by the repository that have finished most recently. This could be further

narrowed to the individual commit URL and page number per repository at each point during data collection. We encountered this issue during our data collection. A large repository crashed and we lost all the progress made when the program reverted to the last repository completed.

The final optimization discussed in this paper for the data collection tool is the implementation of multithreading. This would increase the efficiency of data collection and could be expanded to the limitation of the hardware used with enough enterprise tokens to significantly increase the rate limit.

9.2 Graph Auto Encoders

As primary improvements to the GAE we recommend changing what the GAE is trained to encode, and increasing the amount of data the GAE is trained on. GAEs can create encodings used for reconstructing graph structure (the edges between different nodes) or for rebuilding the original features of the nodes based on the encodings [37]. Our Autoencoder is currently trained to reconstruct only the structure of the original graph based on our encoded graph features. However, it could be more useful to develop a GAE which is trained to create encodings that decode to the original graph/node features.

As mentioned in the limitations section, the number of commits per repository had to be severely limited due to memory constraints. Currently, every graph for every repository is being loaded into memory at the same time. If instead the code is loaded in subsets so as not to overflow main memory, significantly more of the commit data for each of the repositories could be used to train the GAE. If this were to be implemented with stochastic gradient descent, as we are currently using to train the GAE the shuffling of data, it would have to be done intelligently when loading the data into memory.

9.3 Model Training

As mentioned in our conclusion, the models used did not seem to be able to find any correlation between the encodings generated by the GAE and the target variables for the tasks. This points to two possibilities:

1. There is no correlation between the generated graph encodings and the targets.
2. There is a correlation between the generated graph encodings and the targets, however, the models used were unable to capture the complexity of this relationship.

One solution to the first possibility is to change the architecture used for learn graph encodings. The easiest way to do this would be by changing the number of hidden features and number of hidden layers

in the current version of the GAE being used. The second solution would be changing the GAE to create encodings that are decoded to recreate the original features of the graph as discussed in the previous section.

The second possibility allows for a lot more wiggle room as many different machine learning algorithms can capture much more complex relationships between features and targets. Limited testing was done on a Feed Forward Neural Network which showed promising results. However, this had to be dropped due to time constraints.

9.4 "Resurrected" Repositories Trends

To enhance the research on resurrected repositories, students can analyze and look into any underlying patterns or randomness in these revivals. A sustained examination of committed activity over extended periods may reveal patterns or associations with shifts in the technology market. Integrating market trends into this analysis could unveil potential links between the revival of repositories and the prevailing demand for specific technologies within the industry.

Moreover, a more qualitative exploration into the reasons contributors return to inactive projects could provide a deeper understanding of the personal or community factors that cause a revival. Monitoring the activity on social media and developer forums could illuminate the role of community interest in rekindling activity. A study of how active project dependencies might influence the resurgence of other projects would also be possible. Such a multifaceted approach promises not only to broaden our understanding of open-source project dynamics but also to refine predictive models for future repository activity.

Appendices

A Data Set Definitions

Listed below are the data definitions for the data that was collected by our data collection tool. These were used to organize our CSV data storage. Commit Data

- **dateCollected** - the date the data point was collected
- **url** - API URL to the commit
- **sha** - the sha for the commit
- **owner** - the account name of the owner of the repository
- **repo** - the name of the repository
- **author** - the account name of the committer
- **date** - the date the commit was pushed
- **totalAdditions** - the number of lines of code the commit added
- **totalDeletions** - the number of lines of coee the commit removed
- **message** - the message for the commit
- **numFiles** - the number of files altered by the commit (3,000 max)
- **numComments** - the number of comments on the commit
- **commentsURL** - the url to the comments on the commit
- **TUPLE(fileName, status, additions, deletions, changes, raw_url, contents_url)** - the per-file data points in the commit

File Structure Data

- **dateCollected** - the date the data point was collected
- **owner** - the account name of the owner of the repository
- **repo** - the name of the repository

- **TUPLE(fileName, fileSize)** - the per-file data points in the repository

File Type Data

- **extension** - the file extension
- **size** - the number of bytes of a specific file extension in a repository
- **numFiles** - the number of files of a specific file extension in a repository

General Data

- **dateCollected** - the date the data point was collected
- **owner** - the account name of the owner of the repository
- **repo** - the name of the repository
- **cloneURL** - the cloneable URL of the repository
- **stars** - the number of stars a repository has
- **numOpenIssues** - the number of open issues a repository has
- **numIssues** - the number of issues a repository has had
- **dateCreated** - the date the repository was created
- **dateUpdated** - the date the repository was last updated in any way
- **datePushed** - the date the repository was last pushed to
- **numCommits** - the number of commits a repository has

Code Quality Data

- **Total nloc** - Total non-commented lines of code (NLOC) represent the total count of lines of code excluding comments and blank lines.
- **Avg.NLOC** - Average non-commented lines of code (NLOC) per file.
- **AvgCCN** - Average Cyclomatic Complexity Number (CCN) represents the average complexity of the codebase. CCN measures the number of linearly independent paths through a program's source code.

- **Avg.token** - Average tokens per line. Tokens are the smallest units of programming language syntax.
- **Fun Cnt** - Function count refers to the total number of functions or methods within the codebase.
- **File Threshold Count** - Represent a threshold value set for a certain metric (like file size, line count, etc.) beyond which additional actions or considerations are triggered.
- **Fun Rt** - Function rate represents the ratio of functions to total codebase size.
- **nloc Rt** - Non-commented lines of code (NLOC) rate represents the ratio of NLOC to the total codebase size.
- **Halstead Volume** - Halstead Volume is a software metric that quantifies the size of a program by considering the number of distinct operators and operands used in it.
- **Maintainability Index** - Maintainability index is a software metric that measures how maintainable (easy to modify and maintain) the codebase is based on various factors such as code complexity, code duplication, and code size.

Issue Data

- **dateCollected**
- **owner** - the account name of the owner of the repository
- **repo** - the name of the repository
- **author** - the author of the issue
- **authoAssociation** - how the author of the issue is associated with the repository
- **number** - the issue number of the repository
- **title** - the title of the issue
- **state** - the state of the issue (open, closed, etc)
- **locked** - if the repository has been locked or not
- **numComments** - the number of comments on the issue
- **commentsURL** - the URL to the comments of the issue
- **dateCreated** - the date the issue was created

- **dateUpdated** - the date the issue was updated in any way
- **dateClosed** - the date the issue was closed (if it was closed)
- **activeLockReason** - if the issue is locked, this is the reason why it was locked
- **totalReactions** - the total number of emoji reactions the issue has received
- **labels** - the developer-assigned labels for the issue
- **assignees** - the account names that are assigned to the issue
- **LIST reactions** - a list of values that are the specific emoji reactions allowed by GitHub in the following order:
  - +1
  - -1
  - laugh
  - confused
  - heart
  - hooray
  - rocket
  - eyes

B Langs JSON

Listed below are the languages contained in our JSON file. In our code this is converted to a one-hot encoding to be used in the GAE.

- **Javascript** - associated file extensions: .js, .jsx, .cjs, .mjs, .iced, .liticed, .cs, .coffee, .litcoffee, .vue, .less
- **Typescript** - associated file extensions: .ts, .tsx
- **Python** - associated file extensions: .py, .pyi, .pyc, .pyd, .pyo, .pyw, .pyz, .ipynb, .pyproj, .python, .pypirc
- **C** - associated file extensions: .c, .o, .h, .cmake
- **CPP** - associated file extensions: .cc, .cpp, .o, .h, .cmake, .a, .ui, .cxx, .hxx, .hpp
- **CSharp** - associated file extensions: .o, .h, .cs, .csproj
- **Go** - associated file extensions: .go
- **Java** - associated file extensions: .java, .class, .gradle, .properties, .expected, .jar, .jsp, .dpj, .kt
- **PHP** - associated file extensions: .php, .phps, .php3, .php4, .php5, .phtml
- **Ruby** - associated file extensions: .rb, .rhtml, .erb
- **Scala** - associated file extensions: .scala, .sc
- **Archive** example file extensions: .zip, .7z, .gz, etc.
- **Media** - example file extensions: .jpg, .mp4, .wav, etc.
- **Executable** - example file extensions: .exe, .sh, .bat, etc.
- **Data** - example file extensions: .txt, .json, .xlsx, etc.
- **Markup** - example file extensions: .gitignore, .html, .xml, etc.
- **Other** - any file extensions which do not fit any of the above categories

References

[1] M. Tech, "What is software engineering? — computer science — michigan tech."

[2] T. E. of Encyclopedia Britannica, *Margaret Hamilton — Biography & Facts*. 07 2019.

[3] L. Cameron, "Margaret hamilton: First software engineer — ieee computer society," 2016.

[4] M. Martinez, "50 years of software — ieee computer society," 01 2019.

[5] Sonatype, "What are software dependencies — sonatype,"

[6] I. Robertson, "Floppy disk - an overview — sciencedirect topics," 1994.

[7] Atlassian, "What is version control," 2019.

[8] T. McMillan, "A history of version control," 09 2021.

[9] J. Miller, "Which version control should developers use? — bairesdev."

[10] Git, "Git - book."

[11] Git, "Git - a short history of git."

[12] J. Errickson, "Getting start with git."

[13] GitHub, "Hello world." https://docs.github.com/en/get-started/start-your-journey/hello-world.

[14] GitHub, "Global distribution of developers."

[15] I. GitHub, "Build software better, together," 2019.

[16] Synopsys, "What is open source software and how does it work? — synopsys," 2023.

[17] "About: A2 (operating system)."

[18] M. Volpi, "How open-source software took over the world," 01 2019.

[19] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pp. 9–9, 2007.

[20] M. H. Halstead, *Elements of software science*. North-Holland, 1979.

[21] C. T. Bailey and W. L. Dingee, "A software study using halstead metrics," in *Measurement and evaluation of software quality*, 1981.

[22] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE Software*, vol. 33, no. 6, pp. 27–29, 2016.

[23] P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994. Oregon Workshop on Software Metrics.

[24] Y. Ma and H. Liang, "Research on task-oriented maintainability index allocation method of warship overall," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pp. 2950–2955, 2017.

[25] C. Gote, I. Scholtes, and F. Schweitzer, "git2net - mining time-stamped co-editing networks from large git repositories," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 433–444, 2019.

[26] F. Heseding, "Fabianhe/pyrepositoryminer: Efficient repository mining in python," May 2021.

[27] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 908–911, Association for Computing Machinery, 2018.

[28] G. Gousios and D. Spinellis, "Mining software engineering data from github," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, p. 501–502, IEEE Press, 2017.

[29] A. Bhatia, E. E. Eghan, M. Grichi, W. G. Cavanagh, Z. M. J. Jiang, and B. Adams, "Towards a change taxonomy for machine learning pipelines: Empirical study of ml pipelines and forks related to academic publications," *Empirical Softw. Engg.*, vol. 28, apr 2023.

[30] K. Muthukumaran, A. Choudhary, and N. B. Murthy, "Mining github for novel change metrics to predict buggy files in software systems," in *2015 International Conference on Computational Intelligence and Networks*, pp. 15–20, 2015.

[31] C. Gote, I. Scholtes, and F. Schweitzer, "git2net - mining time-stamped co-editing networks from large git repositories," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, may 2019.

[32] GitHub, "Github activity data," 2018.

[33] L. N. Jr., "Github app installation token and authenticating as a github app," 2023.

[34] A. MacDonald, "test-repo," 2023.

[35] Python, "Pythongraphlibraries - python wiki."

[36] Y. Wang, B. Xu, M. Kwak, and X. Zeng, "A simple training strategy for graph autoencoder," in *Proceedings of the 2020 12th International Conference on Machine Learning and Computing*, ICMLC '20, (New York, NY, USA), p. 341–345, Association for Computing Machinery, 2020.

[37] T. N. Kipf and M. Welling, "Variational graph auto-encoders," 2016.

[38] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[40] GitHub, "Github rest api documentation," 2022.