Allegheny College
Computer And Information Science Department

Senior Thesis

EvaluRate: A comprehensive analytical tool evaluating the effective maintenance of GitHub repositories.

by

Orion Grieco

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Gregory Kapfhammer**
Co-Supervisor: **Janyl Jumadinova**

Abstract

In respect to vetting software, or verifying its long-term integrity, there is no one-size-fits-all solution, let alone a large-scale application to satisfy such a need. In this paper, the prospects of developing a comprehensive analytical tool to evaluate the maintenance of software repositories are explored. EvaluRate serves as a means of effectively assessing a variety of metrics to determine the overall quality of the maintenance given to a software repository in GitHub. The operations of EvaluRate span across a grand multitude of repositories, and the results are compiled into a single report, that is additionally publicly available for users to access.

Table of contents

1 Introduction **6**
- 1.1 Background . . . 6
- 1.2 Problem Statement . . . 6
- 1.3 Objectives . . . 7
- 1.4 Significance of the Software . . . 7
- 1.5 Scope and Limitations . . . 7
- 1.6 Ethical Implications . . . 8
  - 1.6.1 Privacy and Data Security . . . 8
  - 1.6.2 Data Collection Issues . . . 8
  - 1.6.3 Bias and Fairness . . . 8
  - 1.6.4 Second or Third-Party Risk . . . 8

2 Related Works **10**
- 2.1 Existing Evaluation Tools and Methods . . . 10
- 2.2 Evaluation Metrics . . . 10
  - 2.2.1 PHANTOM . . . 10
  - 2.2.2 reaper . . . 11
- 2.3 Challenges in Evaluation of Repository Maintenance . . . 12
  - 2.3.1 PHANTOM . . . 12
  - 2.3.2 reaper . . . 12
- 2.4 Gaps in Existing Research . . . 13

3 Methodology **16**
- 3.1 Research Design . . . 16
- 3.2 Data Source(s) and Selection . . . 17
  - 3.2.1 Data Source(s) . . . 17
  - 3.2.2 Source Selection Rationale . . . 17
- 3.3 Data Collection Methods . . . 18
  - 3.3.1 Procedure(s) for Data Collection . . . 18
  - 3.3.2 Repository Selection . . . 18
  - 3.3.3 Sampling Technique(s) . . . 19
  - 3.3.4 Metric Information . . . 19
- 3.4 Data Analysis Methods . . . 24
  - 3.4.1 Description of Data Analysis Techniques . . . 24
  - 3.4.2 Justification of Data Analysis Techniques . . . 24
- 3.5 Control and Randomization . . . 25

4 Experimentation **26**
- 4.1 Basis of Experimentation . . . 26
- 4.2 Experimental Design . . . 26
- 4.3 Determinants of Success . . . 26

5 Results **27**
- 5.1 Data Overview and Presentation . . . 27
  - 5.1.1 Evaluation of Results Produced by Neural Network Assessment Technique . . . 27
  - 5.1.2 Evaluation of Results Produced by K-Folds Cross Validation Technique 28

5.1.3 Statistical Analysis Experimental Results 30
5.2 Key Findings 30
5.3 Interpretation of Results 30
5.3.1 Comparisons with Existing Research 30
5.4 Implications of Findings 30

6 Conclusion **32**
6.1 Summary of Results 32
6.2 Future Work 32
6.3 Future Ethical Implications and Recommendations 33

List of Figures

List of Tables

1 Introduction

1.1 Background

When creating new programs or software, oftentimes developers or contributors will utilize dependencies to facilitate an easier process in developing code or software, and the functionality therein. Dependencies are usually proprietary programs or software made by developers, with the primary use-case being in open-source projects made by other developers. By using dependencies, developers can save time and effort in developing software, and can also ensure that their software is up-to-date and functional. In the case of open-source software, dependencies are usually made available through a repository, which is a collection of files and folders that contain the dependency's source code. These repositories are usually hosted on GitHub, and are often made available to the public.

GitHub itself is a highly collaborative platform that allows users to work with others to develop software, or seek help with troubleshooting bugs in their code. Users on GitHub can also host their own code repositories, for both public and private access. In the case of a user creating a public repository, others can fork the repository, which creates a local copy stored on the user's device. However, if a repository is private, only the user who created the repository can access it, and access permission can be given out to other users at the discretion of the user who created the repository. Users can also track the development of software developed by themselves or others via Git, a version control software supported by and integrated into GitHub; even large companies utilize GitHub for their internal projects and product development [9].

Well-maintained repositories are crucial in fortifying the reliability, usability, and security of software projects. This consequently entices engagement between the developers of the repository dependencies, and the developers of software projects utilizing them. This further can incite other developers and contributors to maintain these repositories, especially if they are utilized by critically important software services or digital infrastructure, in effect reducing associated risks with outdated or inadequate code, as well as cost overhead. However, beyond the intricacies of analyzing a wide degree of software, and the benefits of ones well-maintained, the current methods of evaluating the maintenance of repositories are limited in scope and application. However, despite the advantages of well-maintained repositories, current methods for evaluating the maintenance of repositories are limited in scope and application [10]. This gap underscores the need for more comprehensive tools and methodologies to assess repository maintenance effectively.

1.2 Problem Statement

With the significant importance placed on maintaining code repositories, the current means for evaluating them are limited. The majority of these tools are either proprietary or require users to have a GitHub account associated with the repository to access them. Many existing tools also operate in a singular manner, meaning only one repository at a time can be evaluated. These tools often highlight errors within individual files rather than assessing the overall integrity of the repository [**?**].

Furthermore, the methods that exist for evaluating repositories can be biased or inconsistent. This is further amplified by the vast quantity of repositories that exist on GitHub that may require different approaches of evaluation. Such issues that can arise from the lack of suitable evaluation measures are compatibility conflicts, compromised project integrity, and reliance upon outdated packages or libraries [2].

For example, compatibility conflicts occur when different parts of a software project operate as intended together due to dependencies running on different versions of the same library or package. The compromised integrity of a project means that the overall quality and reliability of the software can be at risk if some parts of the project are not well-maintained. Reliance on outdated packages or libraries can introduce security vulnerabilities and functional issues.In extreme cases, it may cause a program that utilizes a faulty dependency to become completely inert or defective, resulting in a costly re-structuring of the software and subsequent code [3].

Hence, there is a clear need for more comprehensive and reliable tools to assess the integrity and maintenance of code repositories effectively. EvaluRate is offered as n effective means to evaluate multiple repositories simultaneously, provide an overall assessment of repository health, and reduce the biases and inconsistencies of current methods.

1.3 Objectives

The primary objective(s) of this research is to create a tool that evaluates GitHub repositories to ultimately determine their maintenance quality. The goals of EvaluRate are to provide a comprehensive and effective means of evaluating the maintenance of a repository, by utilizing a wide range of metrics such as the size of a code repository, the amount of contributors, and the presence of a repository license or test code files to calculate maintenance quality. By way of thorough assessment of pertinent repository attributes, EvaluRate will produce consistently objective analyses, aiding developers and contributors to guarantee the security and well-being of their own repositories and projects, as well as software they may rely upon as dependencies in the development of their own software.

1.4 Significance of the Software

The creation of an automated evaluation tool for codebases serves to greatly benefit developers and contributors, open and closed-source alike. Through ensuring that a repository is being effectively maintained, EvaluRate can improve the quality and security of open-source projects, or those that utilize open-source assets. This would in turn further establish a degree of confidence and collaboration amongst developers and contributors, and allow for users to make well-informed decisions when choosing dependencies for their projects. Beyond contributing to the practicality of software maintenance, EvaluRate also bolsters the comprehension surrounding the maintenance and evaluation of code repositories.

1.5 Scope and Limitations

This research is based on the development of a tool situated for evaluating software or code repositories based on maintenance quality. The empirical scope of the project includes the identification and utilization of relevant metrics to evaluate a repository's quality of maintenance. Further included is the testing and validation of the tool across a wide range of repositories to ensure effectiveness and consistency, as well as machine learning models to further support the tool's validity. The functional scope of this tool is limited to the GitHub platform, as it solely utilizes publicly hosted repositories on GitHub, in part through the GitHub REST API. The software produced in this research does not extend to other version control software, such as GitLab or BitBucket, and is limited in its effective scope of application. Additionally, while the tool aims to provide a fair and comprehensive means

of analysis and evaluation, it may not account for every possible factor that can have an effect on the integrity of a repository.

1.6 Ethical Implications

1.6.1 Privacy and Data Security

Since EvaluRate will be accessing GitHub repositories and further analyzing them, data that might be considered as private or sensitive may be processed as a result. In such instances, it is important to ensure that all user data processed is handled in a responsible and careful manner to ensure the security of the data. This also affects repositories, as they may also contain sensitive information. Furthermore, since these repositories will contain intellectual property of other individuals, it is important to respect the privacy of the developers and organizations involved, and ensure that their work is not misrepresented.

1.6.2 Data Collection Issues

EvaluRate will only have access to public repositories on GitHub, which may provide an inadequate reflection on a developer's or an organization's overall maintenance quality, especially as many repositories are kept private for security or personal reasons. In some cases, documentation or issue tracking of certain software may be stored by private means on GitHub or other version control sites, inhibiting the accuracy of the software's evaluation.

1.6.3 Bias and Fairness

The algorithms employed by EvaluRate, as well as the metrics used may have the potential of introducing bias, which could lead to skewed or inaccurate assessments. Maintaining an objective means by which repositories will be evaluated is critical to ensuring fairness in assessment, as well as ensuring that all code will be reviewed in an objective and equitable manner. EvaluRate also uses one primary source for obtaining relevant information on GitHub repositories: the GitHub REST API. While this is a highly reliable source for the context of operations of EvaluRate, there is the possibility of other programs or APIs that are capable of providing valuable information on GitHub repositories that operate outside of the direct scope of GitHub itself.

1.6.4 Second or Third-Party Risk

Since EvaluRate utilizes the GitHub REST API to a high degree for accessing critical information needed to evaluate repositories, there is a high risk associated with a reliance on a single source of information. In the case of the GitHub REST API changing drastically in structure, or shutting down, an alternate means of accessing repository data would need to be established to ensure that EvaluRate can operate uninhibited. Furthermore, EvaluRate itself relies on third-party dependencies to perform its functions. These dependencies may also be subject to change, which could lead to unintended consequences that may negatively impact the performance of EvaluRate, and in turn the reliability of its operations and accuracy. There is also the possibility that EvaluRate may provide false positives or false negatives of a repository's maintenance quality, which can misinform users on what dependencies would ultimately be of use to them. This could potentially lead to certain dependencies being overlooked, or discontinued in usage entirely, resulting in the undermining of certain users

or their software. By ensuring that EvaluRate operates with a high degree of accuracy, it will be able to mitigate these risks. There also lies the potential for third-parties to become over-reliant on this software without considering other alternatives. In this situation, it is important to note that the use of other tools that can appropriately evaluate GitHub repositories is recommended for a more accurate assessment on the part of the user.

2 Related Works

2.1 Existing Evaluation Tools and Methods

In respect to tools that perform analysis on code repositories on GitHub, there are several that currently exist. PHANTOM is a tool created to handle mass-evaluation of software projects in a resource-cautious manner. It ultimately determines if a GitHub repository is an engineered project by way of time-series clustering. Another of these tools is called reaper, created by students at RIT. The tool itself functions by ultimately marking a repository with a 1 or a 0 to signify if a GitHub repository contains a engineered software project.

2.2 Evaluation Metrics

2.2.1 PHANTOM

PHANTOM is a tool that utilizes Git logs to evaluate repositories. The tool compiles k-means time-series clusters from Git logs, and does so through unsupervised machine learning to effectively carry out its tasks of analysis [10]. A benefit of utilizing Git logs as opposed to other sources is that all data obtained from Git logs can be cross-analyzed, regardless of the data came from open or close-source repositories. PHANTOM records five different attributes of repositories from Git logs [10]. These attributes are *commit frequency*, *integration frequency*,*integrator frequency*, *committer frequency*, and *merge frequency*[10]. The *commit frequency* metric is a count of the weekly number of commits. *Integration frequency* consists of the number of integrations per week. An integration consists of extending a program's functionality, potentially by way of connecting said program to other software tools or services. *Integrator frequency* Is the weekly number of unique contributors that made integrations to a repository [10]. *Committer frequency* is the weekly number of unique contributors that have made a commit to a repository. The last metric, *merge frequency*, consists of the weekly number of merges [10].

PHANTOM further operates by adhering to six primary rules, the first of which being all attributes that are being assessed must be obtainable from a Git log. The project itself consisted of testing over 1.8 million repositories, meaning that data and time overheads were points of concern for ensuring effective operation. The tool was also designed to be resource-efficient, so it was necessary to ensure that the tool could be run on a single machine [10]. The second rule consisted of all k-means time-series must be accurately consolidated by feature vectors. In this case, a feature vector is a set of features that is going to be used to construct the time-series clusters used for analysis. The third rule is that a propose method for training an unsupervised learning model must identify relevant k-means time-series clusters based on filtering those that contain an engineered software project [10].

By ensuring that testing the effectiveness of parameters for training unsupervised learning models produce the intended results, it can be assured that PHANTOM can accurately determine if a repository contains an engineered software project. The fourth rule is that PHANTOM is required to handle its processes on a large-scale operation, especially if running on single machine [10]. Given that is carrying out analyses of over 1.8 million repositories, it would need to be capable of handling a large load of input and raw data from Git logs, and be able to accurately process and produce consistent and reliable output. The fifth rule of PHANTOM is that the results produced by its unsupervised model(s) must be relatively accurate in comparison to supervised models [10]. PHANTOM was created to be an unsupervised model due to the massive data overhead the project was faced with; this

left PHANTOM with a risk of providing results with lower accuracy as now it "[utilizes] unsupervised methods, such as clustering" [10]. This is done in order to reduce the time spent on data processing and analysis by PHANTOM, but at the cost of overall accuracy of the output produced.

The final rule PHANTOM adheres to is the unsupervised learning method employed must be capable of filtering engineered projects from different types of datasets [10]. For example, datasets coming from an open-source background, and another from an industrial point of origin would have different contexts and must be evaluated equitably and analyzed based on the same parameters [10]. The PHANTOM project also accounted for false positives and negatives. To combat this, the precision, recall, F-Measure, and Matthews Correlation Coefficient were all collected by each model employed by PHANTOM, and subsequently compared for accuracy. The study involving PHANTOM also compared these scores to that when reaper was used, and both software programs were comparatively tested to determine which one ultimately was the most accurate.

In the case of evaluating PHANTOM against reaper, there were five categories of sample repositories constructed for testing purposes. The first category consisted of repositories that both software programs had classified as containing an engineered software project. The second of which contained repositories marked as false by both tools. The third category consisted of projects that were considered not engineered projects by PHANTOM, but engineered by reaper. The fourth category included projects that were considered engineered by PHANTOM, but not engineered by reaper. The fifth category consisted of a mixed classification of projects, with slight overlap where there was partial agreement between the prediction models of reaper and PHANTOM [10].

2.2.2 reaper

The program reaper evaluates a given GitHub repository by seven different metrics. In addition to these seven metrics, repository size is also evaluated. The repository star count and the primary language are also recorded by reaper, but are not utilized in the calculation of the engineered repository score. The first metric by which reaper evaluates a repository is *Community*. This attribute keeps track of the unique amount of users that contribute to a repository. The second attribute of reaper, *Continuous Integration*, serves to determine if a continuous integration system is being used to support the stability of a given source code repository. The third attribute *Documentation* records the ratio of "comment lines of code… to the number of non-blank lines of source code" [8]. This metric is used to keep track of maintainability of a repository, as source code comments can serve to provide insight on the purpose of code, or its respective functionalities.The fourth attribute, *History*, records the average commits pushed per month as an indicator of "a bug fix, feature addition, preventative maintenance, [or] vulnerability resolution…" [8]. The fifth attribute, *Issues* records the average number of issues events created per month for a given repository. This feature is intended to quantify the amount of issues as a frequency to keep track of project management.

The *License* attribute of reaper accounts for the presence of a license in a repository via the GitHub License API. The usage of the API in this case identifies the existence of "popular open source licenses by analyzing files such as LICENSE and COPYING…" [8]. The program determines if the repository contains a license through parsing through each file within a given repository, and checks all lines for matches to excerpts of the 12 most popular open source license files used on GitHub. The final attribute in this program, *Unit*

Testing, calculates a ratio of source lines of code in present test files to the number of lines of code in all source code files. This operation is performed through pattern searching and identification, where the program searches for statements within source code files sch as `import unittest` or `from unittest import TestCase` [8]. The tool `grep` was used in this case to compile a list of files that shared patterns that indicate a test class or code file, such as the two aforementioned. `cloc` was also used to calculate the ration between the respective types of lines of code. In the case a repository utilizes more than a single unit test framework, reaper counts the total number of test files then subsequently calculates the test code lines to total source code lines ratio.

2.3 Challenges in Evaluation of Repository Maintenance

2.3.1 PHANTOM

While an unsupervised model does save costs in respect to time and memory overheads, it will ultimately cost more in terms of accuracy and reliability as a result [7]. This effect is also compounded by using a k-means clustering algorithm, which can suffer from reduced accuracy when computation time is decreased [7]. Furthermore, most accurate models of reaper, including the most accurate PHANTOM model, estimated the amount of engineered projects to make up roughly 19-24% of the repositories evaluated [10]. The remaining PHANTOM models estimated this amount to be between 38-55% [10]. This is a significant difference, and it is likely that the PHANTOM models are overfitting to the data, potentially as a result of a compounded reduction in accuracy due to relying on unsupervised machine learning models in tandem with a k-means cluster model. The only instances where PHANTOM matched reaper's accuracy was when both programs had unanimous predictions [10]. In these cases, there was a lesser degree of error between reaper and PHANTOM, and PHANTOM had higher accuracy scores.

Furthermore, while reaper used seven metrics in its analysis of repositories [8], only one metric was assessed by PHANTOM [10]. This leaves gaps in the evaluations of repositories, as five attributes were extrapolated from Git logs, leaving other sources unutilized. By relying solely on Git logs for cross-analysis capabilities, there exists a potential for associative bias, as all the data originates from a singular source. While it may seem efficient to use a single source for data scraping or collection, this approach can be problematic when considering bias and accuracy. Using one source may prevent the acquisition of relevant information through alternative means.

Additionally, attempting to classify engineered software projects using different datasets via a singular data collection method may lead to discrepancies between the datasets in terms of the metrics used. Some datasets might lack critical attributes or data values essential for the proper functioning of PHANTOM, or contain ones that differ from pre-recognized parameters of the unsupervised model(s) [10]. The assumption that using Git logs alone can adequately process a wide variety of datasets is risky. This method may not capture the full scope of necessary information, potentially affecting the accuracy and reliability of PHANTOM's evaluations.

2.3.2 reaper

In respect to the *Continuous Integration* attribute of the reaper program, it can only account for specific programs. Only eight different continuous integration programs are tracked, with those being Travis CI, Hound, Appveyor, Shippable, MagnumCI, Solano, CircleCI,

and Wrecker [8]. Beyond only tracking these eight services, the tool is unable to track stateful continuous integration services, as it looks for a configuration file within the source code of a repository [8]. By this approach to detecting continuous integration services, only stateless services are able to be tracked. Reaper also struggles to fully assess comments and associated documentation with the *History* metric, as it is unable to read external documents outside of the scope of source code. In this case, a potential solution would be to employ a NLP model that is trained on such data in order to determine the intent and context of external documentation within the repository.

The *Issues* attribute of reaper is also susceptible to deception, as listed issues within GitHub Issues may not pertain to the repository itself, but may be a conversation between users commenting on an error or bug [8]. An example of this is a "(developer) could create user stories as GitHub issues and label them as User Story" [8]. Such instances can artificially inflate the issue frequency metric as established by reaper or other programs, and can alter the data recorded on associated or dependent metrics. The *License* metric of reaper was flawed, as it could trigger a false positive if any file within the repository contained textual excerpts of the license file being searched for.

By parsing through the repository as a whole, there is a large time overhead, especially for large or commercial and industrial use software products, especially as the designed algorithm is reading through each individual file for a textual match with an excerpt of 12 possibly present license files. In a case similar to that of the License metric, the *Unit Testing* metric is also susceptible to false positives. If a repository does not contain a unit testing framework that is identified by reaper, the program will default to classifying files in directories labeled "*test*, *tests*, or *spec*" [8] as test files. Though this approach may seem effective, there is potential for misclassification or misidentification of files, providing a false positive, or even overlook test files not contained within a directory with a name pre-defined as a test directory by reaper.

2.4 Gaps in Existing Research

In current research, there appears to be gaps in what is universally considered to be objective metrics to evaluate software. While there are several that can be used to determine the quality of a program at a given state or time-frame, there does not appear to be any specifically tailored for maintenance. Furthermore, methods currently employed by existing tools have the potential to overlook certain aspects of a software product, such as systems being locked to search pre-determined directory locations that are presumed to exist [8].

Additionally, the models themselves or attributes thereof are not the most effective in establishing accurate calculations. Take for example, the unsupervised learning models employed by PHANTOM [10]. While using this approach is more cost-effective in respect to supervised or reinforced models, the lack of accuracy is pronounced, with the most accurate models holding an accuracy range between 0.6-0.7 in respect to about 59% of the repositories tested [10].

In respect to these flaws within PHANTOM's structure, both inhibit creating an effective means of establishing a system to detect engineered software projects, the aspect of an unsupervised learning model may be one of the largest pitfalls to the validity of the software. In this case, the software models employed by PHANTOM will make deterministic choices that effect the eventual prediction of a repository being deemed to be an engineered software project are at risk of making rushed, inaccurate decisions. While it is argued that "methods that base filtering on supervised machine-learning algorithms require manual labelling of

the training data" [10] are time-cost intensive, it will serve to provide the proper structure needed to develop an accurate and reliable model.

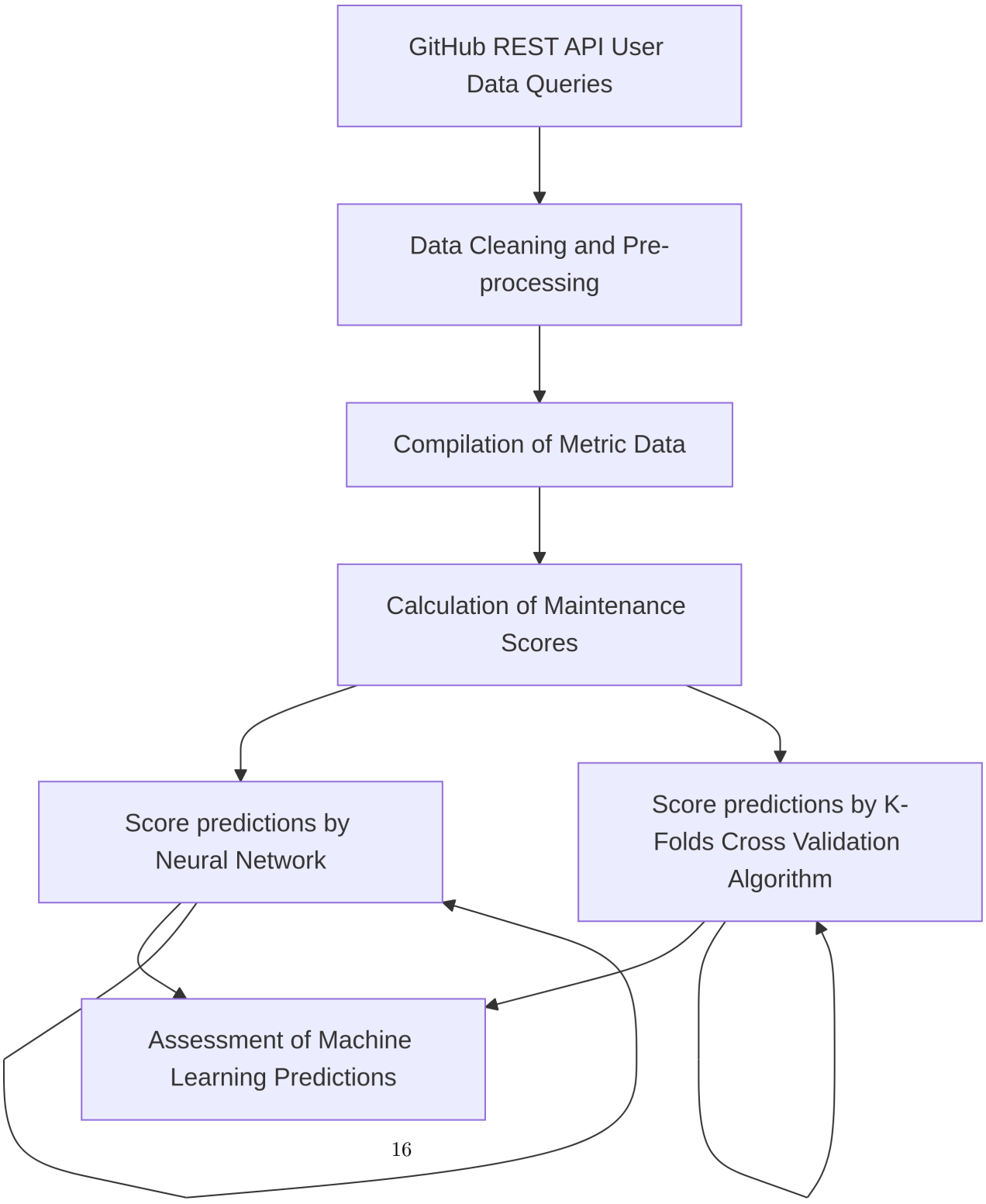
The further lack of reliable accuracy in PHANTOM's results underscores the importance of employing increasingly capable or versatile means of evaluating codebases or repositories. With only 59% of repositories having a reported accuracy ranging between 60-70% at peak levels [10]. There are ways to improve the performance of training supervised learning models, such as dataset proportioning, which compresses a portion of the dataset used for training a machine learning model while also minimizing accuracy losses [6]. Such a method utilizes compression to reduce the time overhead of training a model, while preserving the dataset's integrity as to allow a trained model to learn enough features to maintain a respectable degree of accuracy in comparison to being trained on a non-compressed dataset [6]. The datasets used in this example utilizing dataset proportioning consisted of between 25000 to 2.4 million data points [6]. The range of error between these datasets never rose above 2.5%, staying within a range of 0-2.25%, with the error rate of the largest dataset hovering around 1%, and remained within this range as the data proportions tested were increased [6].

As for reaper, the *Continuous Integration* metric is not sustainable in long-term operations due to its exclusivity. Other studies of this metric alone have covered several tools that facilitate practices of continuous integration through a variety of means [11], implying that more than tracking a stateless tools would be needed to adequately assess the conditions of applicable attributes of this metric. Furthermore, continuous integration associated attributes can consist of reducing build and test time constraints [11], resolving scalability issues in respect to security and deployment [11], improving performance and reliability [11], and even detecting flaws and faults within the CI itself [11]. These factors, especially the latter attribute, would be critical in evaluating the effectiveness of CI tools themselves to better determine the validity of this characteristic, especially in the extremely limited evaluation pool of 8 tools as is the case with PHANTOM [10].

Additionally, the current primary data source for reaper is GHTorrent, which has since become defunct, and no longer serves as a database. The GHTorrent domain currently is used as a redirect page to an Indonesian gambling site. The high dependence placed upon GHTorrent by reaper jeopardizes the integrity of the project as a whole, especially as GHTorrent is now a non-existent, unstable non-primary data source. GHTorrent operated by scraping data mainly from GitHub for its purpose as an online database, though it does not match up to the scale of GitHub, BitBucket, or Codeberg. By effect, GHTorrent may run the risk of not being as reliable, as it currently appears to have lost sufficient funding to maintain operations. There also is the potential for data obtained and held by GHTorrent to not be as comprehensive in structure or as versatile in comparison the aforementioned platforms. By engaging with a larger and consistently reliable platform such as GitHub, there is a higher chance of finding appropriate data, as well as having long-term access to the GitHub REST API and other associated resources to ensure that EvaluRate can adequately operate, and gain some degree of future-proofing through this notion.

3 Methodology

3.1 Research Design



The intention of this project is to assess a valid means to determine the quality of maintenance befallen to a code repository hosted on GitHub. This process consists of four independent but conjoined operations. The first of which consists of data scraping, specifically scraping metadata from the GitHub REST API. The second part of this program is to then process the data obtained from the REST API, and break it down into data respective to the metrics that will be evaluated by the software. The subsequent portion of this program then calculates a given maintenance score, which is a binary value of a 1 or 0, to signify a well-maintained or an otherwise inadequately maintained repository, respectively.

The fourth part of this software is then a score validation reinforcement of the initial calculations, using two machine learning approaches. The first of which is a neural network, which will be operating as a supervised machine learning model, as it is processing the data in respect to pre-calculated scores. the neural network utilizes a percentage of correct calculations relative to what was produced by the prior program as its accuracy estimation(s) of the model's predictions are compared to that of the internal calculations performed by EvaluRate. The second machine learning method utilized is a K-folds cross validation model, which will compare true and false predictions, of those consisting of one of the four following categories: a true positive, a false positive, a true negative, and a false negative. The ratios of each respective true/false ratio for positive and negative predictions are used in determining the accuracy of the K-Folds cross validation model.

3.2 Data Source(s) and Selection

3.2.1 Data Source(s)

The data used in this research comes directly from the GitHub REST API. GitHub itself has over 100 million unique users, and over 420 million repositories. An estimated minimum 28 million of these repositories are public, around 6.7%. Public repositories allow for unrestricted open access, meaning that anyone can see and access the repositories and the code within. A tool called PyDriller as also used in this experiment, and it produced data as extrapolated values from the data obtained from GitHub REST API queries. PyDriller itself is not a source of data, but a supplemental tool for data extrapolation from data procured from the GitHub REST API. Its mention is included within this section to help explain where some metrics are obtained that do not exist within the GitHub REST API.

3.2.2 Source Selection Rationale

GitHub contains an incredibly vast amount of data, even with 6.7% of the repositories being publicly accessible, the raw amount of roughly 28 million repositories is more than sufficient to develop EvaluRate effectively, or any similar tool. The GitHub REST API itself is what grants users and developers direct access to the raw data behind the scenes, which contains a swath of comprehensive data on repositories and users alike. Data attributes such as commit count, contributor count, as well as time series data such as tracking the amount of commits within specified time ranges can further provide insight into the conditions within code repositories.

The GitHub REST API provides live data, ensuring that the information received is the most accurate and up to date. The output itself is also paginated into a collection of dictionaries in JSON format, making way for easy parsing and processing of the data collected from the API. GitHub REST API also has a very large documentation, allowing for users to easily understand and work with the API. There also is a extremely active

community on GitHub, one which its members regularly and engages with one another at incredibly high frequency, whether it be to solve bugs or errors in GitHub itself, or resolve issues in an open-source project. This creates a sort of safety net, as it establishes the notion of an active userbase that could provide insight into effective practices with certain language, packages, or the services offer by GitHub, such as the REST API.

GitHub is also a largely trusted platform, with millions of users and hundreds of millions of respective repositories. The software is also owned and maintained by Microsoft, ensuring a high degree of stability and security on the platform, guaranteeing that the data obtained is reliable and credible. Furthermore,The GitHub API is modeled on the REST format, or Restful State Transfer. The REST API format optimizes user-server interactions, and supports caching. Caching itself can remove interactions between the serve and user entirely, as the data is stored on the client device requesting data. PyDriller itself is a supplementary tool used for the extrapolation of data values from the data contained within the GitHub REST API queries, and focuses on producing data of proprietary metrics that otherwise to not exist within the parameters of the GitHub REST API.

3.3 Data Collection Methods

3.3.1 Procedure(s) for Data Collection

The primary method employed to acquire the data is through sending API requests of code repositories linked to usernames or user identification numbers that are fed as input. EvaluRate utilizes a function that accrues a list of randomly generated GitHub user identification numbers, where the amount of which is user-specified. The user ID numbers are each sent through requests to the GitHub REST API to determine if they are in fact valid, as there is not a publicly accessible user directory or user list to obtain such ID numbers otherwise. These user ID numbers are then compiled into a JSON file for further processing in the collection of repositories associated with the compiled user ID numbers. This is the main method used in the experiment, and the source of all data that is used in training and testing the machine learning models employed within. Furthermore, EvaluRate employs checks of all user ID numbers generated for testing. The first check employed searches for a valid ID, where the ID number is checked within the GitHub REST API to verify that a http `200 OK` code is returned, meaning that the user ID is found within the system, and the request made to the API server was successful. The second check then verifies that there are repositories associated with the user, specifically that the user has created repositories that the user currently owns. If either of these checks happen to fail, the user ID is rejected, and ultimately discarded from the compiled list of valid user IDs.

There is also a secondary manner which was used to obtain repositories, by calling the command `gh api users` in the Command Prompt or PowerShell terminal. Using this approach provides output in the form of a serialized list containing dictionaries. Each dictionary contained a unique username and their associated metadata. However, this produces a portion of the presumable entire list, and does not provide the entirety of the true list of GitHub users, as there are only 60 dictionaries containing user metadata returned as output from this method.

3.3.2 Repository Selection

Once the raw user data is obtained, the relevant information such as usernames or user ID numbers are sent through a series of requests to the GitHub REST API. The requests made

to the API attempt to obtain the metadata attached to each user, such as the user's own repositories. In the approach of using a link to obtain repository information associated with a user, only the metadata of the specified repository will be returned. However, if a file were used as input, in this case a compilation of user metadata, the file will be parsed for each user's respective repository metadata, returning the sum whole of all repositories made and owned by the users contained within the files. The repositories returned are only those that are owned by the user, however there exists the option to view repositories that the user is a contributor of as well. the metadata of each repository consists of a multitude of metrics, those of which are made up of numerical, categorical and textual values, and API reference links.

Given that the users are chosen at random, the repositories of each user are also randomly chosen. Similar to how the amount of user IDs generated is user-specified, this principle also befalls the process of repository selection. However, the repositories are checked for specific attributes, such as the presence of a license, presence of issues, or valid language tags. Given that a repository fails even one of these checks, it is then rejected for evaluation and will no longer be used within the dataset.

3.3.3 Sampling Technique(s)

The data from the repositories themselves were not used in their entirety. Specific metrics were chosen from the whole thereof, as there exist values that are not conducive to the effective assessment of repository maintenance. The chosen values of the repositories are the repository size, license, contributors, commit frequency, commit frequency disparity, primary contributor commits, language count, language makeup, primary language, and issue resolution frequency. There are also filters that will determine if a repository is ultimately included or discarded from the datasets used as a whole. For example, in the case a repository does not contain metadata regarding repository languages or a license, in either or both cases the repository becomes discarded from the dataset.

3.3.4 Metric Information

3.3.4.1 Repository size The repository size in the GitHub REST API is calculated in bytes, and is synonymous with the sum of bytes of code per language used in a repository. This ended up being a secondary means of ensuring the validity of the repository, as its respective size and bytes of code are compared before analysis to make sure that they match before computation. This metric is intended to serve as a record of scale of a repository, given that as a repository scales up in size and becomes more complex or lengthy, it will require more resources and human effort to maintain it.

$$RS = \sum_{i=1}^{n} S_i$$

RS : Total Repository Size (in bytes)
S_i : Size of the i-th file (in bytes)
n : Total number of files in the repository

3.3.4.2 License This metric is quite simple, as it only calculates the presence of a license. The main focus of this project is to evaluate software or code repositories on GitHub, with the underlying assumption these have the potential to be used as dependencies in future projects by other GitHub users or developers. The lack of a license causes difficulties for the reproducibility of code and software stored within repositories, as well as poses potential legal issues from usage of said code or software. By ensuring that a repository does in fact have a license attached, it provides security and safety to the repository, as the legal caveats and permissions given to users in respect to reproduction, alterations, and other forms of utilizing the repository are given a legal outline to follow.

$$L = \begin{cases} 1 & \text{if a license is detected in the repository} \\ 0 & \text{otherwise} \end{cases}$$

L : License Detection Result (1 = True, 0 = False)

3.3.4.3 Contributor Count The attribute of *Contributor Count* within this experiment is to serve as a measure of available effort, as work within a repository can be broken up between contributors. As projects scale up in size and scope, there becomes a greater need of larger degree of capability or mobility within a project. Including new developers or contributors, especially those who have increasingly variable skills, can become paramount to the survival and support of the intended functionality of software, in effect ensuring that errors or bugs can be resolved swiftly. Having more contributors also supplements the base capacity of a development team, as there exists additional raw developmental power, or a surplus thereof, that can be relegated to handle proprietary tasks based on a developer's skillset, or what is of need for the repository and the project(s) within.

$$CC = \sum_{C_i=1}^{n} C_i$$

CC : Total Contributor Count
C_i : Contributions made by the i-th contributor
n : Total number of contributors

3.3.4.4 Commit Frequency *Commit Frequency* is used to gauge the general activity of a repository, as is comparable to a raw measure of the general productivity within a repository. This way, the effort produced by the developers or contributors maintaining a code repository on GitHub can be estimated. This metric is calculated as a ratio respective of the commits between the initial creation of the repository, until the last commit made in the repository. The ratio used in this experiment is an average of the number of commits made per day.

$$CF = \frac{C_T}{T}$$

CF : Commit Frequency (average commits per day)

C_T : Total number of commits in the repository

T : Lifetime of the repository (time between first and last commits, in days)

3.3.4.5 Commit Frequency Disparity This metric is presumably the most objective means of assessment in the quality of maintenance befallen to a repository. *Commit Frequency Disparity* is a metric that measures the gap in time between the last commit made to a repository, to the current day, if such a gap exists. This metric is intended to be utilized to measure the absolute frequency of upkeep outside of the *Commit Frequency* metric, as this considers the prospect of a project being neglected for a prolonged period of time.

$$CFD = D_C - D_L$$

CFD : Commit Frequency Disparity (in days)

D_C : Current Date

D_L : Date of the Last Commit

3.3.4.6 Primary Contributor Commit(s) The intention of using of *Primary Contributor Commits* is to keep track of any dependency or hyper-dependency on a single, or otherwise primary contributor. This metric is obtained through the use of PyDriller, and is a proprietary data point that does not exist within the traditional GitHub REST API. This metric is produced by calculating the percentage of the total code produced by the contributor listed with the most commits and code produced, and deriving that from the sum of over code produced and commits by the other contributors within the repository. The code produced by the primary contributor is calculated as a percentage, while the commits makeup as a ratio of the total commits within a repository is calculated as a fraction or a decimal.

3.3.4.7 Language Count The number of languages can be used to gauge the relative complexity of a repository in terms of how many languages are used, and by extension the minimum lingual comprehension required as established by *Language Count*. By assessing the number of languages used in the creation and maintenance of a repository, the information can be used to aid in the calculation of the amount of effort required to ensure proper and adequate maintenance of a repository.

$$LC = \sum_{i=1}^{n} B_i$$

LC : Language Count

B_i : Bytes of Code in Language i

n : Total Number of Languages

3.3.4.8 Primary Language Makeup The *Primary Language Makeup* can provide insight into potential over-reliance issues, or at least a primary usage of a specific language. The information provided by this metric can also be used to estimate the significance of certain languages, such as if over 50% of a code repository is developed using Lua. The primary language is calculated by calculating the largest singular percentage of bytes of code produced using a language, and extracting that as a percentage of the sum total of bytes of code within a repository.

$$P(PL) = \frac{\text{PLBOC}}{\text{TBOC}} \times 100$$

PLBOC : Primary Language Bytes of Code
TBOC : Total Bytes of Code

3.3.4.9 Issue Turnover Ratio As software is being developed, is it an eventuality that developers will run into bugs within the code, or related aspects of the design process of a software program. Ensuring that proper care is taken to swiftly resolve errors can mitigate future occurrences of the bug, as well as save costly re-designs in the potential if a bug were left ignored and development were to continue. Software restructuring and bug fixing becomes exponentially more costly as the development process proceeds towards reaching the deployment and maintenance stages. In fact, in the coding stage alone, the cost of bug fixing during the coding stage is 7x higher than if they were resolved in the initial design or planning stages [4]. However, it is 30-100x costly in comparison if a bug or defect were to be fixed in the deployment and maintenance stage of software development [4].

The Issue Turnover Ratio (ITR) is a metric designed to evaluate the efficiency and effectiveness of issue resolution within a GitHub repository. It provides a quantitative measure of how well contributors are managing and resolving issues over time. By analyzing the ratio of closed issues to open issues, the ITR offers valuable insights into the repository's maintenance practices, responsiveness, and overall health.

The *Issue Turnover Ratio* serves as a valuable metric for evaluating the reliability of a repository. A high *Issue Turnover Ratio* reassures users that their reported issues are likely to be addressed promptly, or that a software dependency they are using is aptly maintained. This in turn makes the repository in question a more attractive choice for adoption by users. Conversely, a low *Issue Turnover Ratio* may discourage users from relying on the repository, as it suggests potential neglect or lack of support towards a given repository. This metric is also viable in assessing the maintenance quality of open-source projects, where contributors often rely on issue tracking systems to report bugs, request features, or document tasks. By analyzing the *Issue Turnover Ratio*, it enables contributors to identify potential bottlenecks in issue resolution and allocate resources more effectively.

However, there are factors that can impact the accuracy of the *Issue Turnover Ratio*. The size and scope of a repository can impact the ultimate calculation of this metric value. Larger repositories with more features or dependencies may naturally have more issues, which can affect the *Issue Turnover Ratio*. Additionally, the availability of contributors and resources can influence the repository's ability to resolve issues efficiently. The activity level of contributors, the complexity of issues, and the rate at which new issues are created. For example, a repository with a high rate of issue creation may struggle to maintain a high *Issue Turnover Ratio*, even if contributors are actively resolving issues. Similarly, complex

issues that require significant time and effort to resolve may temporarily lower the *Issue Turnover Ratio*.

Additionally, while the *Issue Turnover Ratio* can serve to be an important indicator of the measure of diligence for resolving issues, it does have limits that can potentially inhibit its accuracy. The severity of an issue, for example, is not examined by this metric. Working to resolve damage caused by a bug, or a fatal flaw in the code of a repository would presumably take far longer than a simple formatting or type conversion issue. *Issue Resolution Frequency* also does not take into account the aspect of time, in part due to the difficulty with creating standardized evaluation metrics for the vastly differing time-frames of repository development across GitHub.

Therein also lies the potential for repositories to have been made for simple purposes, possibly to the extent that no issues have been made within the lifetime of the repository. There also exists the occurrence of a significant disparity in the lifetime of the repository in respect to issues produced and closed, which could lead to this metric becoming skewed. The *Issue Resolution Frequency* metric provides a current window into the issue resolution status of a given repository, but does not capture time-based trends, project simplicity, or changes made over time, as it is a static metric.

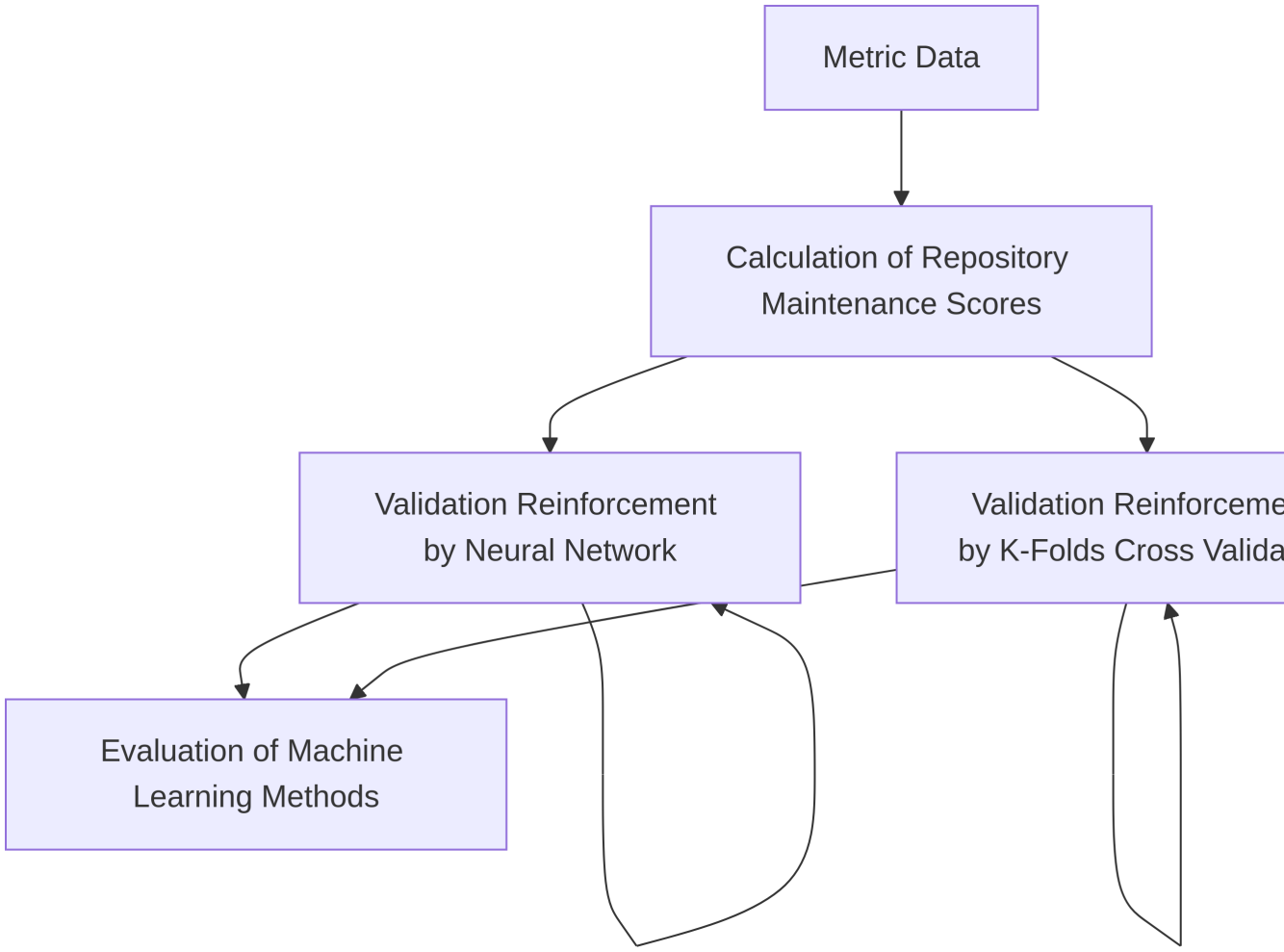
$$ITR = \frac{CI}{OI}$$

ITR : Issue Turnover Ratio

CI : Closed Issues

OI : Total Issues

3.4 Data Analysis Methods



3.4.1 Description of Data Analysis Techniques

Once the relevant information regarding the metrics for each repository is obtained, the maintenance scores are calculated using the data obtained from the metric data previously processed. Once the maintenance scores are calculated, they are ultimately used to train and test the neural network and K-Folds Cross Validation algorithm. Both of these approaches serves to appropriately assess and validate the results computed by the primary operations of EvaluRate.

3.4.2 Justification of Data Analysis Techniques

Both the neural network and K-Fold Cross Validation techniques are employed to validate the results produced by the internal calculations by EvaluRate. These algorithms also calculate maintenance scores, though these serve to be supplementary to the performance of

EvaluRate. The scores calculated by both the K-Folds Cross Validation algorithm, as well as the neural network are separate from each other, and act as a means to provide insight to the accuracy of the scores initially calculated by EvaluRate.

3.5 Control and Randomization

For the sake of experimental control, repositories that did not have a valid language tag were pruned from the dataset, as well as any others that did not have relevant information in regard to the specified metrics. Furthermore, as this tool is not operating on an objective measure of assessment, it is a probability assessment at base, and thus utilizes repositories that are assumed to be well-maintained. The definition of a well-maintained repository in this experiment consists of repositories that have been produced by large companies,or those produced in a manner where there are use-cases for users outside of the developers or contributors that created the repository and the software therein [8]. Repositories used in this manner are used as a baseline measure to provide reliable input to EvaluRate and the machine learning techniques employed within to assure a reasonable degree of operations by establishing a pseudo "ground-truth" of what a well maintained repository is. The context of a "ground-truth" in this case is a validating source that can and does objectively verify what a well-maintained repository is, including metrics of assessment in this regard, as well as the the raw attributes thereof.

GitHub code repositories made by well-established companies and developers with a wide user-base are included in the dataset as a foundational block. This is due to the assumed responsibility of both parties in this case to employ proper and consistent techniques and methods to produce professional-grade software that is believed to be objectively well-maintained. For further implemented control measures, the metrics are calculated the same for each repository, and no special conditions or exceptions are made during any point of the experimentation process. The repository scores are also calculated in the same manner for each repository, again in a standard format applied to each repository. The further bolster the accuracy of the machine learning techniques employed, they will be supplemented with a hold-off measure. This measure creates an evaluation architecture where each repository within the dataset will have a maintenance score predicted by the applicable machine learning approach after is has been trained on all other repositories within the dataset. This process is repeated multiple times for the sake of accuracy, and to reduce bias through accruing a larger sample size by this approach to stabilize the validity of the predictions made in this manner.

The repositories are taken at random however, given that the user IDs attached to them are randomly generated and validated individually. However, the repositories chosen to be the foundational block during model training is pseudo-random: There is a pool of companies and developers that will be chosen at random by EvaluRate. The number of foundational repositories used by each contributor or organization in this manner will not be limited: This further allows for an increased degree of randomness, as the variety in number of repositories for each respective contributor serves to affect the strength that the foundational repositories have on the ultimate maintenance calculations. This can be employed to further test the effect of certain large developers or organizations on EvaluRate's performance, or a combination thereof.

4 Experimentation

4.1 Basis of Experimentation

The core of this experiment resides within the quality assessment of GitHub repositories, and the further validation of the previous assessment of these repositories. The experiment is intended to validate the results produced by EvaluRate. This tool is designed to assess repository maintenance through a series of combined metric calculations, as well as through the use of machine learning techniques. The main purpose of the experiment outlined in this paper is to ultimately determine if EvaluRate can serve as an accurate means of assessing repository maintenance quality. The primary questions of experimentation are the following:

- [**RQ-1**] - How can accurately can EvaluRate calculate repository maintenance scores based on the selected metrics?
- [**RQ-2**] - What is the effectiveness of the neural network and K-Fold Cross-Validation models in validating the calculated maintenance scores?
- [**RQ-3**] - How do the selected metrics contribute to the overall evaluation of repository maintenance?

The experiment is also based upon the following hypotheses:

- [**H-1**] - Repositories with higher contributor counts, commit frequencies, and issue resolution rates will have higher maintenance scores.
- [**H-2**] - The accuracy attained by the neural network and K-Fold Cross-Validation models will achieve an accuracy within 80-85% in validating the calculated maintenance scores.

4.2 Experimental Design

The approaches used in this experiment will consist of a neural network using a hold-off approach for predicting repository maintenance scores, and a K-Folds Cross-Validation algorithm. Both of these approaches will also be using a confusion matrix for additional context to further understand the capabilities and pitfalls of either approach. EvaluRate will produce calculations of two different scores based on the same metrics of the same repositories. However, there also exists a normalized variant of the score that also uses a bell curve to account for potential abnormalities within the data and models used.

4.3 Determinants of Success

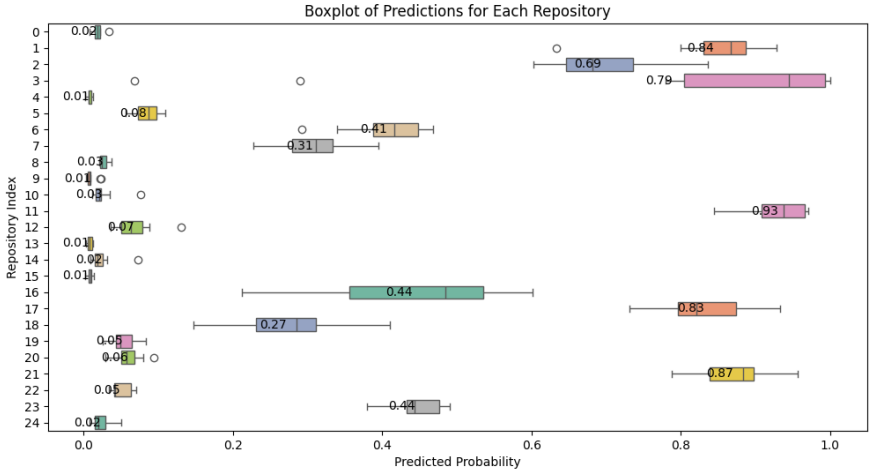
The base definition of EvaluRate's success lies in the accuracy in the machine learning algorithms employed to validate the calculated results. Given that either or both of the machine learning approaches used in the experiment, those being the neural network employing a hold-off approach, or the K-Folds Cross-Validation algorithm, hold a degree of accuracy within 80-85%, would be the primary determinant of success. Anything considered beyond that threshold would be considered as over-fitting, as the opposite case of under-fitting would be true in the case of the accuracy of either or both approaches dropping below 80%. In spite of the parameters of [**H-2**] outlining that both approaches will produce predictions with an accuracy of 80-85% each, this is an expected score for both models, and even one of the models performing under acceptable parameters and results would be considered a partial or otherwise qualified success.

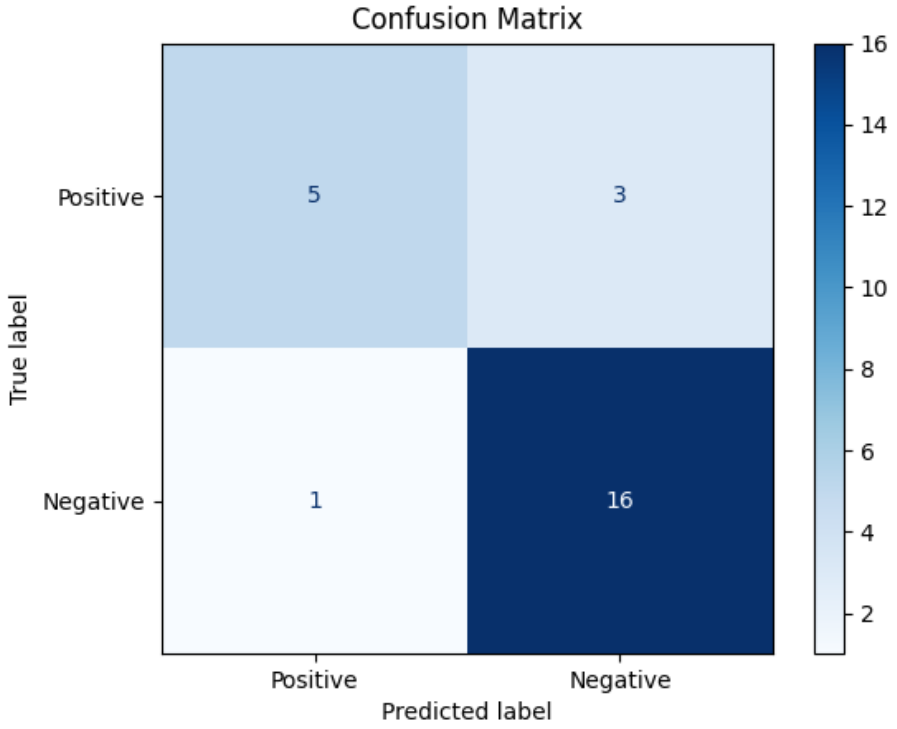
5 Results

5.1 Data Overview and Presentation

5.1.1 Evaluation of Results Produced by Neural Network Assessment Technique

The neural network employed in the experiment achieved an accuracy of 84% in its predictions, validating the maintenance scores calculated by EvaluRate. This result falls within the expected accuracy range of 80-85%, as outlined in the experimental hypotheses ([H-2]). The neural network was trained using a hold-off approach, where a majority of the dataset was used for training, and the remaining entry was reserved for testing. The model demonstrated strong predictive capabilities, with an accuracy of 84%, indicating that it effectively learned the relationships between the calculated metrics and the maintenance scores. This reported accuracy suggests that the neural network was able to generalize well to unseen data, potentially avoiding issues of overfitting or underfitting.

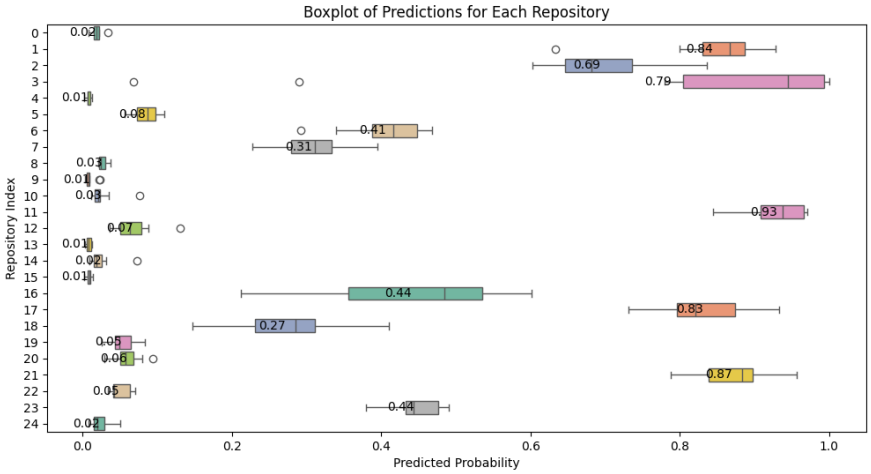


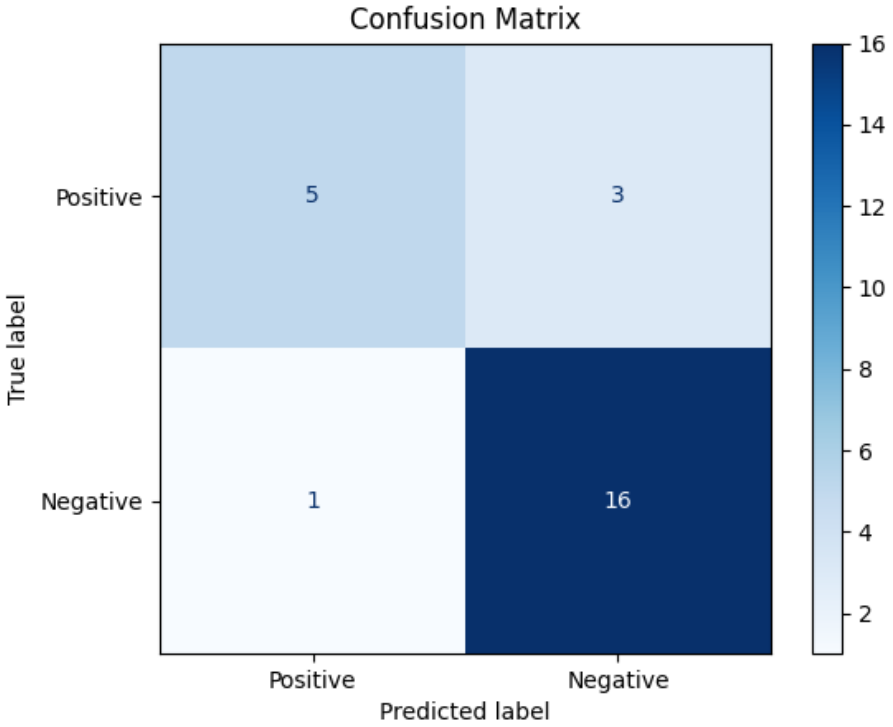


5.1.2 Evaluation of Results Produced by K-Folds Cross Validation Technique

The K-Folds Cross-Validation approach employed in the experiment yielded an accuracy of 64%, which falls below the expected range of 80-85% outlined in the experimental hypothesis ([H-2]). This establishes the experiment as a qualified success, as only the neural network model had passed the baseline accuracy threshold to garner success as defined by the hypotheses of this experiment. This result highlights several challenges and limitations associated with the K-Folds method in the context of predictive analysis.

The K-Folds Cross-Validation algorithm splits the dataset into (n) equal parts, with n being representative of the "K" in K-folds. This algorithm subsequently utilizes one part for testing and the remaining parts for training. This process is repeated (n) times, and the results are averaged to produce an overall accuracy score. The 64% accuracy indicates that the model struggled to generalize effectively across the dataset, leading to a higher rate of misclassifications compared to the Neural Network approach.





In the point of the failure of the K-Folds Cross-Validation model, the dataset may not have been large or diverse enough to allow the K-Folds algorithm to identify consistent patterns across all splits, leading to reduced accuracy. Furthermore, if the dataset contained an uneven distribution of well-maintained and inadequately maintained repositories, the model may have been biased toward the majority class, affecting its ability to make accurate predictions. The interdependencies between the metrics used (contributor count, commit

frequency, issue turnover ratio) may have been too complex for the K-Folds approach to capture effectively without the need to further develop or include additional features or preprocessing to ensure that it can adequately assess the dataset used during the experimentation process. Overfitting to Training Data: The model may have overfit to certain splits of the training data, resulting in poor generalization to the test data.

5.1.3 Statistical Analysis Experimental Results

5.2 Key Findings

In this case, the artificial neural network model had a generally higher performance margin in comparison to that of the K-Folds Cross-Validation algorithm. This may in part be due to the higher degree of robustness contained within a neural network, as well as the vulnerabilities contained within a K-Folds algorithm, such as data leaking.

5.3 Interpretation of Results

5.3.1 Comparisons with Existing Research

The experiment conducted with EvaluRate employed two machine learning approaches: a neural network model and K-Folds Cross Validation model, to validate repository maintenance scores. These approaches were designed to address limitations in existing tools like Reaper and PHANTOM, which use rule-based [10] and unsupervised learning methods [8], respectively.

5.3.1.1 reaper Reaper evaluates repositories using seven predefined metrics (e.g., community, documentation, license) [8] and assigns a binary score (1 or 0) to indicate whether a repository is an engineered software project [8].Reaper uses deterministic methods, such as searching for specific configuration files or license excerpts, to evaluate repositories. This makes it straightforward but less flexible.

5.3.1.2 PHANTOM PHANTOM uses k-means clustering, an unsupervised machine learning method, to group repositories based on temporal activity patterns derived from Git logs [10]. PHANTOM also evaluates repositories using five metrics [10] extracted from Git logs, focusing on temporal trends rather than static attributes [10]. This can in turn compromise the integrity of the model, as time-series data is of a higher priority and focus of measure than the tool, rather than other applicable metrics that could have an equivalent or more significant effect than solely time-based properties or variables. Phantom was also designed to handle large-scale evaluations (1.8 million repositories) [10] on a single machine, PHANTOM prioritizes efficiency over accuracy. Furthermore, the peak accuracy across all of PHANTOM's training was only 60-70% which in the case of large-scale testing, is not a reliable means of accuracy for sustained usage, or to even be deemed a reliable means of assessment of a GitHub repository.

5.4 Implications of Findings

Before the findings are broken down, it is paramount to note that one of the primary faults of this project is a lack of objective determination.

The 84% accuracy achieved by the neural network demonstrates that EvaluRate's calculated maintenance scores are reliable and consistent with machine learning validation. This result supports the hypothesis ([H-2]) that the neural network would achieve an accuracy within the range of 80-85%. The findings highlight the potential of neural networks as a robust validation method for repository maintenance evaluation tools.

6 Conclusion

6.1 Summary of Results

The neural network's performance in achieving 84% accuracy underscores its effectiveness in validating EvaluRate's maintenance scores. By leveraging diverse metrics and a comprehensive dataset, the model demonstrated strong predictive capabilities, providing confidence in the reliability of EvaluRate as a tool for assessing GitHub repository maintenance.

The neural network's superior performance over the K-Folds Cross Validation approach can be attributed to its ability to handle complex, non-linear relationships between metrics, its adaptability to diverse datasets, and its strong generalization capabilities. In contrast, the K-Folds Cross-Validation approach struggled with the dataset's size, diversity, and complexity, leading to a lower accuracy of 64%. These findings highlight the importance of selecting appropriate machine learning methods based on the characteristics of the dataset and the problem being addressed. Future work should focus on optimizing the K-Folds approach through stratified sampling, feature engineering, and larger datasets to improve its performance.

6.2 Future Work

One of the main considerations is enhancing the time overhead, especially in respect to the acquisition of the data itself, as requests are limited by the GitHub REST API. One proposed method to aid in cutting down on time constraints related to data processing is the use of transfer learning [6]. This approach is an optimization technique that is designed with the purpose to reduce time overheads of training machine learning models, as well as to improve performance of the respective machine learning models being trained [6]. The matter of transfer learning itself is a semi-supervised classification technique [6] that involves the application of a source domain and task to their respective target counterparts [6]. The method itself makes use of large swatch of unlabeled data, and a small set of labeled data. [6] This break-down of data allows for a smaller partition of data to serve as a template for the remaining majority of the data, allowing for a faster training process by applying labels, distributions, and domains from the template to the whole of the dataset [6].

This use of transfer learning also allows for the training and testing datasets to have different characteristics, specifically "the domains, tasks, and distributions used in training and testing" [6] of machine learning models. This supplements the feasibility of carrying out highly complex and intricate training processes, which in the case of EvaluRate, may be required to enhance the machine learning methods employed to reinforce the respective validity and performance of both models. By allowing the dataset to be split into various domains, each with a respective target function and task, it allows for a model being trained to account for potentially multiple variables within its training with each partition of the data provided to it [6]. In this case, it can be applied to EvaluRate to assess predictions based on potential individual relationships that the metrics evaluated have upon one another, or on the maintenance score itself. Utilizing transfer learning, especially with an emphasis on using other version control services such as Bitbucket, Beanstalk, AWS CodeCommit, or Apache Subversion, can be used to establish a standard across services.

By working with data from different version control services in independent domains, it is possible that a machine learning model could train off of data from each version control service to evaluate for well-maintained software projects or repositories. This would allow

the machine learning model(s) produced by this method to acclimate and adapt to patterns found across several different version control software programs or services, effectively analyzing and pathing out a common relation between the different data domains provided to it. This would allow for a standard architecture to be established, simplifying the process of predicting and calculating what attributes per respective platform can be equated to one another to create respective parameters for evaluation. This itself is valuable, because it can provide insight into variations of prevalent characteristics of code bases or repositories on these platforms, especially if one or multiple thereof hold unique variables or data regarding code bases that could serve to improve the effectiveness of predicting if a repository is well-maintained or otherwise.

This approach also supports extremely large databases, ranging from "terabytes or even petabytes" [6] per run during the training process of machine learning models, especially high-performance computing systems [6]. This proactively resolves potential issues that may arise in relation to time and computational constraints that would presumably arise when scaling up the scope of EvaluRate, especially to the degree where the software is regularly consuming and processing several terabytes or more of data in the training of its machine learning models. This would allow EvaluRate to handle the increased workload of handling larger data queries from GitHub, especially when also receiving data from request queries sent out to other version control software or services.

Another approach to improving EvaluRate is the use of dataset proportioning. In respect to the previous aspect of scaling up EvaluRate, this tactic offers support with compression of data, to allow for to storage overhead issues [5] for large swaths of data. This approach also has the potential to improve accuracy to predictions made by machine learning models trained through this process [5].

In this case, a dataset is broken down into chunks, with a partition of this dataset serving as a domain source. This approach is offered as a solution to the drawback of using a compression autoencoder [5], which has a primary fault of prolonged training times [5]. A compression autoencoder itself is a neural network architecture designed with the purpose in mind to compress or encode large bodies of data down to its most vital aspects, and subsequently reconstruct the data from its compressed state [Liu et al. [5]][12].

6.3 Future Ethical Implications and Recommendations

The accessibility of the artifact is an object of concern, as it allows individuals, or potentially organizations to access and view data from repositories of GitHub users without an account of means of verification. Traditionally, an individual attempting to access content on GitHub would need an account in order to access any meaningful features and content on the site. This tool provides a window into GitHub for any potential use-case or type of user, and lacks a means of protecting the data used in this experiment. Implementing a sign-in page that is linked to a GitHub account to access EvaluRate would be an effective means of protecting the user data that was used to train and test EvaluRate and the machine learning models within. In spite of this tool being created with a public-oriented use-case in mind, implementing security measures to protect personal data, and the artifact itself will be required in the future, especially if it is to be further developed and deployed on a larger scale for additional version control software besides GitHub.

Another point of contention is the objectivity of the evaluations performed by this tool. It has been already established that there would be a need to a way to identify repositories as objectively well maintained or otherwise inadequately maintained. There currently is no

manner by which the repositories are labelled with a "ground truth", by which it is it meant that the predictions made by both machine learning algorithms are based on objectively true identification scores made by EvaluRate prior to machine learning assessment.

References

[1]]CodeClimate [n. d.]. Code Climate Documentation. https://docs.codeclimate.com/

[2] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L. Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology* 122 (2020), 106274. https://doi.org/10.1016/j.infsof.2020.106274

[3] Russ Cox. 2019. Surviving Software Dependencies. *Associated of Computing Machinery* 17, 2 (2019). https://queue.acm.org/detail.cfm?id=3344149

[4] Ljubomir Lazic. 2008. Cost Effective Software Test Metrics. *WSEAS Transactions on Computers* 7 (2008), 599–619. https://doi.org/10.5555/1458369.1458372

[5] Tong Liu, Shakeel Alibhai, Jinzhen Wang, Qing Liu, and Xubin He. 2021. Reducing the Training Overhead of the HPC Compression Autoencoder via Dataset Proportioning. *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)* (2021), 1–7. https://doi.org/10.1109/NAS51552.2021.9605407

[6] Tong Liu, Shakeel Alibhai, Jinzhen Wang, Qing Liu, Xubin He, and Chentao Wu. 2019. Exploring Transfer Learning to Reduce Training Overhead of HPC Data in Machine Learning. *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)* (2019), 1–7. https://doi.org/10.1109/NAS.2019.8834723

[7] Mario Lucic, Mesrob I. Ohannessian, Amin Karbasi, and Andreas Krause. 2016. Tradeoffs for Space, Time, Data and Risk in Unsupervised Learning. (2016). arXiv:1605.00529 [stat.ML]

[8] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[9] Jordan Novet. 2017. GitHub, popularized by indie software developers, now books more than 100m a year from businesses. *CNBC* (2017).

[10] Peter Pickerill, Heiko Joshua Jungen, Mirosław Ochodek, Michał Maćkowiak, and Miroslaw Staron. 2017. PHANTOM: Curating GitHub for engineered software projects using time-series clustering. *Empirical Software Engineering* 25, 4 (2017), 2897–2929.

[11] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

[12] Edward Keefer Tong Wu, Wenfeng Zhao and Zhi Yang. 2018. Deep compressive autoencoder for action potential compression in large-scale neural recording. *Journal of Neural Engineering* 15 (2018), 1741–2552. https://doi.org/10.1088/1741-2552/aae18d