การเรียนรู้หลายผลลัพธ์สำหรับการทำนายการประเมินผลและการพิจารณาใหม่ของกิตฮับพูลรีเควสบนโอเพ่นซอร์สโปรเจค

นายพีระชัย บรรยงรักษ์กุล

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาสถิติ ภาควิชาสถิติ
คณะพาณิชยศาสตร์และการบัญชี จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2565

MULTI-OUTPUT LEARNING FOR PREDICTING EVALUATION AND REOPENING OF GITHUB PULL REQUESTS ON OPEN-SOURCE PROJECTS

Mr. Peerachai Banyongrakkul

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Statistics
Department of Statistics
Faculty of Commerce and Accountancy
Chulalongkorn University
Academic Year 2022

Thesis Title	MULTI-OUTPUT LEARNING FOR PREDICTING EVALUATION AND REOPENING OF GITHUB PULL REQUESTS ON OPEN-SOURCE PROJECTS
By	Mr. Peerachai Banyongrakkul
Field of Study	Statistics
Thesis Advisor	Assistant Professor Suronapee Phoomvuthisarn, Ph.D.

Accepted by the Faculty of Commerce and Accountancy, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master of Science

. . . . Dean of the Faculty of Commerce and Accountancy

(Professor Wilert Puriwat, Ph.D.)

พีระชัย บรรยงรักษ์กุล: การเรียนรู้หลายผลลัพธ์สำหรับการทำนายการประเมินผลและการพิจารณาใหม่ของกิตฮับพูลรีเควสบนโอเพ่นซอร์สโปรเจค. (MULTI-OUTPUT LEARNING FOR PREDICTING EVALUATION AND REOPENING OF GITHUB PULL REQUESTS ON OPEN-SOURCE PROJECTS) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ผศ.ดร.สุรณพีร์ ภูมิวุฒิสาร, 118 หน้า.

โมเดลการพัฒนาซอฟต์แวร์ของกิตฮับแบบพูลถูกนำมาใช้งานกันอย่างแพร่หลาย โดยที่คอนทริบิวเตอร์สามารถสร้างพูลรีเควส (คำขอดึงโค้ด) เพื่อรวมการเปลี่ยนแปลงเข้ากับที่เก็บรวมโค้ดหลักของโครงการ และอินทิเกรเตอร์มีหน้าที่ในการพิจารณาพูลรีเควสพวกนี้เพื่อรักษาคุณภาพ และความเสถียรภาพ อย่างไรก็ตาม จำนวนคำขอที่มากอาจทำให้อินทิเกรเตอร์ต้องรับภาระงานเพิ่มขึ้น และทำให้มีความล่าช้าในการตอบกลับคอนทริบิวเตอร์ งานวิจัยก่อนหน้านี้ใช้เทคนิคของการเรียนรู้ของเครื่องและเทคนิคทางสถิติด้วยข้อมูลตาราง แต่อาจสูญเสียข้อมูลที่สำคัญได้ นอกจากนี้ เฉพาะการยอมรับและระยะเวลาในการพิจารณาอาจเป็นปัจจัยที่ไม่เพียงพอสำหรับการประเมินพูลรีเควส อีกทั้งการเปิดเพื่อพิจารณาของพูลรีเควสอีกครั้งสามารถเพิ่มค่าใช้จ่ายในการดูแลรักษา และเพิ่มภาระการทำงานของนักพัฒนาที่มีงานต่างๆมากอยู่แล้ว วิทยานิพนธ์นี้จึงนำเสนอวิธีการใช้การเรียนรู้เชิงลึกแบบหลายผลลัพธ์แบบใหม่ ซึ่งสามารถทำนายการยอมรับ ระยะเวลา และการเปิดเพื่อพิจารณาใหม่ของพูลรีเควสล่วงหน้า โดยรองรับแหล่งข้อมูลที่หลากหลายอย่างมีประสิทธิภาพ ในที่นี้หมายถึงข้อมูลแบบตารางและข้อมูลข้อความ วิธีการของผู้จัดทำยังใช้เทคนิค SMOTE และ VAE เพื่อจัดการกับความไม่สมดุลของการเปิดเพื่อพิจารณาพูลรีเควสใหม่ ผู้จัดทำได้ประเมินผลการทดลองของวิธีการด้วยพูลรีเควส จำนวน 143,886 จาก 54 โครงการที่รู้จักกันอย่างแพร่หลายใน 4 ภาษาโปรแกรมยอดนิยม ผลการทดสอบแสดงให้เห็นว่าวิธีการของผู้จัดทำมีประสิทธิภาพมากกว่าบรรทัดฐานแบบสุ่ม และช่วยเพิ่มความแม่นยำได้ถึง 8.68% และเพิ่ม F1-Score ได้ถึง 6.77% ในการทำนายการยอมรับ และ 6.07% สำหรับ MMAE ในการทำนายระยะเวลา พร้อมทั้งเพิ่มความแม่นยำที่ถูกปรับเพื่อความสมดุล ไปถึง 9.43% และ AUC ไปถึง 9.37% ในการทำนายการเปิดเพื่อพิจารณาใหม่ เมื่อเปรียบเทียบกับวิธีการที่มีอยู่ในปัจจุบัน

ภาควิชา	สถิติ	ลายมือชื่อนิสิต	....
สาขาวิชา	สถิติ	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2565		

6480460026: MAJOR STATISTICS
KEYWORDS: PULL-BASED SOFTWARE DEVELOPMENT / PULL REQUEST / GITHUB / DEEP LEARNING / MULTI-OUTPUT LEARNING / CLASSIFICATION

PEERACHAI BANYONGRAKKUL : MULTI-OUTPUT LEARNING FOR PREDICTING EVALUATION AND REOPENING OF GITHUB PULL REQUESTS ON OPEN-SOURCE PROJECTS. ADVISOR : Asst. Prof. SURONAPEE PHOOMVUTHISARN, Ph.D., 118 pp.

GitHub's pull-based development model is widely used by software development teams to manage software complexity. Contributors create pull requests for merging changes into the main codebase, and integrators review these requests to maintain quality and stability. However, a high volume of pull requests can overburden integrators, causing feedback delays. Previous studies have used machine learning and statistical techniques with tabular data as features, but these may lose meaningful information. Additionally, acceptance and latency may not be sufficient for the pull request evaluation. Moreover, reopened pull requests can add maintenance costs and burden already-busy developers. This thesis proposes a novel multi-output deep learning-based approach that early predicts acceptance, latency, and reopening of pull requests, handling various data sources, including tabular and textual data, effectively. Our approach also applies SMOTE and VAE techniques to address the highly imbalanced nature of the pull request reopening. We evaluate our approach on 143,886 pull requests from 54 well-known projects across four popular programming languages. The experimental results show that our approach significantly outperforms the randomized baseline. Moreover, our approach improves Accuracy by 8.68% and F1-Score by 6.77% in acceptance prediction, and MMAE by 6.07% in latency prediction, while improving Balanced Accuracy by 9.43% and AUC by 9.37% in reopening prediction over the existing approach.

Department:	Statistics	Student's Signature	....
Field of Study:	Statistics	Advisor's Signature	....
Academic Year:	2022		

ACKNOWLEDGEMENTS

First of all, I would like to sincerely express my appreciation to my thesis advisor, Asst. Prof. Suronapee Phoomvuthisarn, for his invaluable guidance and support throughout my thesis journey. Furthermore, I am grateful for his patience and dedication in providing constructive feedback on my writing, which has helped me to refine my ideas and present them more effectively. I would also like to extend my gratitude to the committee members for providing valuable comments and practical suggestions on the study. Their insightful feedback and suggestions have significantly contributed to the improvement of this research.

In addition, I would like to convey my heartfelt gratitude to my beloved parents, who have been the driving force behind my academic achievements. Their unwavering support, encouragement, and sacrifices have been instrumental in enriching my academic journey. They have always believed in me and pushed me towards higher education, providing me with the necessary financial assistance and emotional support.

Finally, I would like to thank the faculty and instructors of the Department of Statistics. The knowledge and skills that I have gained from their teaching have been instrumental in shaping my thesis and enhancing my knowledge and skills in the field. I would also like to thank my friends for their constant words of encouragement and for exchanging ideas with me.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Page

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

In modern software development, developer teams usually deal with the complexity of software using isolated development and branching (Bird and Zimmermann, 2012). Thus, several open-source software projects employ the pull-based model (Bird et al., 2009), an emerging paradigm for distributed software development, enabled by GitHub[1] (Gousios et al., 2014). It allows *contributors* to make contributions (e.g., software changes) flexibly and efficiently (Gousios et al., 2016). Unlike classical distributed development, GitHub enables developers to fork the main repository that they want to contribute to and make their changes independently and locally (Yu et al., 2015). When a set of changes is ready, they require to create a requesting event, called a *pull request*, to ask for merging the changes with the main repository. Then, the project's *integrators* are responsible for evaluating the pull request and deciding whether to accept or reject the changes. The role of integrators in the pull-based model is crucial (Dabbish et al., 2013) because they must not only make important decisions but also ensure that pull requests are evaluated in a timely matter. In popular projects, the volume of incoming pull requests is too large (Tsay et al., 2014). Therefore, it may increase the burden on already-busy integrators (Gousios et al., 2015) and cause contributors to experience delayed feedback (Gousios et al., 2016).

Thus, several studies using Machine Learning, statistical techniques, and survey have been proposed to explore and support the pull-based development model, especially integrators. There have been two main ways to study the pull request evaluation, consisting of the decision to merge (i.e., *acceptance*) and the merging time (i.e., *latency*). Most works investigated factors influencing the pull request acceptance (Gousios et al., 2014; Tsay et al., 2014; Gousios et al., 2015; Soares et al., 2015; Ortu et al., 2020; Zhang et al., 2021a) and latency (Gousios et al., 2014; Yu

[1]https://github.com/

et al., 2015; Gousios et al., 2015; Zhang et al., 2021b), while a few works focused on building a prediction model for acceptance (Chen et al., 2019; Nikhil Khadke, 2012; Jiang et al., 2020) and latency (de Lima Júnior et al., 2021) There are some previous studies working on other fields of pull request aside from the evaluation, for example, pull request prioritization (Van Der Veen et al., 2015; Azeem et al., 2020), pull request overdue acceleration system (Maddila et al., 2022), and integrator assignment to pull requests (de Lima Júnior et al., 2018).

Acceptance and latency seem to be insufficient for the pull request evaluation. After a pull request is closed by an integrator, in some cases, it may be opened again for further modification and code review (Mohamed et al., 2018). This pull request is called a *reopened pull request*. Even though reopened pull requests rarely happen (Jiang et al., 2019), they may create conflicts with newly submitted pull requests (McKee et al., 2017), add software maintenance costs, and increase the burden for already-busy developers (Mohamed et al., 2018). Two studies (Mohamed et al., 2018, 2020) have developed models for predicting reopened pull requests, but they make predictions at the first decision, which may be too late. Integrators would benefit from earlier prediction results to identify pull requests more likely to be reopened and come up with timely solutions. However, early prediction is very challenging due to limited available information. If only common tabular features that are available, similar to existing approaches, are used, it may not be sufficient to create accurate predictions.

In this research, we introduce a novel multi-output deep learning-based approach that predicts acceptance, latency, and reopening of pull requests at the time of submission. Specifically, the predictions can be generated and provided to integrators as feedback immediately after the pull request is created. We make use of deep learning to focus on automating and enhancing performance while overcoming the limited information available at submission time and the highly imbalanced nature of reopened pull requests. In particular, to tackle the limited information, we incorporate both tabular data and text data from the pull request description as a new source of information and handle the nature of the text by using pre-trained models (e.g., Word2vec, FastText, and BERTs). To overcome the highly imbalanced nature, we employ SMOTE and VAE techniques.

Additionally, we address the relationship between pull request outputs, as previous research has shown that reopened pull requests have lower acceptance rates and longer evaluation times than non-reopened ones (Soares et al., 2015; Jiang et al., 2019) by sharing learning between outputs. Regarding the methodologies, we address a gap in pull request prediction research by using a programming language-specific experimental setting that balances specificity and generalization. It avoids the cold-start problem for new projects and overly generalized models, which has been a limitation in prior research that evaluated models at the project or all-in-one level. Also, previous studies have highlighted the significance of programming language in pull request evaluation (Rahman and Roy, 2014; Soares et al., 2015). For example, the projects from R language made more accepted pull requests on average than rejected ones while the projects written by Python, Java, and Ruby made the opposite. In addition, Ruby-based projects are often found with an increasing number of rejected pull requests per month.

We performed extensive experiments on four well-known programming languages including, Python, R, Java, and Ruby, along with popular representatives of open-source software projects that use the pull-based development model on GitHub. Our approach outperformed the randomized baseline with an Accuracy of 0.762, a Precision of 0.878, a Recall of 0.791, and an F1-Score of 0.832 in acceptance prediction, an MMAE of 1.163 in latency prediction, and a Balanced Accuracy of 0.618, an AUC of 0.689, and a TPR of 0.694. in reopening prediction, on average. Compared to an existing approach, our approach improved Accuracy, Precision, Recall, F1-Score, MMAE, Balanced Accuracy, AUC, and TPR by 8.68%, 1.01%, 11.49%, 6.77%, 6.07%, 9.43%, 9.37%, and 30.07%, respectively.

The main contribution of this research is that we aim to propose a novel approach that combines tabular features and textual features learned from pre-trained word embeddings and leverages SMOTE along with VAE and shared learning deep neural network to have the ability to predict pull request acceptance, latency, and reopening at the time of submission.

1.1 Problem Statements

We conclude the introduction into the problems and gaps that we aim to address in this study listed as followings:

- The emergence of pull-based development increases the burden on already-busy developers which can cause low productivity, especially in large projects.
- Pull request reopening is not of interest to most existing studies, but it is crucial aside from other main evaluations (i.e., acceptance and latency) because it can cause conflict and increase costs.
- The highly imbalanced nature of reopened pull requests presents a challenge for machine learning-based prediction models.
- Most existing studies had more focus on investigating factors influencing the pull request evaluation, than prediction.
- Existing prediction approaches did not address the nature of pull request textual description, usually requiring advanced feature extraction techniques.
- Early prediction of reopened pull requests is challenging due to limited available information at submission time.

1.2 Objectives

We aim to address the mentioned problem statements by developing the following research objectives:

- To design an architecture of an approach, that can capture the relationships between the outcomes, for predicting pull request acceptance, latency, and reopening.
- To handle the highly imbalanced nature of pull request reopening with a proper oversampling technique.
- To construct the rich features by handling textual data effectively through exploring advanced texutal feature extraction techniques that can address limited available information.

1.3 Our Solution

We propose a novel deep-learning-based multi-output (shared learning) classification approach that is generalized for predicting three pull request outcomes which are acceptance, latency, and reopening early at the time of pull request submission. The approach incorporates the following techniques:

- SMOTE combined with Variational Autoencoder (VAE) to address the highly imbalanced nature of pull request reopening.
- Various pre-trained embeddings to address textual description of pull requests and handle limited information available at the pull request submission.
- Implementation of a shared learning deep learning architecture to handle the relationships between pull request outputs and perform classifications.

1.4 Research Questions

In this study, our empirical evaluation aims to answer the following research questions:

- **RQ1**: Is the proposed approach suitable to predict the pull request evaluation and pull request reopening? - This is a sanity check that allows us to check on the suitability of our proposed approach in predicting pull request evaluation and reopening by comparing it with the randomized baseline. We hypothesize that our approach will outperform the baseline, demonstrating its ability to accurately predict pull request acceptance, latency, and reopening at pull request submission.

- **RQ2**: Does the proposed approach outperform the existing approach? - This aims to compare our proposed approach, which incorporates tabular and textual features, oversampling, shared learning, and a deep learning classifier, with the existing approach. We anticipate that our approach will enhance the predictive performance in predicting pull request evaluation and reopening, surpassing the existing approach.

The details of the research questions are elaborated more in Section 6.1.

1.5 Scopes of Work

To do the experiment, we have a list of the scopes as followings:

- We conduct a study on data from Aug 2010 to Aug 2022, analyzing four well-known programming languages: Python, R, Java, and Ruby, and popular representatives of open-source software projects utilizing the GitHub pull-based development for each programming language.
- Our evaluation focuses on the predictive performance achieved by our approach based on our empirical study. We, however, acknowledge that user evaluation and feedback are important to the adoption of the approach which we plan to address in future work.

1.6 Expected Benefit

Our approach is intended to enhance GitHub pull request management, particularly for large software open-source projects, by offering predictive results in the form of pull request evaluations and reopening, enabling integrators to make informed decisions regarding the risk involved in handling a particular pull request. This will streamline the review process by facilitating efficient and systematic prioritization, ultimately improving the overall efficiency of pull request management.

1.7 Organization of the Document

This document consists of seven chapters, plus one appendix:

1. Introduction – Presentation of reasons for doing this study including short background, motivations, problem statements, objectives, our solution, scopes of this project, and expected benefit
2. Background – Review of concept and theoretical background along with literature
3. Dataset – Overview of the data and processes of data collection and data labeling we apply to perform the experiment in this project
4. Proposed Approach – Explanation of proposed approaches with design and methodologies
5. Implementation - Detailed description of the implementation of the proposed approaches, including any relevant software tools or frameworks utilized.
6. Evaluation and Results – Statements of research questions, experimental setting, performance measures, experimental results, further discussion, and any threats to validity
7. Conclusions - Summary of the study and discussion about future works
8. Appendix A - Comprehensive detail of the model configuration of our approach.

CHAPTER II

BACKGROUND

This chapter provides the conceptual and theoretical background of the research, followed by a comprehensive literature review on the topic of pull request evaluation and reopening prediction. In the first section, we present the key concepts and theories related to the research, including an overview of the pull-based software development model, the significance of prediction in software development, and machine learning. The literature review section presents a critical analysis of existing studies on pull request evaluation and reopening prediction, identifying gaps in knowledge and areas that require further investigation.

2.1 Concept and Theoretical Background

In this section, we provide background information about the pull-based software development model along with the pull request life cycle and the machine learning techniques that we aim to employ in this study.

2.1.1 Pull-Based Software Development Model

To have a better understanding of the pull-based model, this section consists of the pull request workflow and the pull request life cycle.

Pull Request Workflow

With the emergence of Git, the way distributed software development (DSD) is carried out has been revolutionized (Gousios et al., 2014). Git enables open-source projects to deploy a recent paradigm of DSD, named pull-based development. Git is the fastest growing compared to others. Stack Overflow's annual developer survey report that Git was the overwhelming favorite of responding developers, showing as high as 93.9% in 2022. It draws ahead of the second place, like Subversion, which dominates developers only 5.2% (Stack Overflow, 2022). In the pull-based software development model, developers pull software changes (e.g., usually source code files) from other repositories and merge them locally, instead of pushing the changes to a project's main repository directly by patch submission and acceptance through mailing lists or Bugzilla[2], like traditional distributed development (Yu et al., 2015).

Figure 2.1 is an overview of the pull-based development workflow. When developers want to make some contributions to an open-source project, GitHub pull-based development provides a chance that contributors to the project need not share access to the main repository anymore. Instead, contributors can create forks and make their changes (e.g., deletions and additions to the source code) locally without having to worry about breaking the overall software product. Once the contributors have completed their code changes, they push the changes back to their forked repository. When a set of changes is ready to be submitted to the main repository, they require to create an event, called a pull request, to request for discussing and merging new code changes with the main repository. An integrator will inspect and review the changes in the contributor's forked repository. If any discussion is required, the integrator and the contributor can provide comments (i.e., including in-line comments) under the pull request. Moreover, external developers who are not directly involved can view and join the discussion. The contributor needs to create additional commits if further improvements or edits are required from the integrator before an approval. With approved the pull request, the new updates from the contributor are merged into the main project. It is, however, up to the integrator's decision to accept or reject the incoming pull requests. The risk of consequences, therefore, depends on the integrator's experience. So, although developing software via pull requests will help in the matter of flexibility and efficiency of distributions, it also increases the workload for software developers because of the large volume of incoming pull requests, especially in the case of the popular projects (Gousios et al., 2015).

[2]https://www.bugzilla.org/

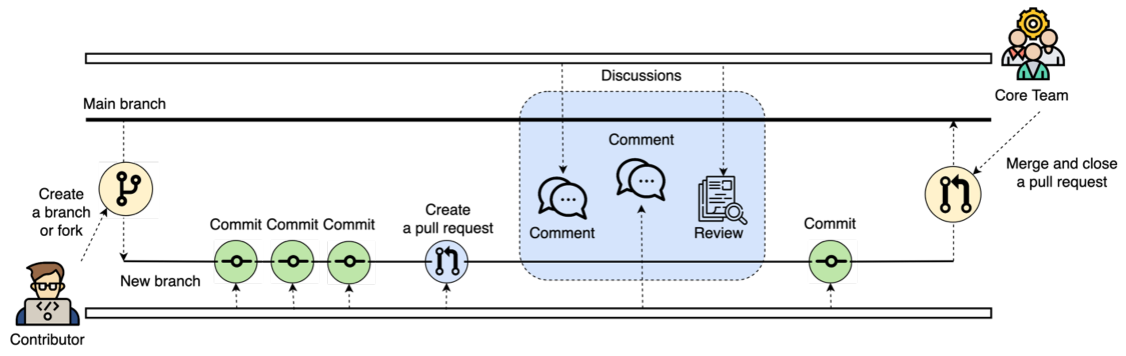


Figure 2.1: An overview of the pull-based development workflow

Pull Request Lifecycle

There are three states of pull requests on GitHub as shown in Figure 2.2 including:

- **Open**: the pull request has been proposed by the contributor and it is during discussions or waiting for the integrator's decision on whether it will be accepted or rejected.
- **Merged**: the integrator satisfies the changes in the pull request. The integrator, thus, approves closing it by merging them with the main branch.
- **Closed**: the integrator is not satisfied with the changes. The integrator, thus, closes it by rejecting the pull request.

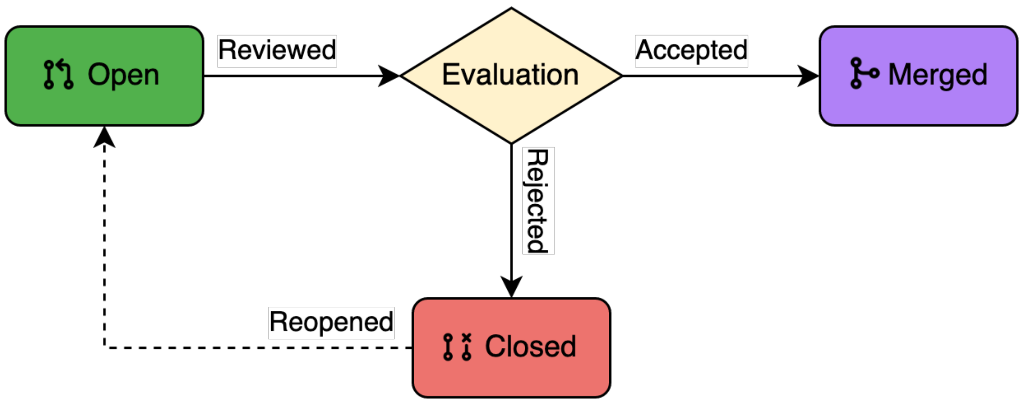


Figure 2.2: Pull request lifecycle on GitHub

Pull requests, however, sometimes remain open forever because the integrators are too busy to tackle the pull requests, or it is simply because the integrators do not want to discourage the contributor by explicitly rejecting the pull request. Apart from the states above, the pull requests can be reopened after their close when the decision is changed, or further code review is required. The contributor can attempt further updates to reopen the reviewing process, this may lead to a new decision from the integrator. These pull requests are called reopened pull requests. Reopening a pull request is considered as a bad risk because it can cause the integrator to take more effort (e.g., add software maintenance costs and increase the burden for an already busy integrator) (Jiang et al., 2019). Moreover, it may cause conflicts with newly submitted pull requests if pull requests are reopened a long time after their close (McKee et al., 2017). For example, changes from a current pull request create a bug caused by not accepting a previous pull request. This can happen due to integrator evaluation glitches. Therefore, in order to fix the current bug, reworking the previous pull request is required for the integrator.

Therefore, identifying the quality of pull requests is important and this is called *pull request evaluation*. Evaluating pull requests is a complex iterative process involving many stakeholders. Currently, there are two main ways of evaluation that researchers use to investigate the pull request workflow which are *acceptance* and *latency* (Yu et al., 2015). The researchers focusing on the pull request acceptance study about the factors influencing the decision of integrators on whether accepting or rejecting pull requests. The researchers focusing on the pull request latency explore the factors influencing the lifetime of pull requests. Moreover, the relationship between both aspects is studied in (Soares et al., 2015). They found that an increase in the evaluation time for a pull request reduces the chances of its acceptance. There are also the relationships between both pull request evaluation aspects and reopening, for example, reopened pull requests have lower acceptance rates and longer evaluation time than non-reopened pull requests (Jiang et al., 2019). Therefore, the pull request evaluation and the pull request reopening notification have benefits to the integrators by encouraging their decisions, prioritizing the pull requests, and speeding up the review process which can lead to accelerating software product development.

2.1.2 Developing a Predictive Model

A predictive model is a statistical process of using data to forecast outcomes (Geisser, 1993), usually in the future. Developing a predictive model can be represented as

$$Y = f(X) \tag{2.1}$$

where $f(\cdot)$ is a function for an algorithm and X is a set of features whereas Y is a set of prediction outcomes. There are several approaches to the predictive model including Machine Learning and Deep Learning. However, we focus on using Deep Learning, especially in the classification task. Deep Learning is a subset of Machine Learning that utilizes a hierarchical level of artificial neural networks, inspired by the biological brain, to the process of learning. The most common reasons that Deep Learning is preferred to traditional Machine Learning are a large amount of data, no requirement for hard-core feature engineering and domain experts (automation), and higher performance (Janiesch et al., 2021). This section consists of classification with deep learning, classification evaluation metrics, multi-output learning, text feature extraction, and handling imbalanced data techniques.

Classification with Deep Learning

- **Binary Classification**: Binary classification is the task of classifying the elements of a set into two groups. Common applications are spam detection, credit card fraud detection, medical testing, and sentiment analysis. To perform binary classification with Deep Learning, Sigmoid is an activation function that is mostly used in the output layer as it maps any input to an output ranging from 0 to 1 representing a probability. The formula of the Sigmoid function is defined as in Equation 2.2. However, this is similar to the SoftMax function with two classes.

$$\sigma\left(z\right)=\frac{1}{1+e^{-z}} \tag{2.2}$$

For loss function, binary cross-entropy, also known as log loss, is purpose-built for binary classifiers. During training, the cross-entropy loss function exponentially increases the penalty for wrong predictions to drive the learning parameters (e.g., weights and biases) more aggressively in the right direction. Suppose that a sample belongs to the positive class (1), and the model predicts that the probability of being a positive class is 0.9. The log loss is $–log(0.9)$ or 0.04. But if the model outputs a probability of 0.1 for the same sample, the error is $–log(0.1)$ or 1. This is obvious that if the predicted output is more wrong, the penalty is much higher. Binary cross entropy is calculated as in Equation 2.3.

$$-\left(ylog(p)+\left(1-y\right)log(1-p)\right) \tag{2.3}$$

- **Multi-Class Classification**: Multi-class classification is the task of classifying the elements of a set into more than two groups. With Deep Learning, the most common activation function for an output node is SoftMax. SoftMax converts a real vector to a vector of categorical probabilities. The elements of the output vector are in the range (0, 1) and sum to 1. Each vector is handled independently. The result could be interpreted as a probability distribution. The formula of the Sigmoid function is defined as in Equation 2.4.

$$\sigma\left(z_i\right)=\frac{e^{z_i}}{\sum_{j=1}^{K}e^{z_j}} \quad for\ i=1,2,\ldots,K \tag{2.4}$$

In the matter of loss function, there are two functions that are usually used which are categorical cross-entropy and sparse categorical cross-entropy. Both are the loss computation between the labels and predictions. However, if labels are provided in a one-hot representation, categorical cross entropy is required. On the other hand, if labels are provided as integers, sparse categorical cross entropy is needed. The formula is similar to the binary one, but we need to calculate a separate loss for each class label per observation and sum the result as shown in Equation 2.5.

$$-\sum_{c=1}^{M} y_{o,c} \log\left(p_{o,c}\right) \tag{2.5}$$

Where M is number of classes whereas y is a binary indicator (0 or 1) if class label c is the correct classification for observation o and p is predicted probability observation o is of class c.

Classification Evaluation Metrics

There are several common evaluation metrics for classification models such as Accuracy, Precision, Recall, F1-Score, AUC, TPR, FPR, and MMAE.

- **Accuracy**: Accuracy is a commonly used metric for evaluating the performance of classification models. It measures the proportion of correct predictions made by the model among all predictions made.
  - Binary-Class Accuracy is the accuracy used in the binary classification which is calculated by dividing the sum of True Positive (TP) and True Negative (TN) results by the total number of observations examined as shown in Equation 2.6.

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{2.6}$$

  - Multi-Class Accuracy is defined as the average number of correct predictions across all classes. Equation 2.7 shows the formula of accuracy for multi-class classification where N is the total number of observations, $|G|$ is the number of classes, $g(x)$ is the predicted label, $\hat{g}(x)$ is the ground truth label, and I is the indicator function, which returns 1 if the classes match and 0 otherwise.

$$Accuracy = \frac{1}{N}\sum_{k=1}^{|G|}\sum_{x:g(x)=k} I(g(x) = \hat{g}(x)) \tag{2.7}$$

– Balanced Accuracy (BA) is a performance metric that measures the average accuracy of a binary classifier over two classes with imbalanced class distribution. In other words, Balanced Accuracy is the average of the two proportions of correct predictions for each class, and it ranges from 0 to 1, where a score of 1 indicates perfect classification performance, and a score of 0 indicates random performance. It is a useful metric to evaluate classifiers on imbalanced datasets, where the distribution of the classes is not equal, and the accuracy alone can be misleading. By taking into account both True Positive Rate (TPR) and True Negative Rate (TNR) (see Equation 2.8), Balanced Accuracy provides a more comprehensive view of the classifier's performance and can help to identify the trade-offs between false positives and false negatives.

$$\text{BA} = \frac{TPR + TNR}{2} \tag{2.8}$$

- **Precision**: Precision measures the proportion of True Positive (TP) predictions among all positive predictions made by the model as shown in Equation 2.9. In other words, Precision represents the accuracy of the positive predictions made by the model.

$$Precision = \frac{TP}{TP + FP} \tag{2.9}$$

- **Recall**: Recall also known as Sensitivity or True Positive Rate (TPR), is a performance metric used to evaluate the effectiveness of a classification model in identifying positive instances. It measures the proportion of True Positive (TP) predictions among all actual positive instances in the dataset as stated in

Equation 2.10.

$$Recall = \frac{TP}{TP + FN} \tag{2.10}$$

- **F1-Score**: F1-Score is a widely used performance metric that combines Precision and Recall into a single number. It is defined as the harmonic mean of Precision and Recall as shown in Equation 2.11.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.11}$$

- **AUC**: Area Under the Curve or AUC is commonly used in the context of ROC or Receiver Operating Characteristic. It measures of the total area under the ROC curve to indicate the performance of a binary classification model that addresses the trade-off between True Positive Rate (TPR) and False Positive Rate (FPR) across different classification thresholds. It ranges from 0 to 1, with higher values indicating better performance.

- **FPR**: FPR or Fall-Out is the proportion of actual negative instances that are incorrectly classified as positive by the model. Equation 2.12 is the formula of FPR.

$$FPR = \frac{FP}{FP + TN} \tag{2.12}$$

AUC, TPR, and FPR have been used in several works (Kale et al., 2022; Trauer et al., 2021) to handle the classification with highly imbalanced data (i.e., anomaly detection). TPR can reflect the probability of detection or hit rate while FPR can represent the probability of false alarm.

- **MMAE**: MMAE is a performance measure that the ability to address the ordinal property of classes. Specifically, it can assess the distance between an actual class and a predicted one. It can also apply to the imbalanced multi-class classification because of the macro-averaged mean technique. MMAE has been used in many works (Baccianella et al., 2009; Choetkiertikul et al., 2015, 2018; Wattanakriengkrai et al., 2019) to tackle the ordinal multi-class classification problem. The formula of MMAE is defined in Equation 2.13 where K is a set of classes, $|K|$ is the number of classes, k is a class within K, y^i is the true class, and n_k is the number of true classes with class k. σ is the indicator function.

$$MMAE = \frac{1}{|K|}\sum_{k=1}^{K}\frac{1}{n_k}\sum_{i=1}^{n}|\hat{y}^i - k|\sigma|y^i = k| \tag{2.13}$$

Multi-Output Learning

Multi-output learning is to learn and simultaneously predict multiple related outputs given an input. It is an important learning problem for decision-making since making decisions in the real world often involves multiple complex factors (Xu et al., 2020). The outputs can be of various types and structures, for example, real-valued vector (regression) and binary vector (binary classification). However, we will focus on multi-output learning for classification. Let $X = \mathbb{R}^d$ denote the d-dimensional input (i.e., feature) space and $Y = Y^1 \times Y^2 \times \bullet\bullet\bullet \times Y^n$ denote the output space which contains n class variables (i.e., outputs) in which output space Y^i has k_i possible classes (i.e., $Y^i = C_1^i, \ldots, C_{k_i}^i$). Giving a training dataset, $D = \{x_j,\ y_j\}_{j=1}^{M}$ where $x_j = \left[x_1^j, x_2^j, \ldots, x_d^j\right] \in X$ and $y_j = \left[y_1^j, y_2^j, \ldots, y_n^j\right] \in Y$. Thus, the task of multi-output classification is to learn a mapping function, $f : X \rightarrow Y$, from the dataset D that can predict a proper output vector, $f(x) \in Y$, for unseen input x (see Figure 2.3). Multi-output classification is sometimes called multi-label classification when all outputs have two possible classes but when some outputs have more than two possible classes, it is called multi-class multi-output classification, also known as, multi-dimensional classification, instead where our work falls into this type of problem.

In Deep Learning, there are three common ways to design an architecture of multi-output classification which are hard parameter sharing, partial parameter sharing, and soft parameter sharing. In hard parameter sharing, it is generally applied

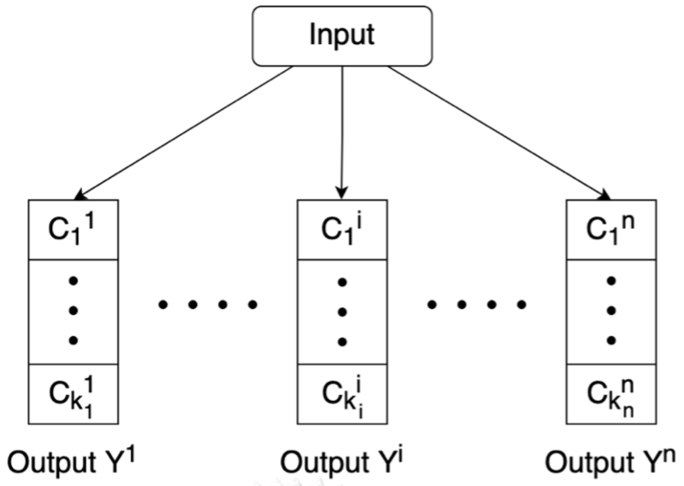


Figure 2.3: An illustration of a multi-output learning classification

by sharing the hidden layers between all outputs, while separating nodes for specific outputs in the output layer. Therefore, every output learning shares the same set of parameters and optimizes mean or weighted loss function of the same parameters. An example of the existing works employing this architecture is a multi-output classification for predicting human emotion, age, gender, and race (Saroop et al., 2021) and a multi-modal multi-output model for automatic assessment of voice pathologies on the GRB scale (Arias-Londoño et al., 2019). In partial parameter sharing, there is weight sharing on the first parts of hidden layers while the rest layers are trained independently for each output with their own loss function and optimizer. On the other hand, there are separated training paths for each output in every hidden layer in soft parameter sharing. Also, each output variable has its own set of parameters and loss. The work example of partial parameter sharing and soft parameter sharing is a multi-output learning framework for solar atmospheric parameters inferring from stokes profiles (Knyazeva et al., 2022).

Text Feature Extraction

To deal with Machine Learning, the input requires to be in the form of a numerical vector. As we aim to include text in our approach, textual information needs some technique to represent as numbers. There are several approaches to implement text feature extraction, such as, BoW (Harris, 1954), N-Gram (Broder et al., 1997), and TF-IDF (Salton and Buckley, 1988). However, these approaches destroy the sequential nature of the text and cause the sparsity problem. There are also highly advanced techniques with deep learning approaches such as state-of-the-art pre-trained word embeddings (e.g., Word2vec (Mikolov and Others, 2013), FastText (Bojanowski et al., 2016), and BERT (Devlin et al., 2018)) which have the purpose to handle sequential data with complex dependencies (e.g., phrases and multi-word expressions). The below followings will describe the pre-trained word embeddings that we aim to apply in our project.

- **Word2Vec**: Word2vec is one of the most popular state-of-the-art pre-trained word embeddings developed by Google. It was pre-trained on about 100 billion words from the Google News dataset. The Word2vec models are shallow, two-layer neural networks that take text as their input and provide a vector space with each unique word in the corpus being assigned a corresponding vector in the space as their input. Similar words locate near each other in the space. The output vectors, therefore, have the ability to capture the semantic and syntactic qualities of words. There are two ways to utilize Word2vec which are continuous bag-of-words (CBOW) and continuous skip-gram. In the continuous bag-of-words architecture, the model has the objective to predict the current word from the surrounding context words. In the continuous skip-gram architecture, the model aims to use the current word for predicting the surrounding context words. CBOW is faster while skip-gram does a better job for infrequent words. However, the Word2vec embeddings are context-independent because it maintains a look-up table that maps a single vector representation for each word. This means that they cannot capture the difference of the same word when being between two different sentences.

- **FastText**: FastText is a popular pre-trained word embedding technique developed by Facebook Research. It extends the concept of Word2Vec by representing words as n-gram character sequences, rather than individual words. This allows FastText to capture the morphological variations of words, making it particularly effective for handling out-of-vocabulary words and rare words. FastText uses a skip-gram architecture for training, similar to Word2Vec, but also includes additional information from the character n-grams. FastText embeddings are context-independent and can be used for a variety of natural language processing tasks, such as text classification, named entity recognition, and sentiment analysis. FastText is known for its efficiency and ability to handle large vocabularies, making it a popular choice for applications that require fast and accurate text representations.

- **BERT**: BERT is a recent pre-trained transformer-based technique for Natural Language Processing (NLP) developed by Google. It was pre-trained from BookCorpus (Zhu et al., 2015) and English Wikipedia[3] on two tasks which are masked language modeling (MLM) and next sentence prediction (NSP). BERT achieved state-of-the-art performance on several recent natural language understanding tasks (e.g., GLUE (Wang et al., 2018), SQuAD (Rajpurkar et al., 2016), and SWAG (Zellers et al., 2018)). Currently, there are numerous variants of BERT that are trained in variable ways for differing purposes (e.g., fewer computation resources and outstanding performance for specific tasks), such as, RoBERTa (Liu et al., 2019), ELECTRA (Clark et al., 2020), and DistilBERT (Sanh et al., 2019). In addition, there are several BERT-based models that are fine-tuned by developers in the Hugging Face community[4] for the specific domain and specific downstream tasks (e.g., Wav2vec 2.0 (Baevski et al., 2020) for speech representation task, BERTOverflow (Tabassum et al., 2020) for software development domain, and PEGASUS (Zhang et al., 2019) for news summarization task). However, unlike Word2vec, the BERT embeddings are context-dependent because they allow us to have multiple numerical vector representations for the same word, based on the context in which the word is used.

[3]https://en.wikipedia.org
[4]https://huggingface.co/

Handling Imbalanced Data Techniques

- **SMOTE**: Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al., 2002) is a widely used algorithm for dealing with imbalanced datasets in machine learning. The main goal of SMOTE is to generate synthetic samples of the minority class to balance the class distribution, thereby improving the performance of the classifier. The algorithm works by selecting a minority class sample and creating new synthetic samples by interpolating between that sample and its k nearest neighbors. This results in a dataset with an equal number of samples for each class, which can be used to train a classifier that is less biased towards the majority class. A simplified pseudocode for SMOTE algorithm is stated in Algorithm 1. SMOTE has been used in the software engineering domain, for example, the work (Khan, 2020) used SMOTE to oversample defective modules in their software defect prediction study.

- **VAE**: Variational Autoencoder (VAE) is an artificial neural network architecture introduced by Diederik P. Kingma and Max Welling (Kingma and Welling, 2014). VAE can learn the underlying distribution of input data and generate new samples that are similar to the original data. VAE consists of two components: an encoder that maps input data to a latent space and a decoder that maps samples from the latent space back to the original data space (refer to Figure 2.4). During training, VAE minimizes a loss function that consists of two terms: a reconstruction loss that measures how well the decoder can reconstruct the input data, and a regularization term that ensures the latent space follows a certain prior distribution. In the context of oversampling, VAE can be used to generate synthetic data points that are similar to the original data. By sampling from the learned latent space of VAE and decoding the samples back to the data space by its decoder, synthetic data points can be created that can be used to augment the original data. This can be useful in situations where the original data is imbalanced and additional data points are needed for training classifiers or other machine learning models.

Algorithm 1 : A simplified pseudocode for SMOTE algorithm

```
    Input: minority class samples $X_{min}$, number of synthetic samples to generate $k$
    Output: synthetic samples $X_{syn}$
 1: Compute $n_{neighbors}$ based on minority class size
 2: Compute distance between minority class samples and all other samples
 3: for each minority class sample $x_i$ in $X_{min}$ do
 4:     Find the $k$ nearest neighbors of $x_i$, excluding $x_i$ itself
 5:     for each nearest neighbor $x_j$ do
 6:         Compute the difference between the feature vectors of $x_i$ and $x_j$
 7:         Multiply the difference by a random number between 0 and 1
 8:         Add the result to the feature vector of $x_i$ to create a new synthetic sample $x_{new}$
 9:         Add $x_{new}$ to the synthetic samples $X_{syn}$
10:     end for
11: end for
12: return $X_{syn}$
```

One weakness of using SMOTE over VAE for oversampling is that it is a heuristic-based method that generates synthetic samples by interpolating between existing minority class samples, which may result in oversampling of noisy samples or outliers. VAE can generate synthetic samples that are more diverse and representative of the original data. Additionally, SMOTE may not be able to capture the underlying structure of the data and may lead to overfitting (Dai et al., 2019). On the other hand, VAE learns the underlying distribution of the data and can better capture the true underlying structure of the data. However, VAE requires a larger amount of data to train effectively and can be computationally expensive.

2.2 Literature Review

As we address three prediction outputs, the related work of this paper is mainly divided into three parts, including: pull request acceptance, pull request latency, and pull request reopening.

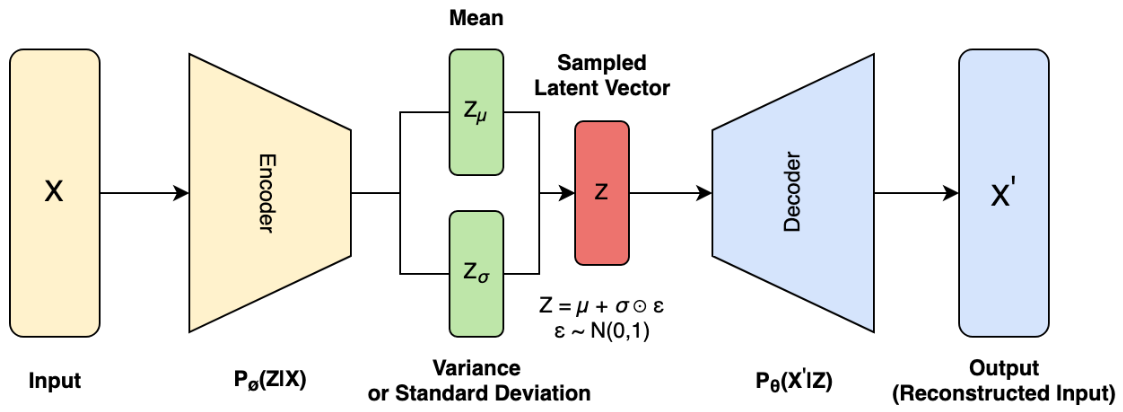


Figure 2.4: An architecture of VAE

2.2.1 Pull Request Acceptance

There are a number of previous works that focus on studying pull request evaluation, especially acceptance. Most of them leverage machine learning and statistical techniques to investigate factors influencing the pull request acceptance. The work in (Gousios et al., 2014) interpreted the relationship between characteristics (e.g., pull request, project, and contributor) and the acceptance through their best-performance classifier (i.e., Random Forest). They found that the decision to merge is mainly affected by whether the pull request modifies recently modified code. With the multidimensional association rule, the work in (Soares et al., 2015) shows that pull requests in projects written by different programming languages have different characteristics of merge decisions. For example, Java, JavaScript, and C++ have the merged probability reduced, whereas C#, C, and Scala increase the chances of the occurrence of a merge. Apart from the programming language, they found that number of commits, files added, external developer, and first pull request also affect the acceptance.

In addition to common sources of features as in stated works, social and technical factors were explored in (Tsay et al., 2014) through a multi-level mixed effects logistic regression model. Their main finding is that a pull request with test cases and less discussion submitted by strong social-connection contributors is more likely to be accepted. Moreover, the work in (Ortu et al., 2020) applied Logistic

Regression to observe that affect metrics (e.g., emotions, moods, and sentiment) of comment as well as experience and politeness of contributor influence the merge decision of an integrator because the model with these factors achieved higher performance than the one without them. Differing from other existing works, the work in (Zhang et al., 2021a) presented a comprehensive analysis of the factors influencing pull request decisions through statistical methods, such as, Spearman correlation coefficient, Cramér's V, and mixed-effects logistic regression models. Their list of factors was identified by conducting a systematic literature review.

Aside from the works focusing on the influencing factors, a few studies concentrate on building a high-quality predictive model. The work in (Nikhil Khadke, 2012) built a predictive model to tackle the pull request acceptance problem using Logistic Regression, Decision Trees, Random Forest, and Support Vector Machine along with a parameter tuning technique through learning common features. Their evaluation showed that the Random Forest-based model achieved the highest accuracy. Another work is (Chen et al., 2019) that derived new features to build the predictive model from crowdsourcing. From their experiment, the quantitative features extracted from dozens of recent papers outperformed the selected qualitative features they studied. The acceptance task, moreover, has been studied by (Gousios et al., 2014) and (Jiang et al., 2020). With their pull-request-based experiment, Gousios et al. (Gousios et al., 2014) also found that the Random Forest algorithm achieved the highest performance in the merging task. The work in (Jiang et al., 2020) proposed the project-based prediction approach, called CTCPPre, which is composed of an XGBoost classifier and four sources of features (i.e., code, text, contributor, and project). They, however, claimed that their approach outperformed the previous approach, by (Gousios et al., 2014), which employed Random Forest as their best classifier.

2.2.2 Pull Request Latency

Another aspect of pull request evaluation is latency. Most of the works related to this aspect introduce approaches to explore the influence of factors in the latency of pull requests in the GitHub projects and estimate the pull request lifetime by developing the prediction model. Apart from the study of pull request merging, Gousios et al. (Gousios et al., 2014) also leveraged the Random Forest model with a similar set of features to tackle the latency problem. The authors evaluated their models by discretizing the pull request lifetime into three classes (i.e., one hour, one day, and more than one day). They discovered that the time to merge is influenced by the developer's previous track record, the size of the project and its test coverage, and the project's openness to external contributions. Also, they found that the higher the contributor's merge percentage was, the lower would be the time taken for the pull requests to be integrated.

Other than treating the problem as a classification problem, variants of logistic regression were also used in (Yu et al., 2015) to model pull request latency on the GitHub open-source projects which employ a continuous integration service. The author performed comparisons between three linear regression approaches to identify that the contribution of their new features could make better performance than the previous work. The first model was solely built using the features listed in (Gousios et al., 2014). The second one included features associated with pull request review and integration processes. The last one was developed by adding features related to the use of continuous integration in the process. With the experiment, the third model achieved the best determination coefficient. The linear model was also used in the recent work by Zhang et al. (Zhang et al., 2021b). Their objective was to discuss the differences and variations in the influence of factors on pull request latency in varied scenarios and contexts using statistical modeling (e.g., the mixed-effect linear regression models). Their findings showed that the relative importance of the factors in different scenarios or varied contexts was distinctive. For example, the process-related factors were more important than the pull request description when the pull request was closed.

Another recent work by Júnior et al. (de Lima Júnior et al., 2021) handled this pull request evaluation problem with both the regression (i.e., estimating the numeric values of pull request lifetimes) and classification techniques. For the regression problem, they did project-specific experiments on four types of algorithms, including, the linear regression model, the regression trees model (i.e., M5Prime), the ensemble model (i.e., Random Forest), and the support vector machine model (i.e., SMOReg). In the classification task, they used the k-NN model (i.e., IBk), the decision tree model (i.e., J48), the Bayesian model (i.e., Naive Bayes), the ensemble model (i.e., Random Forest), and support vector machine (i.e., SMO). With their project-specific evaluation, they found that the linear model obtained the best performance in the regression task and Random Forest featured the best performance in the classification task.

2.2.3 Pull Request Reopening

To our best knowledge, there are only a few works related to reopening pull requests. Mohamed et al. (Mohamed et al., 2018) designed their approach named DTPre to predict the reopened pull requests after their first close using three sources of features, consisting of, code, review, and contributor. Due to an imbalanced dataset, the author dealt with it using an oversampling technique. In the evaluation phase, they ran the project-specific experiments on seven open-source GitHub projects through four various common classifiers and also compared the prediction performance with and without the oversampling technique. They found that Decision Tree with oversampling was the best approach. The recent work (Mohamed et al., 2020) by the same research team, Mohamed et al. extended their work by performing further cross-project experiments on the reopened pull request prediction through the same dataset. Their objective was to handle the cold-start problem for the new software project that has a limited number of pull requests. Reopening in pull requests was also studied in (Jiang et al., 2019). The author conducted a case study to understand reopened pull requests over seven popular projects in GitHub. Unlike the previous work that focused on prediction, they mainly aimed to investigate the impacts of reopened pull requests on code review, the root causes of a reopened pull request, and the differences between pull requests caused by different reasons. The main finding of this paper was that reopened pull requests had lower acceptance rates, longer evaluation time, and more comments than non-reopened ones.

2.2.4 Gaps in Literature

The existing literature on pull request evaluation has primarily focused on factors influencing acceptance and latency. The studies have examined various technical and social factors using traditional machine learning methods to build predictive models. However, there is limited research on the topic of pull request reopening, which can have a negative impact on software teams. Additionally, there is a lack of models that provide timely predictions for integrators immediately after a pull request is created. Another gap is that text data from the pull request description and the imbalanced nature of reopened pull requests have not been effectively handled. The relationship between pull request outputs, such as acceptance, latency, and reopening, also needs to be addressed. Furthermore, prior research on pull request prediction has typically evaluated models at either the project or all-in-one level, which can result in a cold-start problem for new projects or overly generalized models. Therefore, there is a gap in the literature in terms of using a programming language-specific experimental setting to balance specificity and generalization, which can improve the applicability and relevance of the models in real-world software development scenarios.

CHAPTER III

DATASET

In this chapter, we provide an overview of the dataset used in this study, including details on data collection, sampling, filtering, and labeling.

3.1 Overview of Dataset

We used GitHub data from well-known open-source projects developed under popular programming languages. To collect the data and build our dataset, GitHub REST APIs and a tool for web scraping were employed. Finally, we did data filtering to derive the final set of data, consisting of 143,886 pull requests (samples) from 54 open-source projects across four programming languages (i.e., Python, R, Java, and Ruby). The pull requests that we collected were created from Aug 2010 to Aug 2022. Table 3.1 illustrates our dataset including the overview of programming language characteristics and summarizes the statistical characteristics of the dataset for each language in terms of the number of pull requests per project in values of minimum, median, maximum, mean, and standard deviation. For example, in the Python language, there are 11 projects and 9,773 pull requests while the average number of pull requests per project is 888.45. As can be seen from the table, it seems that there are two groups of languages which are a small community and big community. Python and R are the small community programming language whereas the rest are the big ones.

3.2 Data Collection

Our data were collected from two sources which are the GitHub server and the GitHub Website (see Figure 3.1). To interact with the GitHub server (the left part), programs were developed using Python programming language. We used the GitHub REST APIs to access the data on the repositories of an open-source project written by popular programming languages and retrieve them. The whole

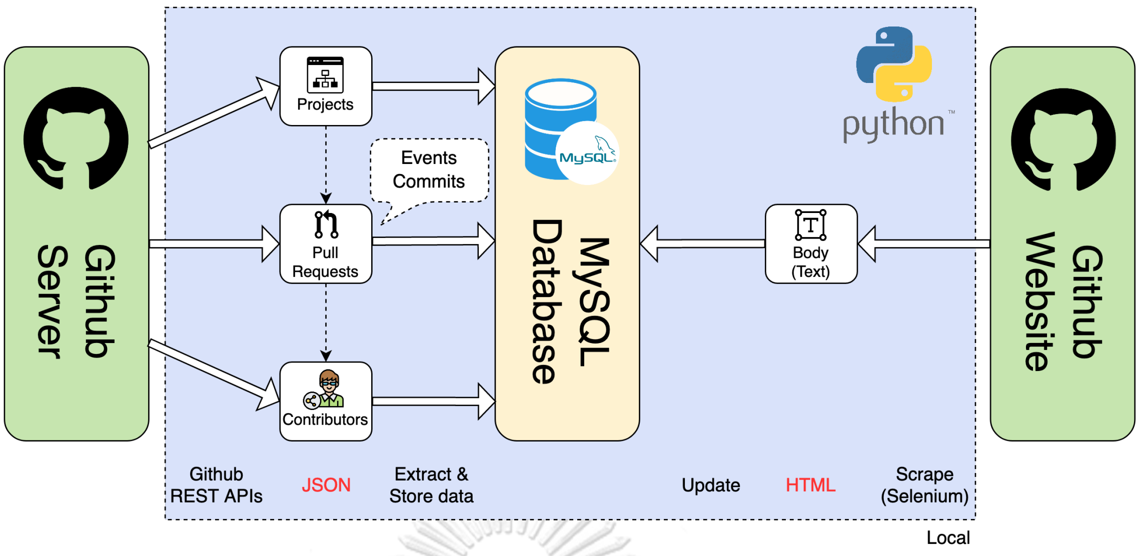


Figure 3.1: Overview of the data collection process

procedure started with sending a request and receiving a response in the form of JSON. We collected the required data of three information components consisting of project, pull request, and contributors. We then extracted only attributes of data that are necessary for our project, such as, pull request submission date, events in pull request, and commits in pull request. Since some data did not meet our requirements (e.g., documentation projects and pull requests without a body (i.e., no description)), we then had some processes to filter those out. Moreover, we considered only pull requests with closed status to at least ensure that they have been made the decision on. We will discuss more data filtering and sampling in the next section (3.3). The database that we used to store the data is MySQL. For the second part, GitHub Website (the right part), there were some data that we required to update (e.g., pull request body) so we developed the Selenium-based programs to scrape and update them on our database.

3.3 Data Sampling and Filtering

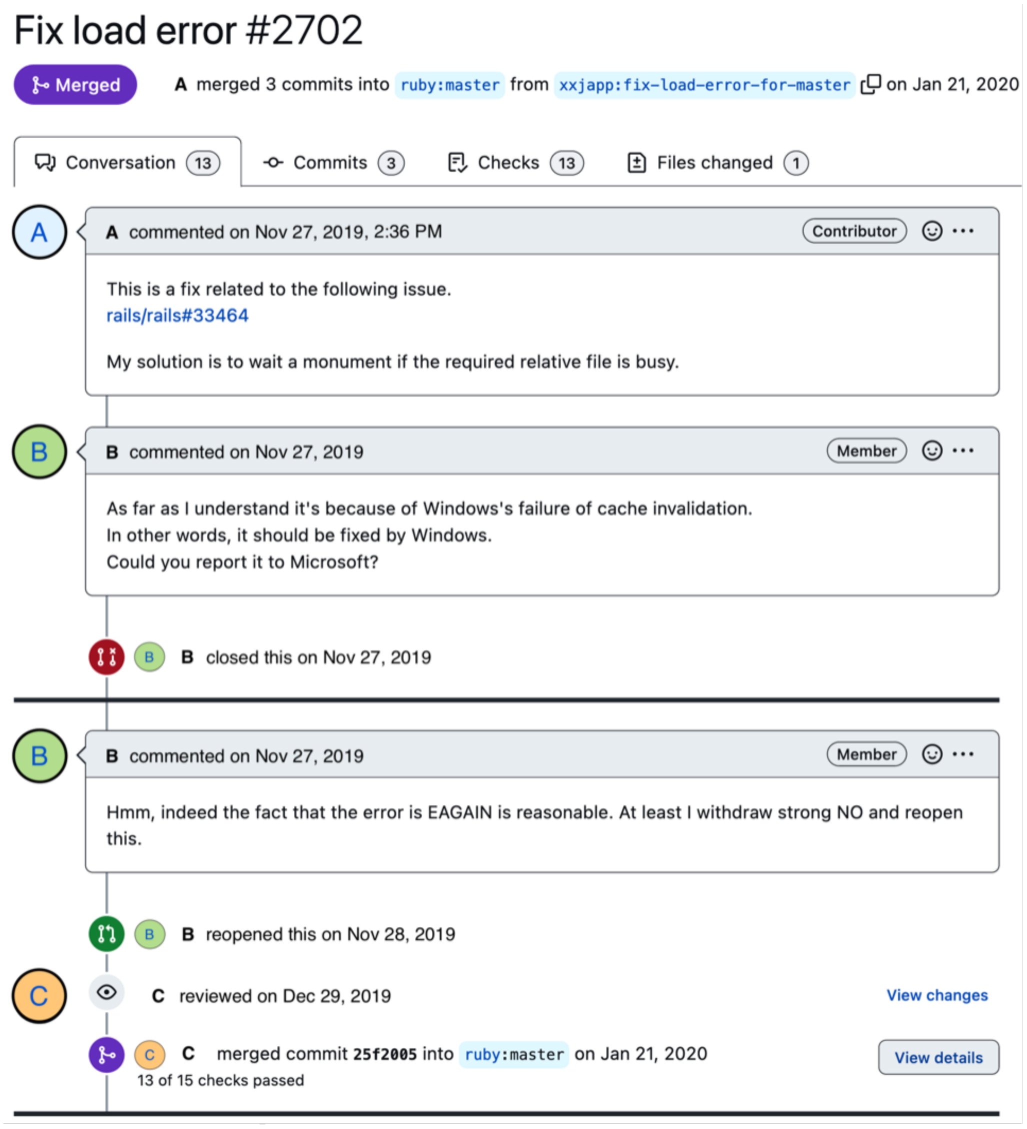
At first, we selected well-known programming languages which are Python, R, Java, and Ruby. We then collected 100 of the most starred open-source projects written by each programming language. Stargazer counts are routinely used by researchers as a proxy for project popularity (Papamichail et al., 2016). To ensure that our dataset was composed of relevant projects, we employed a second filter considering number of stars and other metrics which are number of open issues, fork status, number of forks, number of total commits, number of contributors, and number of pull requests. The projects that were included in our dataset required to meet the following programming language-specific criteria:

- not a fork version of another project,
- not a documentation project,
- and more than or equal to the median of number of open issues, number of total commits, number of contributors, and number of pull requests.

However, we were aware of bias caused by the projects with overcrowding pull requests. A model will be overfitting with this kind of project only. Therefore, we decided to not include these projects in our dataset.

3.4 Data Labeling

To illustrate our contributions, Figure 3.2 shows an example of the pull request ID 2702 in the Ruby project written by the Ruby programming language. Due to the purpose of the demonstration, the figure has been edited and we mainly show some important parts of the pull request. The title and body indicate that the contributor (Mr.A) would like to propose his changes to fix the issue related to the load error. At first, the integrator (Mr.B) from the core team decided to reject his pull request because it sounds unrelated to Ruby. At the next time, the integrator changed his mind because some parts are reasonable to associate with Ruby. The integrator, therefore, reopened the pull request and let another integrator (Mr.C) review it. With the changes reviewed, the integrator Mr.C changed the decision made by the integrator Mr.B and accepted this pull request. As an example, there are always reopening risks when the integrator's initial decision is impaired due to various factors, for ex-



Fix load error #2702

Merged A merged 3 commits into ruby:master from xxjapp:fix-load-error-for-master on Jan 21, 2020

Conversation 13 Commits 3 Checks 13 Files changed 1

A commented on Nov 27, 2019, 2:36 PM Contributor

This is a fix related to the following issue.
rails/rails#33464

My solution is to wait a monument if the required relative file is busy.

B commented on Nov 27, 2019 Member

As far as I understand it's because of Windows's failure of cache invalidation.
In other words, it should be fixed by Windows.
Could you report it to Microsoft?

B closed this on Nov 27, 2019

B commented on Nov 27, 2019 Member

Hmm, indeed the fact that the error is EAGAIN is reasonable. At least I withdraw strong NO and reopen this.

B reopened this on Nov 28, 2019

C reviewed on Dec 29, 2019 View changes

C merged commit 25f2005 into ruby:master on Jan 21, 2020 View details
13 of 15 checks passed

Figure 3.2: An example of a pull request on GitHub

ample, misunderstanding the description and change of the pull request. Therefore, the integrators' decision would be more effective if they could early recognize the risks involved in the pull requests, they are responsible for.

To formulate our predictive problem, we denote t_{pred} as the reference point to the pull request submission time at which a prediction is made for a pull request (i.e., prediction time). **We would like to develop a classification approach that can predict three outputs: 1) acceptance, 2) latency, and 3) reopening for a pull request at time t_{pred}.** To be more specific, the prediction outcomes would be made using only information available at t_{pred}.

1) **Acceptance**: this reflects the decision of an integrator on whether a pull request is accepted or rejected in the final close. There are two nominal classes for acceptance which are:
   - Accepted: a pull request is accepted to merge into the main branch
   - Rejected: a pull request is rejected to merge into the main branch.
2) **Latency**: this reflects the time difference between the pull request submission and the final close (i.e., lifetime). We employ the way to discretize the lifetime from (de Lima Júnior et al., 2021) where the magnitude is maintained (i.e., minutes, hours, days, weeks, or months). We classify latency into five ordinal classes which are:
   - Hour: a pull request that has lifetime less than or equal to 60 mins
   - Day: a pull request that has lifetime greater than 60 mins but less than or equal to 24 hours
   - Week: a pull request that has lifetime greater than 24 hours but less than or equal to 7 days
   - Month: a pull request that has lifetime greater than 7 days but less than or equal to 4 weeks
   - GTMonth: a pull request that has lifetime greater than 4 weeks (greater than 1 month).
3) **Reopening**: this reflects the reopening status of a pull request showing whether it has been reopened. The reopening task can be consider a problem of anomaly detection due to the highly imbalanced nature of the data. In this context, the number of instances in the positive class (i.e., pull requests that are likely to be reopened) is much smaller than the number of instances in the negative class (i.e., pull requests that are likely to be closed). This makes the task of accurately identifying positive instances more challenging, as the

model needs to effectively distinguish between a small number of anomalies and a large number of normal instances. There are two nominal classes for the reopening output which are:

- Reopened: a pull request has been reopened for one time or more than
- NonReopened: a pull request has never been reopened.

Table 3.2 shows the class distribution for the acceptance, latency, and reopening outputs. Following the same notations given in Section 2.1.2, our classification task is to learn a predictive function, $f : X \rightarrow Y$, from the dataset $D = \{x_j, y_j\}_{j=1}^{M}$ that can predict an output vector, $f(x) \in Y$, for unseen input x where we can represent our input (i.e., pull request features) as $X = \mathbb{R}^d$ and our output as $Y = Y^1 \times Y^2 \times Y^3$ in which Y^1 denotes the Acceptance output, Y^2 denotes the Latency output, and Y^3 denotes the Reopening output. Y^1 has two possible classes (i.e., C_1^1 and C_2^1) while Y^2 has five possible classes (i.e., C_1^2, C_2^2, C_3^2, C_4^2, and C_5^2) and Y^3 has two possible classes (i.e., C_1^3 and C_2^3).

Table 3.1: Descriptive statistical information of our pull request dataset

Language	Overview		# Pull Requests / Project				
	# Projects	# Pull Requests	Min	Med	Max	Mean	SD
Python	11	9,773	173.00	764.00	2,574.00	888.45	693.12
R	12	8,310	150.00	456.00	1,706.00	755.45	557.19
Java	12	29,202	504.00	1,247.00	6,636.00	2,433.50	2,055.49
Ruby	19	96,601	662.00	3,760.00	13,431.00	5,084.26	3,864.15
TOTAL	54	143,886					

Table 3.2: The number of samples (pull requests) in each class for the acceptance, latency, and reopening outcomes

Language	Acceptance		Latency					Reopening	
	Accepted	Rejected	Hour	Day	Week	Month	GTMonth	Reopened	NonReopened
Python	7,061	2,712	1,428	3,023	2,541	1,392	1,389	188	9,585
R	6,858	1,452	1,224	2,585	1,808	1,275	1,418	85	8,225
Java	12,644	16,558	5,512	9,401	7,546	3,500	3,243	700	28,502
Ruby	72,853	23,748	17,839	35,062	22,871	10,848	9,981	1,414	95,187

In the case of the pull request ID 2702, t_{pred} is 2:36 PM, 27 Nov 2019. The information available at 2:36 PM, 27 Nov 2019 is used to predict three outcomes of this pull request which are *Accepted* for the acceptance output, *GTMonth* for the latency output, and *Reopened* for the reopening output (see Figure 3.3). From the fact that reopening always occurs before pull request evaluation and may affect the other outcomes, the model will predict the reopening output first and use it as a feature to predict the other outputs.

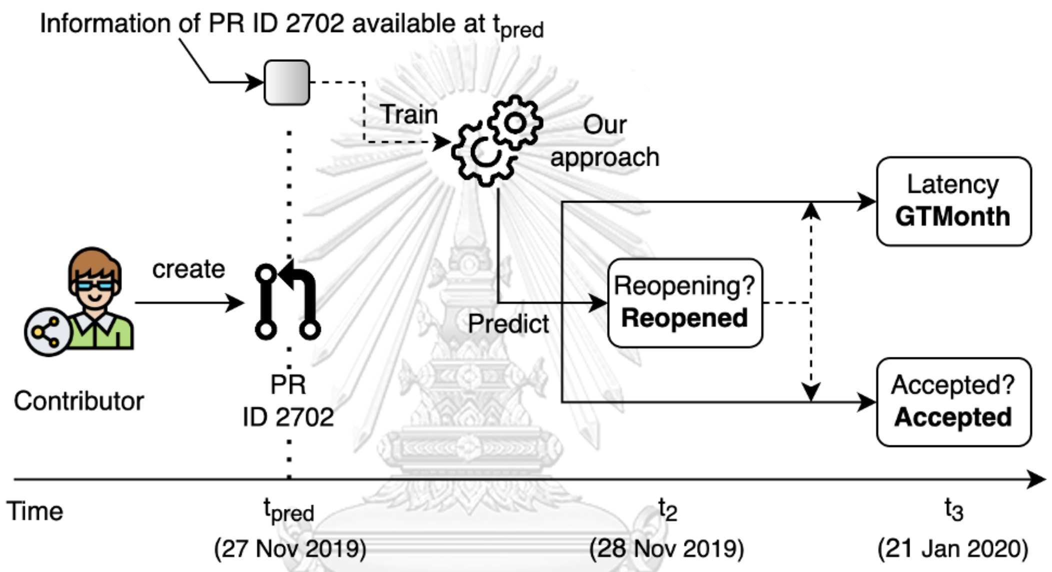


Figure 3.3: A user scenario for the pull request ID 2702

CHAPTER IV

PROPOSED APPROACH

This chapter presents the proposed approach for predicting pull request evaluation and reopening. Our approach combines tabular and textual features, employs oversampling techniques, utilizes shared learning, and incorporates deep learning classifiers. Within this chapter, we provide a comprehensive overview of the framework of our approach, including feature extraction and model architecture.

4.1 Overview Framework

Our proposed approach is a deep learning-based classification approach for predicting acceptance, latency, and reopening of pull requests. Figure 4.1 shows an overview of our approach, which is divided into two phases: the training phase and the execution phase. The training phase involves using historical pull requests to build predictive models. To extract features, we categorize the information of a pull request into two groups: *tabular data* (represented in blue color) and *textual data* (represented in green color). Tabular data is structured data that can be extracted using common feature extraction techniques, resulting in numerical or categorical features. Textual data is unstructured data that require advanced learning techniques to extract meaningful features. Then, oversampling is performed to handle the imbalanced data in the reopening task. Our approach utilizes both types of data (X) along with their corresponding outcomes (Y) to train predictive models using deep learning techniques. The execution phase involves employing the trained models from the training phase to predict three outcomes: acceptance, latency, and reopening, for a new pull request. From the fact that reopening always occurs before pull request evaluation and may affect the other outcomes, our approach will predict the reopening output first and use it as a feature to predict the other outputs.

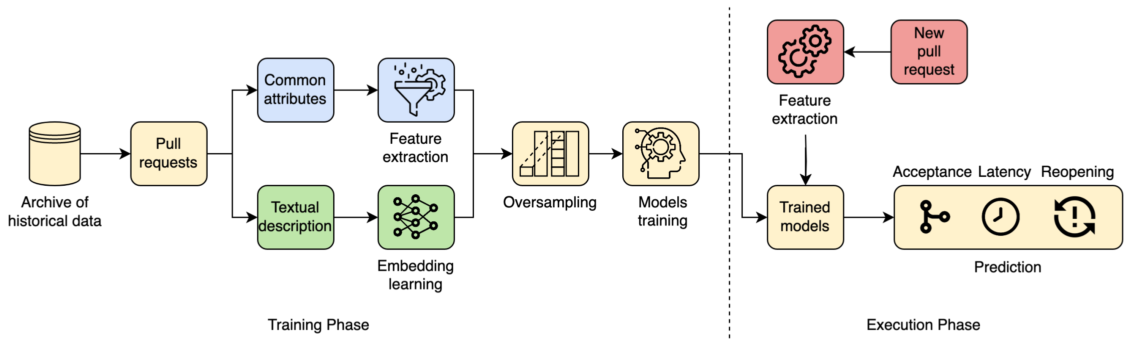


Figure 4.1: An overview framework of our proposed approach

4.2 Feature Extraction

There are two types of feature extraction which are tabular feature extraction and textual feature extraction.

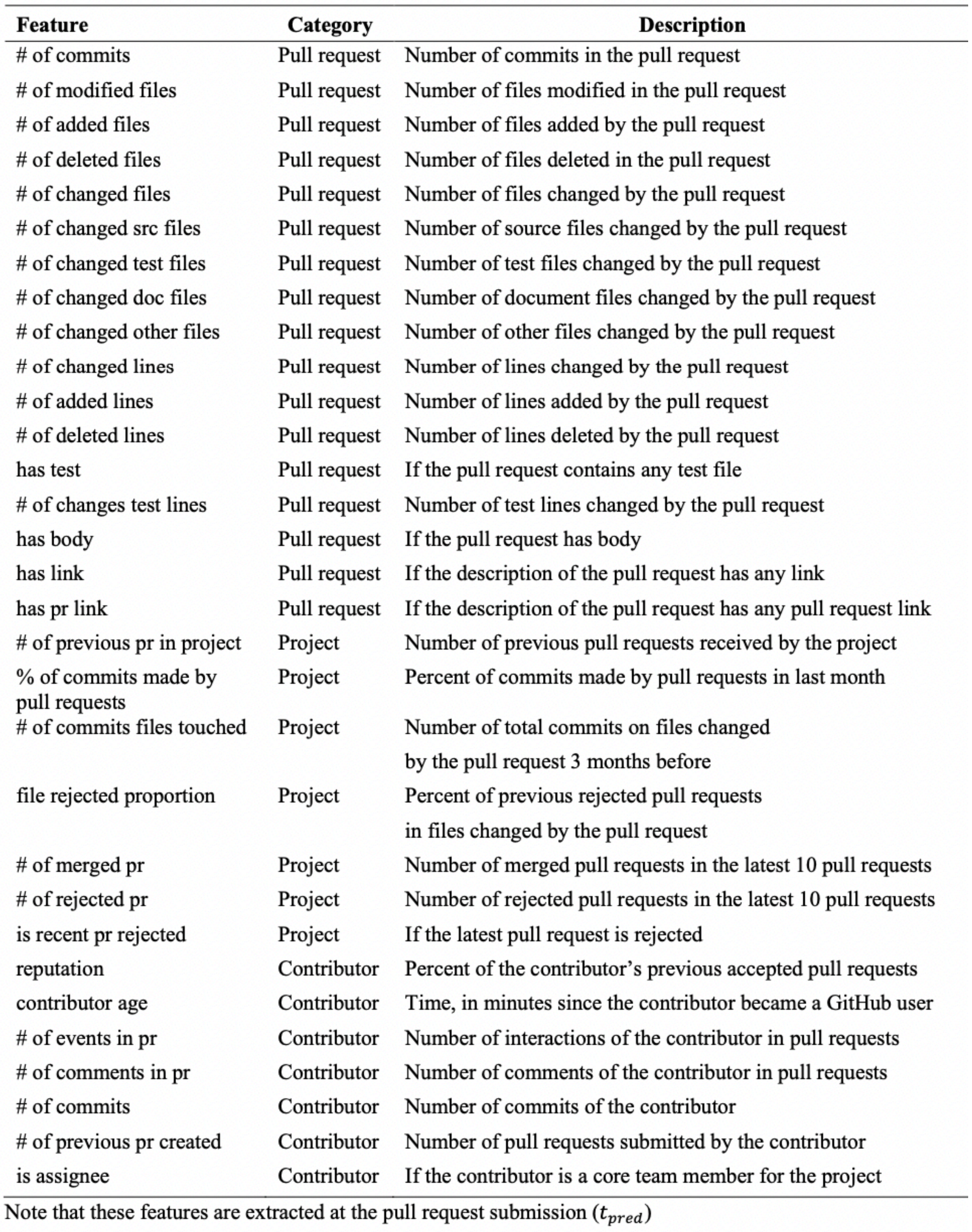
4.2.1 Tabular Feature Extraction

A project repository and a pull request contain many attributes that we can extract and use as a feature to characterize the pull request. To do feature extraction, we define three characteristics describing a pull request including pull request, project, and contributor. Our tabular features are derived from the set of the most frequently used features by the previous works related to the prediction (Gousios et al., 2014; Jiang et al., 2020; Mohamed et al., 2018, 2020; de Lima Júnior et al., 2021). It is important to note that the features we extract for our analysis are obtained at the time of pull request submission (t_{pred}). As a result, certain features related to discussion, such as the number of comments and the number of participants, may not be available during this stage. Table 4.1 provides a comprehensive list of the extracted features, along with their descriptions. Common feature extraction techniques are deployed such as counting, summation, subtraction, and ratio calculations based on the attributes of the pull request.

4.2.2 Textual Feature Extraction

A pull request usually contains two textual pieces of information which are title and body. Contributors use them to summarize and describe the proposed changes.

Table 4.1: List of our tabular features used in our approach



Feature	Category	Description
# of commits	Pull request	Number of commits in the pull request
# of modified files	Pull request	Number of files modified in the pull request
# of added files	Pull request	Number of files added by the pull request
# of deleted files	Pull request	Number of files deleted in the pull request
# of changed files	Pull request	Number of files changed by the pull request
# of changed src files	Pull request	Number of source files changed by the pull request
# of changed test files	Pull request	Number of test files changed by the pull request
# of changed doc files	Pull request	Number of document files changed by the pull request
# of changed other files	Pull request	Number of other files changed by the pull request
# of changed lines	Pull request	Number of lines changed by the pull request
# of added lines	Pull request	Number of lines added by the pull request
# of deleted lines	Pull request	Number of lines deleted by the pull request
has test	Pull request	If the pull request contains any test file
# of changes test lines	Pull request	Number of test lines changed by the pull request
has body	Pull request	If the pull request has body
has link	Pull request	If the description of the pull request has any link
has pr link	Pull request	If the description of the pull request has any pull request link
# of previous pr in project	Project	Number of previous pull requests received by the project
% of commits made by pull requests	Project	Percent of commits made by pull requests in last month
# of commits files touched	Project	Number of total commits on files changed by the pull request 3 months before
file rejected proportion	Project	Percent of previous rejected pull requests in files changed by the pull request
# of merged pr	Project	Number of merged pull requests in the latest 10 pull requests
# of rejected pr	Project	Number of rejected pull requests in the latest 10 pull requests
is recent pr rejected	Project	If the latest pull request is rejected
reputation	Contributor	Percent of the contributor's previous accepted pull requests
contributor age	Contributor	Time, in minutes since the contributor became a GitHub user
# of events in pr	Contributor	Number of interactions of the contributor in pull requests
# of comments in pr	Contributor	Number of comments of the contributor in pull requests
# of commits	Contributor	Number of commits of the contributor
# of previous pr created	Contributor	Number of pull requests submitted by the contributor
is assignee	Contributor	If the contributor is a core team member for the project

Note that these features are extracted at the pull request submission (t_{pred})

For example, the pull request ID 2702 in Figure 3.2 has "Fix load error" as the title and "This is a fix related to the following issue. rails/rails#33464 My solution is to wait a monument if the required relative file is busy" as the body. Title and body can reflect the nature of a pull request (e.g., detail of a review task and complexity of the task). Therefore, a good title and body can reduce the integrator's effort to execute the review task. However, they haven't been taken seriously to use as a pull request predictor once. We, thus, use the text as one of our features to characterize our pull request. To do so, specific text feature extraction is required. We employ various *state-of-the-art pre-trained word embeddings* (e.g., Word2vec, FastText, BERT, and BERTOverflow) to tackle this task (see Figure 4.2). The average pooling is used to derive a final vector representation for the entire text. It is important to note that in order to increase the importance of the title, we assign an additional weight to the title compared to the body (i.e., Text = Title + Title + Body).

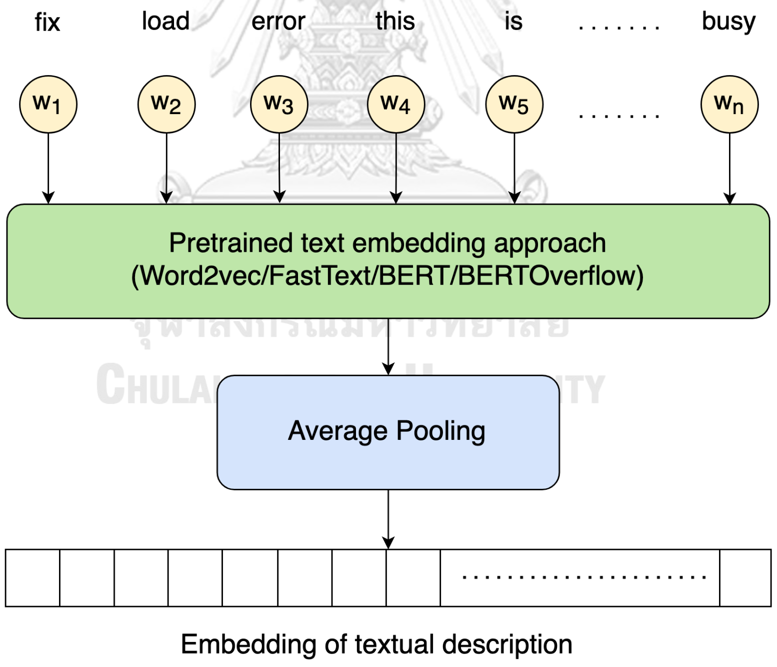

fix
load
error
this
is
....
busy
w1
w2
w3
w4
w5
....
wn
Pretrained text embedding approach (Word2vec/FastText/BERT/BERTOverflow)
Average Pooling
....
Embedding of textual description

Figure 4.2: An architecture of feature extraction for textual description

4.3 Model Architecture

To simulate the real situation and address the relationship between pull request reopening and evaluation, we separate modeling into two main stages: the reopening stage and the evaluation stage. More precisely, the reopening output is predicted first, and it is used as one of the features to predict the pull request evaluation.

4.3.1 Pull Request Reopening Stage

In the reopening stage, we follow a three-stage process of feature extraction, oversampling, and classification (see Figure 4.3). We begin by combining tabular features extracted from common attributes and textual features learned by pre-trained word embedding, as detailed in the previous section (Section 4.2.2). However, since the data is highly imbalanced, with a majority of non-reopening samples and a minority of reopening samples, we use the *Variational Autoencoder (VAE)* to generate additional reopening samples in order to achieve balance with the non-reopening samples. The architecture of our VAE can be found in Section 2.1.2.

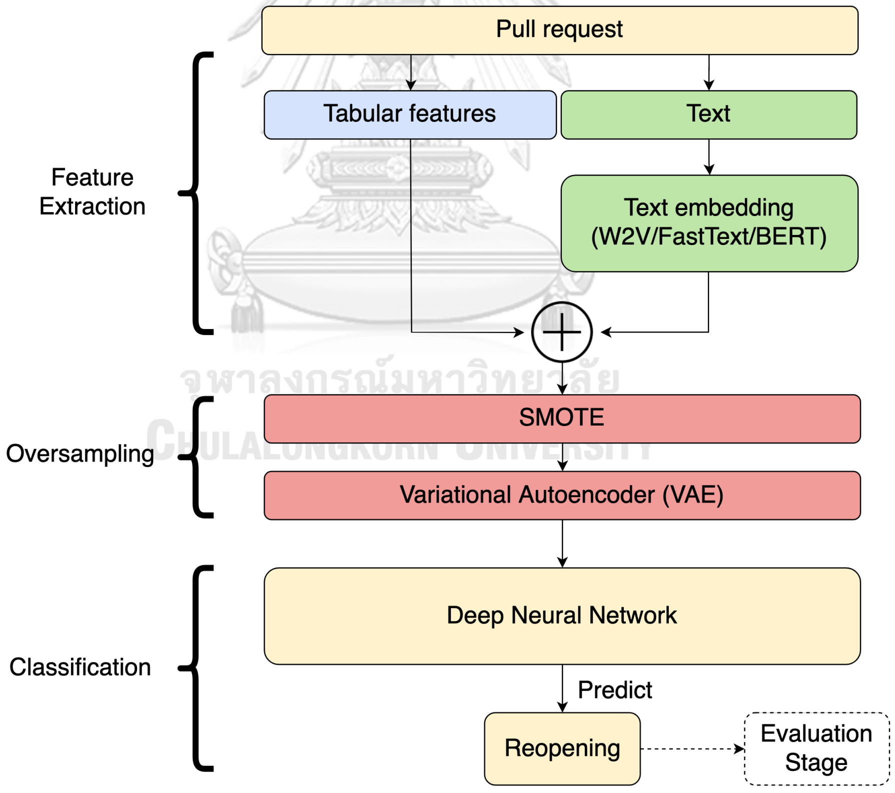


Figure 4.3: A model architecture of the reopening stage of our approach

Figure 4.4 shows the processes that we went through to perform oversampling on the reopening samples using SMOTE and VAE. VAE is a generative model that can learn to approximate a probability distribution of input data by encoding them into a lower-dimensional latent space and then decoding them back to the original data space. By sampling from the learned latent space, VAE can generate new data points that are similar to the original data, effectively increasing the size of the dataset. To ensure that we have enough reopening samples for training the VAE, we first use the *Synthetic Minority Over-sampling Technique (SMOTE)* to oversample the positive class (i.e., *Reopened* class). Next, we train our VAE exclusively on pull requests with the *Reopened* class. We use the decoder part of the trained VAE to generate reopening samples by introducing random noise from a normal distribution. These generated samples are first combined with the original ones and then shuffled to create a training dataset for a *deep neural network (DNN)*. The DNN is trained using this shuffled dataset to predict the probability of reopening as the output. Note that we only balance the number of samples in the training set. To conclude, there are two models that we need to train in this stage, which are the VAE for oversampling and the DNN for the reopening classification.

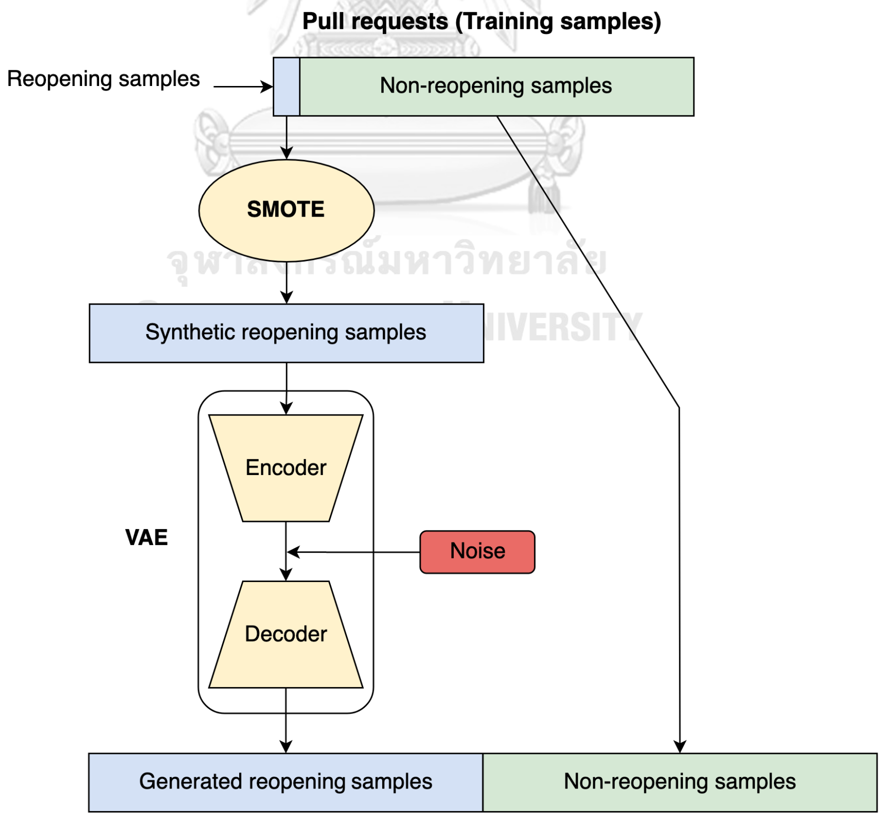

Pull requests (Training samples)
Reopening samples
Non-reopening samples
SMOTE
Synthetic reopening samples
VAE
Encoder
Noise
Decoder
Generated reopening samples
Non-reopening samples

Figure 4.4: An oversampling process through SMOTE and VAE for the reopening samples

4.3.2 Pull Request Evaluation Stage

During the evaluation stage, we extract features from the pull request description and other relevant data, similar to the feature extraction process used in the reopening stage (without oversampling). We combine these features with the reopening probability obtained from the reopening stage. A deep neural network is then trained to predict two outputs: acceptance and latency of the pull request. The approach used in the evaluation stage is depicted in Figure 4.5.

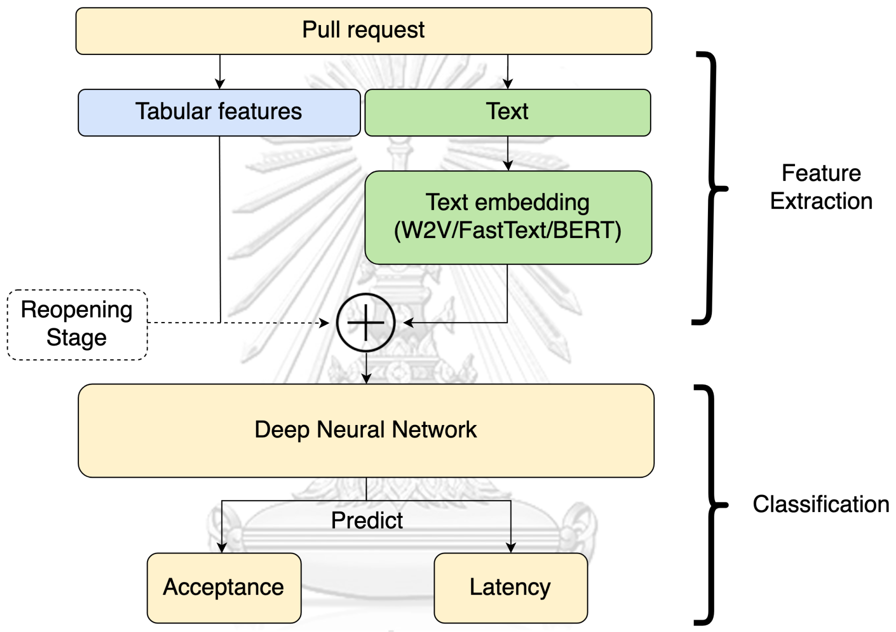


Figure 4.5: A model architecture of the evaluation stage of our approach

The architecture utilized for the deep neural networks is a very common feed-forward neural network, consisting of an input layer, a normalization layer, followed by multiple blocks of dense layers and dropout layers, and an output layer (refer to Figure 4.6). The normalization layer helps to scale the input features to a standard range, making it easier for the neural network to process them effectively. The dense layers are used to learn complex representations of the input data, while the dropout layers help to prevent overfitting by randomly dropping out some of the neurons during training. This architecture is used for downstream prediction in both stages. In the evaluation stage, there is only one model, which is the *multi-output DNN* for predicting the acceptance and latency outputs. Specifically, the DNN in the evaluation stage shares input features, hidden layers, and weights, and loss while the output layer has 2 output nodes for acceptance and latency.

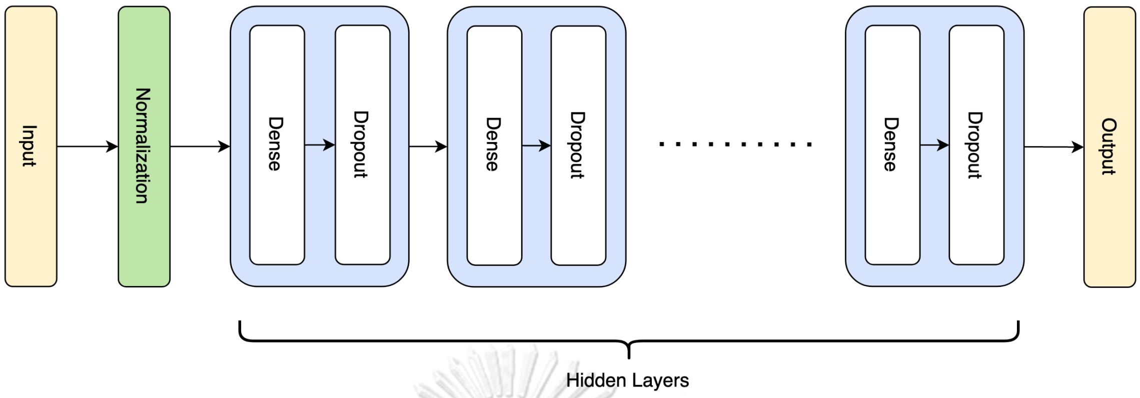


Figure 4.6: An architecture of a deep neural network for downstream tasks

CHAPTER V

IMPLEMENTATION

This chapter focuses on the implementation of the proposed approach for predicting pull request evaluation and reopening. Overview of implementation processes, data preparation process, and modeling process are stated along with examples of source code.

5.1 Overview of implementation

The implementation for this project is divided into two phases: *data preparation* and *modeling*. The data preparation phase comprises four stages, namely data collection, data extraction, data filtering, and feature and label construction. On the other hand, the modeling phase also includes four stages, namely data splitting, data preprocessing, model training, and model evaluation (see Figure 5.1). The Python 3.10.2 environment was used for all the processes, and the implementation was carried out on a Macbook Pro with macOS Monterey Version 12.4, an Apple M1 Pro chip with 8-core CPU and 14-core GPU, 16GB RAM, and 1TB SSD storage.

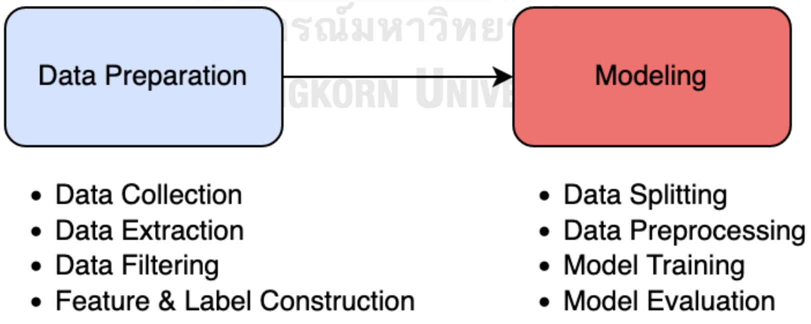


Figure 5.1: Overview of implementation

5.2 Data Preparation

The primary procedure in this phase involves collecting and processing data in order to prepare the dataset for modeling purposes. Our data were collected from two sources: the GitHub server and the GitHub website. Fortunately, GitHub provides GitHub REST APIs[5] to allow users to interact with the server. We used the Python library named *requests* to create and send an HTTP request to the server and receive a response back in JSON format. The data were extracted and stored in our *MySQL* database, offering ease of reading and writing the data. An example of interacting with the GitHub server is shown in Figure 5.2. This code was used to collect a list of 100 project repositories with the highest number of stargazers among a specific programming language. An example of the JSON response from the GitHub server is demonstrated in Figure 5.3. The response shows the information of the Django Debug Toolbar project written in Python. Moreover, all APIs used are stated in Table 5.1, which will be used from now on when referring to those APIs. In this study, there are three information sources that we need to collect for modeling: project repository, pull request, and contributor.

5.2.1 Project Repository

We began collecting a list of 100 project repositories with the highest stargazers for each programming language using the first API, as shown in the API table. We then looped over the list to collect their information using the second API and saved them as a JSON file. After that, a set of workable data was extracted and stored in our database using the Python library named *pymysql*. The extracted data included id, name, full name, URL, creation date, stargazer count, and other social attributes.

5.2.2 Pull Request

With the projects collected, we used the third API to retrieve a list of pull requests under each project. We then looped over the list to collect their information using the fourth API. We also applied the fifth and seventh APIs to collect a list of commits and events under each pull request, respectively. After that, we used the sixth and eighth APIs to gather the details of the commits and events. We extracted a

[5]https://docs.github.com/en/rest

```
import requests
def prepare_proj_request(database, token, page):
    url = "https://api.github.com/search/repositories"
    headers = CaseInsensitiveDict()
    headers["Accept"] = "application/vnd.github.v3+json"
    headers["Authorization"] = "token {}".format(token)
    params = {
        "q": "language:{}".format(database),
        "sort": "stars",
        "order": "desc",
        "per_page": "100",
        "page": str(page)
    }
    return url, headers, params

def send_request(url, token=None, headers=None, params=None):
    while True:
        print("\nsend request to {} ...".format(url))
        try:
            resp = requests.get(url, headers=headers, params=
    params, timeout=30 )
            resp_json = resp.json()
        except:
            print("\nrequest failed, retrying ...")
            if is_connected_to_internet():
                continue
            time.sleep(60)
    return resp_json
```

Figure 5.2: An example of interacting with the GitHub server using *requests*

set of workable data and stored it in our database. The data extracted from the pull request information consisted of id, pull request number, URL, state, title, body, creation date, latest update date, close date, merge date, contributor id, contributor name, and others. For the commit information, the extracted data included sha, URL, committer name, committer id, commit date, changed file count, number of changes, number of added lines, number of deleted lines, and others. The event information included id, creator id, creator name, creation date, event, and others. At this point, we also performed data filtering (see Section 3.3) on both the project and pull request data in order to ensure relevance and quality of our dataset.

Since our approach predicts pull request evaluation and reopening at the time of submission, we needed to update the data back to the submission time. Some attributes were updated using a log of events, such as the title, while the body could

```
{
    "id": 46939,
    "node_id": "MDEwOlJlcG9zaXRvcnk0NjkzOQ==",
    "name": "django-debug-toolbar",
    "full_name": "jazzband/django-debug-toolbar",
    "owner": {
        "login": "xxxxxxxx",
        "id": 9999999,
        ...
    },
    "html_url": "https://github.com/jazzband/django-debug-
   toolbar",
    "description": "A configurable set of panels that display
   various debug information about the current request/
   response.",
    "fork": false,
    "url": "https://api.github.com/repos/jazzband/django-debug
   -toolbar",
    ...
    "size": 7418,
    "stargazers_count": 7097,
    "language": "Python",
    "open_issues": 93,
    ...
    "network_count": 979,
    "subscribers_count": 107
}
```

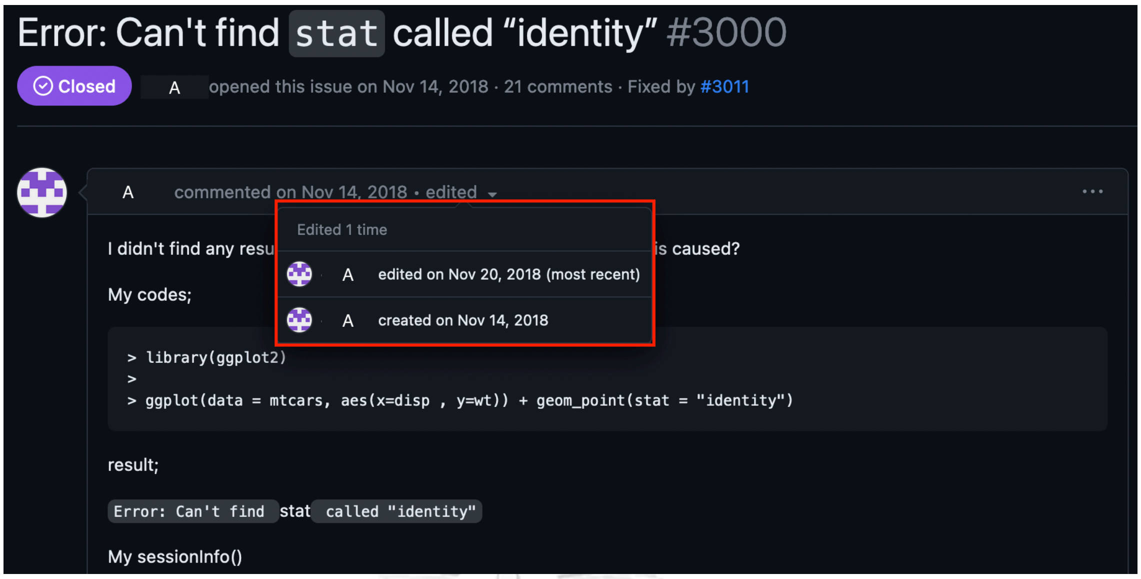
Figure 5.3: An example of the JSON response from the GitHub server

not be updated. Fortunately, the GitHub website has a list of change logs for pull request bodies, as shown in Figure 5.4. Therefore, we used *selenium* to scrape the body originally created at the submission time. After scraping, we update them on the database. Some parts of the Python script in which we used *selenium* to scrape and update the pull request bodies are demonstrated in Figure 5.5.

5.2.3 Contributor

Once we had collected the pull request data and extracted the contributor IDs, we used the ninth API to retrieve detailed information about each contributor. We extracted workable attributes such as ID, name, URL, and creation date, and stored this data in our database.

After obtaining all the relevant information from our data sources (i.e., project


Error: Can't find stat called "identity" #3000

Closed A opened this issue on Nov 14, 2018 · 21 comments · Fixed by #3011

A commented on Nov 14, 2018 · edited

Edited 1 time

A edited on Nov 20, 2018 (most recent)

A created on Nov 14, 2018

I didn't find any resu… …is caused?

My codes;

```
> library(ggplot2)
>
> ggplot(data = mtcars, aes(x=disp , y=wt)) + geom_point(stat = "identity")
```

result;

`Error: Can't find stat called "identity"`

My sessionInfo()

Figure 5.4: An example of change logs of pull request body

repository, pull request, and contributor), we needed to transform them into a set of tabular features that we could use for modeling. These features are summarized in Table 4.1. To derive these features, we used a combination of data manipulation and calculation. For example, to calculate the number of commits in a pull request, we simply counted the number of commits associated with that pull request. Similarly, to compute the number of merged pull requests, we counted the number of previously merged pull requests in the latest 10 pull requests. In addition, we computed the age of a contributor by subtracting their creation date from the date the pull request was submitted. Aside from the tabular features, we also had two textual features which are title and description that required further data preprocessing in the modeling stage. It is essential to note that all information that we used to form the features were extracted at the pull request submission (t_{pred}). The final step in the data preparation phase was to create labels for pull requests. Each pull request was given three labels: the *acceptance* label, the *latency* label, and the *reopening* label, as described in Section 3.4. The output of this phase was the dataset that was ready for the modeling phase.

Table 5.1: List of all used the GitHub REST APIs

	GitHub REST APIs	**Purpose**
1	https://api.github.com/search/repositories	Collect a list of project reprositories
2	https://api.github.com/repos/<project-full_name>	Collect a project reprository information
3	https://api.github.com/repos/<project-full_name>/pulls	Collect a list of pull requests under the project reprository
4	https://api.github.com/repos/<project-full_name>/pulls/<pull_request-number>	Collect a pull request information
5	https://api.github.com/repos/<project-full_name>/pulls/<pull_request-number>/commits	Collect a list of commits under the pull request
6	https://api.github.com/repos/<project-full_name>/commits/<commit-sha>	Collect a commit information
7	https://api.github.com/repos/<project-full_name>/issues/<pull_request-number>/events	Collect a list of events under the pull request
8	https://api.github.com/repos/<project-full_name>/issues/events/<event-id>	Collect a event information
9	https://api.github.com/users/<contributor-login>	Collect a contributor information

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
current_path = os.getcwd()
web_driver_file = os.path.join(current_path, "geckodriver")
driver = webdriver.Firefox(executable_path=web_driver_file)
pull_id = row['id']
html_url = row['html_url']
driver.get(html_url)
body_div = WebDriverWait(driver, 5).until(EC.presence_of_element_located((By.ID, "pullrequest-{}".format(pull_id))))
body_status = ""
try:
    WebDriverWait(body_div, 1).until(EC.element_to_be_clickable((By.CSS_SELECTOR, "svg.octicon-triangle-down:nth-child(2)"))).click()
except:
    body_status = "no update"
if body_status == "":
    WebDriverWait(body_div, 10).until(EC.element_to_be_clickable((By.XPATH, "(//ul[contains(@class,'lh-condensed text-small')]/li)[last()]"))).click()
    try:
        body = WebDriverWait(driver, 20).until(EC.presence_of_element_located((By.CSS_SELECTOR, ".markdown-body.entry-content.comment-body.p-0"))).get_attribute('innerHTML')
        body_status = "updated"
    except:
        body_status = "deleted"
"""
update 'body' on database
"""
```

Figure 5.5: Some parts of the Selenium-based script used to scrape the pull request bodies

5.3 Modeling

The modeling phase consists of four stages which are data splitting, data preprocessing, model training, and model evaluation.

5.3.1 Data Splitting

We began with loading the dataset from our database as a CSV file. To ensure that our model learned only from past data available during training, we sorted the pull requests in the dataset by their closing date. In particular, pull requests in the training and validation sets were closed before those in the testing set. Furthermore, the pull requests in the training set were terminated before those in the validation set. Next, we performed data splitting using a 60/20/20 split for small programming languages and a 80/10/10 split for big programming languages. Once we completed the data splitting, we saved the train, validation, and test sets separately into three CSV files. This was done to facilitate further processing and analysis of each set independently.

5.3.2 Data Preprocessing

The data preprocessing stage involves three sub-steps. The first sub-step is preprocessing for tabular features, which involves standardizing the tabular data to a format suitable for model training. The second sub-step is preprocessing for textual features, which includes text cleaning, tokenization, and vectorization of textual data. The third sub-step is preprocessing for labels, which involves encoding the labels in a suitable format for use in the model training phase. This includes label encoding or one-hot encoding, depending on the type of labels.

Tabular Features

We encountered two types of tabular features: numerical features and categorical features. For numerical features (e.g., # of commits, # of modified files, and contributor age), we performed standardization to normalize the data and bring it to a common scale. We employed *StandardScaler* from the *sklearn* library to handle this task. However, for categorical features (e.g., has test, hast body, and has link), we only had binary features with values of 0 or 1, so no preprocessing was necessary.

```
import csv
data_path = "dataset/{}.csv".format(language)
# train/valid/test
# small
if language in ["python", "r"]:
    split_ratio = (60,20,20)
# large
else:
    split_ratio = (80,10,10)

# read the data
with open(data_path, 'r') as f:
    reader = csv.reader((line.replace('\0', '') for line in f)
   )
    csv_list = list(reader)
    header = csv_list[0]
    data = csv_list[1:]

# split the data
train_len = len(data) * split_ratio[0] // 100
valid_len = len(data) * split_ratio[1] // 100
test_len = len(data) * split_ratio[2] // 100
remainder = len(data) - train_len - valid_len - test_len
if remainder > 0:
    train_len += remainder
train_data = data[:train_len]
valid_data = data[train_len:train_len + valid_len]
test_data = data[train_len + valid_len:]

# write the data
with open("dataset/{}_train.csv".format(language), 'w') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerows(train_data)
with open("dataset/{}_valid.csv".format(language), 'w') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerows(valid_data)
with open("dataset/{}_test.csv".format(language), 'w') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerows(test_data)
```

Figure 5.6: A Python script for data spliting

Textual Features

Initially, we created text data by concatenating two titles and one body. Next, we transformed the text data to lower case and performed cleaning. The cleaning process involved removing any code snippets, HTML tags, stop words, andpunctuation marks. We also replaced multiple spaces with a single space and performed tokenization to break the text into individual tokens. For most of processes, the *nltk* library was applied. Figure 5.7 the piece of code used to preprocess the text.

After the text was cleaned and tokenized, it was passed through pre-trained embedding models to learn low-dimensional semantic representation vectors of the text, without fine-tuning. If the learned vectors were already available, we used them directly instead of feeding the text data into the model for re-embedding. It is important to note that if a vector was not available for a specific token, we used a vector of zeros instead. After that, the average pooling technique is employed to derive a final vector representation for the whole text. As mentioned earlier, we considered several pre-trained models, including, Word2vec, FastText, and BERTs (e.g., $\text{BERT}_{\text{BASE\_UNCASED}}$ and BERTOverflow). Word2vec and FastText were implemented through the Python library, named *gensim* while BERTs were developed via the *transformers* library. The example of learning embeddings for text is shown in Figure 5.8. The output dimension that we used for BERTs was 768 while Word2vec was 200 and FastText was 300.

Labels

We used different methods to preprocess each of the labels. Specifically, one-hot encoding was applied for encoding the classified acceptance (i.e., 0 for *Rejected* and 1 for *Accepted*) and the classified reopening (i.e., 0 for *NonReopened* and 1 for *Reopened*) while label encoding was used for the classified latency (i.e., 0 for *Hour*, 1 for *Day*, 2 for *Week*, 3 for *Month*, and 4 for *GTMonth*) because the latency contains the values that have an ordinal relationship between each other.

5.3.3 Model Training

This phase was related to training deep learning-based models, including VAE and deep neural networks. With training and validation dataset prepared, the tabular features and textual features were concatenated into a single vector for each sample.

```
import nltk
from nltk.corpus import stopwords

tokenizer = nltk.tokenize.ToktokTokenizer()
punctuations = string.punctuation + string.digits + ’”” + ”’”
stop_words = set(nltk.corpus.stopwords.words(’english’))
alphabet_string = string.ascii_lowercase
alphabet_list = list(alphabet_string)
def strip_list_noempty(lst):
    new_list = (item.strip() if hasattr(item, ’strip’) else item for item in lst)
    return [item for item in new_list if item != ’’]

def clean_punctuation(text):
    tokens = tokenizer.tokenize(text)
    punctuation_filtered = []
    regex = re.compile(’[%s]’ % re.escape(punctuations))
    for token in tokens:
        punctuation_filtered.append(regex.sub(’’, token))
    filtered_list = strip_list_noempty(punctuation_filtered)
    return ’ ’.join(map(str, filtered_list))

def clean_text(text):
    # lower
    text = text.lower()
    # use single space to replace multiple spaces
    text = re.sub(re.compile(’[\n\r\t]’), ’ ’, text)
    # remove ’!”#$%&\’()*+,-./:;<=>?@[\\]^_‘{|}~’
    text = clean_punctuation(text)
    # replace any sequence of 2 or more spaces with a single space
    text = re.sub(re.compile(’\s{2,}’), ’ ’, text)
    # remove stop words and remove single character
    text = ’ ’.join([token for token in tokenizer.tokenize(text) if token not in stop_words and token not in alphabet_list])
    return text

for df in [train_df, valid_df, test_df]:
    # create text column by concatenating title and body but if body is empty or null, just use title
    df[’text’] = df.apply(lambda row: row[’title’] + ’ ’ + row[’title’] if row[’body’] == ’’ or pd.isnull(row[’body’]) else row[’title’] + ’ ’ + row[’title’] + ’ ’ + row[’body’], axis=1)
    df[’text’] = df[’text’].apply(lambda x: re.sub(r’<code>(.*?)</code>’, ’’, x, flags=re.MULTILINE|re.DOTALL))
    df[’text’] = df[’text’].apply(lambda x: re.sub(re.compile(’<.*?>’), ’’, x))
    df[’text’] = df[’text’].apply(lambda x: clean_text(x))
```

Figure 5.7: A piece of Python code used to preprocess the text

```
from gensim.models.keyedvectors import KeyedVectors
pretrained_model = "word2vec_models/SO_vectors_200.bin"
word_vect = KeyedVectors.load_word2vec_format(pretrained_model
    , binary=True)
def embed_text(text):
    tokens = text.split()
    if text == '':
        return np.zeros(default_dim)
    vectors = []
    for token in tokens:
        try:
            vectors.append(word_vect[token])
        except:
            vectors.append(np.zeros(default_dim))
    vectors = np.array(vectors)
    vectors = np.mean(vectors, axis=0)
    return vectors
for df in [train_df, valid_df, test_df]:
    df['text'] = df['text'].apply(lambda x: embed_text(x))
```

Figure 5.8: A piece of Python code used to represent the text by learned vectors from Word2vec

Variational Autoencoder (VAE)

The aim of using VAE was to handle highly imbalanced data in reopening output by increasing the number of positive samples (i.e., reopening samples). To train a VAE model, we began by performing oversampling on the positive samples in the training dataset using *SMOTE* from the *imblearn* library (see Figure 5.9). This was done to ensure that we had a sufficient number of positive samples to train the model. The VAE model was trained using *keras* and hyperparameter-tuned using *keras_tuner* with 30-iterations randomized algorithm (refer to Figure 5.10). The model was trained and validated using early stopping technique, a learning rate scheduler, and model checkpoint. The loss function for the VAE model was defined by combining the reconstruction loss and KL loss (refer to Figure 5.11). The validation set's loss was used as the stopping criterion during the tuning process.

After training and fine-tuning the VAE model, the best model was selected and saved to disk. The decoder layer of the best model was used to generate new postive samples by randomly sampling from the latent space of the VAE model. The generated samples were concatenated with the original training data to create a new

```
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=SEED, sampling_strategy='auto')
X_train_resamp, y_train_resamp = sm.fit_resample(X_train,
    y_train[label])
print("shape of X_train_resamp: {}".format(X_train_resamp.
    shape))
print("shape of y_train_resamp: {}".format(y_train_resamp.
    shape))

# filter only positive samples
X_train_1_resamp = X_train_resamp[y_train_resamp == 1]
```

Figure 5.9: A piece of Python code used to upsample the reopening samples using SMOTE

dataset with balanced classes, where the positive class had been upsampled using the best VAE model. Finally, the new dataset was shuffled randomly to ensure that the order of the data does not affect the training of the classification model for the reopening task. It is important to note that the newly generated training dataset was exclusively used for training the classification model for the reopening task and was not used for training the acceptance and latency task models.

Deep Neural Networks

We implemented two deep neural network-based classification models - one for the reopening output and another for the acceptance and latency output. Although both models were quite similar, they differed in the training dataset used and their outputs. The classifier for the reopening output was trained using the generated training set, which was obtained through a combination of the original training set and the samples generated by the VAE model. On the other hand, the classifier for the acceptance and latency output was trained using the original training set only. Specifically, the second model was a multi-output classifier that predicted both the acceptance and latency outputs. Both models were implemented using the *keras* library, and we used *keras_tuner* to perform hyperparameter tuning with a 30-iteration randomized algorithm utilizing the validation set. The hyperparameters we tuned included various aspects of the model architecture, such as, whether

```
class VaeModel(kt.HyperModel):
    def __init__(self, input_dim):
        self.input_dim = input_dim

    def build(self, hp):
        set_seeds(SEED)
        encoder_inputs = Input(shape=(self.input_dim,))
        x = BatchNormalization()(x)
        x = Dropout(dropout_rate)(x)
        x = Dense(layer_2, activation=act_2)(x)
        x = BatchNormalization()(x)
        x = Dropout(dropout_rate)(x)
        x = Dense(layer_3, activation=act_3)(x)
        x = BatchNormalization()(x)
        x = Dropout(dropout_rate)(x)
        z_mean = Dense(latent_dim, name="z_mean")(x)
        z_log_var = Dense(latent_dim, name="z_log_var")(x)
        z = Sampling()([z_mean, z_log_var])
        encoder = Model(encoder_inputs, [z_mean, z_log_var, z
    ], name="encoder")
        latent_inputs = Input(shape=(latent_dim,))
        x = Dense(layer_3, activation=act_3)(latent_inputs)
        x = BatchNormalization()(x)
        x = Dropout(dropout_rate)(x)
        x = Dense(layer_2, activation=act_2)(x)
        x = BatchNormalization()(x)
        x = Dropout(dropout_rate)(x)
        x = Dense(layer_1, activation=act_1)(x)
        x = BatchNormalization()(x)
        x = Dropout(dropout_rate)(x)
        decoder_outputs = Dense(self.input_dim, activation="
    linear")(x)
        decoder = Model(latent_inputs, decoder_outputs, name="
    decoder")
        outputs = decoder(encoder(encoder_inputs)[2])
        vae = Model(encoder_inputs, outputs, name="vae")
        vae.compile(optimizer=Adam(hp.Choice('learning_rate',
    [1e-2, 1e-3, 1e-4])),
                    metrics=[tf.keras.metrics.MeanSquaredError
    ()])

        return vae
```

Figure 5.10: A piece of Python code used for training and tuning an VAE model

to include batch normalization layers, the number of dense layers, the number of hidden nodes in each dense layer, and the dropout percentage.

Figure 5.12 shows a piece of code that we used to balance the class weights

```
reconstruction_loss = losses.mean_squared_error(encoder_inputs
    , outputs)
reconstruction_loss *= self.input_dim
kl_loss = 1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var
    )
kl_loss = tf.reduce_mean(kl_loss) * -0.5
vae_loss = reconstruction_loss + kl_loss
vae.add_loss(vae_loss)
```

Figure 5.11: A code for defining loss for an VAE model

for the outputs. We also employed model checkpoint and learning rate scheduler techniques to improve the training process. In terms of architecture, both models comprised an input layer, a batch normalization layer, a dense layer with the ReLU activation function, a dropout layer, and an output layer (see Figure 5.13 and Figure 5.14). For the reopening and acceptance outputs, we used binary cross-entropy as the loss function, while for the latency output, we used categorical cross-entropy. However, we did shared loss learning for the second classifier by giving an equal weight to each output in order to balance their importance during the training process. This helped to ensure that the model was not overly biased towards one output over the other. By equally weighting the acceptance and latency outputs, we ensured that the model gave each output the same level of attention and consideration during the training process. This resulted in a more balanced and accurate model that was capable of accurately predicting both acceptance and latency outputs. To select the best model, we employed the validation set's AUC for the reopening and acceptance outputs, while we used MMAE for the latency output. After obtaining the best models, we dumped them into the disk to prepare for model evaluation with the test set.

5.3.4 Model Evaluation

As we stated in the previous section, we used different evaluation metrics for each output to choose the best-performing approach. For the reopening and acceptance outputs, we employed the validation set's AUC. For the latency output, we used MMAE, After training, we selected the models with the highest AUC or low-

```
class_weights = compute_class_weight(
    class_weight = "balanced",
    classes = np.unique(y_train[label]),
    y = y_train[label]
)
class_weights = dict(zip(np.unique(y_train[label]),
   class_weights))
print("class_weights: {}".format(class_weights))

model_checkpoint = tf.keras.callbacks.ModelCheckpoint(
    filepath='our_model/{}/best_weight_{}_{}'.format(language,
   label, text_type), monitor='val_loss', verbose=0,
   save_best_only=True,
    save_weights_only=True, mode='auto', save_freq='epoch',
   options=None
)
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss', factor=0.2, patience=5, verbose=0,
   mode='auto',
    min_delta=0.0001, cooldown=0, min_lr=0
)
```

Figure 5.12: A piece of code for defining class weights, model checkpoint and learning rate scheduler for a classifier for predicting the reopening output

est MMAE on the validation set. Specifically, this process started by choosing the pre-trained embedding model (e.g., Word2vec, FastText, BERT, and BERTOverflow) for both pull request reopening and evaluation classifiers. The selection of the best pre-trained model was based on its performance on the validation set, measured by AUC for the reopening output and acceptance output, and MMAE for the latency output. The best-performing pre-trained model was then chosen for each classifier, and they were not necessarily the same. For example, we could have BERT for the reopening classifier but Word2vec for the evaluation classifier, based on their respective evaluation performances.

Ultimately, we performed the final evaluation of our best approach on the test set, which was crucial in assessing the model's effectiveness in real-world scenarios. We evaluated the models based on various metrics, including accuracy, precision, recall, f1-score, and AUC for the acceptance output, while MMAE was used for the latency output. For the reopening output, we used metrics such as Balanced Accuracy, AUC, TPR, and FPR. Most of the metrics were available in the *sklearn*

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 801)]             0

 normalization (Normalization)  (None, 801)            1605
 dense (Dense)               (None, 32)                25664

 re_lu (ReLU)                (None, 32)                0

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 16)                528

 re_lu_1 (ReLU)              (None, 16)                0

 dropout_1 (Dropout)         (None, 16)                0

 dense_2 (Dense)             (None, 1)                 17

 classification_head_1 (Acti  (None, 1)                0
 vation)
=================================================================
Total params: 26,209
Trainable params: 26,209
Non-trainable params: 0
_________________________________________________________________
```

Figure 5.13: An example of a model architecture of the pull request reopening classifier

library while MMAE was from the *imblearn* library. Additionally, we compared the model's performance with a randomized baseline and the existing approach to understand its effectiveness. This comparative analysis helped us identify the strengths and weaknesses of our approach and the scope for further improvements. We will discuss the evaluation and results in the next chapter.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 802)]             0

 normalization (Normalization)  (None, 802)            1605

 dense (Dense)               (None, 32)                25696

 re_lu (ReLU)                (None, 32)                0

 dense_1 (Dense)             (None, 64)                2112

 re_lu_1 (ReLU)              (None, 64)                0

 dense_2 (Dense)             (None, 32)                2080

 re_lu_2 (ReLU)              (None, 32)                0

 dropout (Dropout)           (None, 32)                0

 dense_3 (Dense)             (None, 1)                 33

 dense_4 (Dense)             (None, 5)                 165

 classification_head_1 (Activat  (None, 1)             0
 ion)
 classification_head_2 (Softmax  (None, 5)             0
 )
=================================================================
Total params: 31,691
Trainable params: 30,086
Non-trainable params: 1,605
_________________________________________________________________
```

Figure 5.14: An example of a model architecture of the pull request evaluation classifier

CHAPTER VI

EVALUATION AND RESULTS

In this chapter, we cover the research questions, experimental setting, performance measures, experimental results, further discussion of the findings, and threats to validity.

6.1 Research Questions

In this study, our empirical evaluation aims to answer the following research questions.

RQ1: Sanity Check (Is the proposed approach suitable to predict the pull request evaluation and pull request reopening?)

This aims to perform a sanity check on the suitability of our proposed approach in predicting pull request evaluation and reopening at the submission time. To address this, we compare the performance of our approach against a *randomized algorithm*. Conducting a sanity check using a rule-based model is a common practice in software engineering research (Al-Zubaidi et al., 2018; Shepperd and MacDonell, 2012; Sarro et al., 2016). We repeat the random guessing process 5000 times and take the average performance to ensure statistical significance. Our approach should surpass the baseline, which relies on random guessing, to demonstrate its suitability for the early prediction of pull request evaluation and pull request reopening.

RQ2: Does the proposed approach outperform the existing approach?

The objective of this research question is to compare the predictive performance between the *existing approach* and our approach. This aims to measure the feasibility of 3 components, according to research objectives, that encourage the predictive performance of pull request acceptance, latency, and reopening. This allows us to evaluate whether:

- The designed architecture of our approach is proper to capture the relationships that allows shared learning for predicting pull request acceptance, latency, and reopening.
- Our proposed oversampling technique is proper for handling the highly imbalanced nature of the reopening pull requests.
- Our proposed texture features extracted by employed feature extraction techniques can compensate the limit of available information of early prediction.

Currently, there exists no approach that predicts in the same manner as our proposed approach. Specifically, our approach predicts three pull request outputs (i.e., acceptance, latency, and reopening) at the time of pull request submission. As such, we will use the approach that employs tabular features, feature selection, and traditional machine learning-based single-output classifiers, such as, *Decision Tree*, *Random Forest*, and *XGBoost*, which have been reported as the best classifiers in previous studies (Gousios et al., 2014; Jiang et al., 2020; Mohamed et al., 2018, 2020; de Lima Júnior et al., 2021), to represent the existing approaches. Particularly, each outcome will have a separate classifier. The performance of our approach should be better than the existing approach to indicate that the combination of tabular features and textual features extracted from the pre-trained model, our oversampling technique (i.e., SMOTE combined with VAE), shared learning, and deep learning classifier can overcome the challenges, posed by the limited information available at the time of submission and highly imbalanced data, as well as improve the performance for the prediction of pull request evaluation and pull request reopening.

6.2 Experimental Setting

We used a *hold-out technique* to split data into three sets including a training set, validation set, and testing set. We sorted pull requests based on their close date to mimic deployment in a real situation. This was to make sure that our model will learn from only the past data available at the training time. In detail, the pull requests in the training set and the validation set were closed before the pull requests in the testing set, and the pull requests in the training set were also terminated before the pull requests in the validation set. Our experiment was programming language specific, so we trained an approach for each language. The programming language-specific experimental setting balances specificity and generalization. It avoids the cold-start problem for new projects and overly generalized models, which has been a limitation in prior research that evaluated models at the project or all-in-one level. The small community programming languages used a 60/20/20 split while the big ones used an 80/10/10 split. In our study, the training dataset was used to train our model and we used the validation dataset to choose the best techniques and the best models as well as to tune hyperparameters. Lastly, the testing dataset was applied to evaluate the performance of our model.

To ensure a fair comparison of performance, the approaches were trained and validated on the same experimental environment using the identical dataset with the shared data splitting. The set of tabular features were also shared between both the existing approach and our approach. Moreover, both approaches employed the same tuning hyperparameter technique, which involved using a randomized algorithm with 30 iterations and the shared random seed. Also, the classifiers in both approaches were trained with balanced class weights to avoid bias towards the majority class. Additionally, we used the same performance metrics to select the best model and hyperparameter set. Specifically, we used AUC for the acceptance and reopening output, and MMAE for the latency output.

6.3 Performance Measures

We used common binary classification metrics, such as, Accuracy, Precision, Recall, F1-Score, and AUC for the acceptance task. However, we applied Macro-Averaged Absolute Error (MMAE) for the latency task because it can assess the distance between an actual class and a predicted one. It can also be applied to the imbalanced multi-class classification because of the macro-averaged technique. MMAE has been used in many works (Baccianella et al., 2009; Choetkiertikul et al., 2018; Wattanakriengkrai et al., 2019) to tackle the ordinal multi-class classification problem. Note that for the MMAE metric, lower values indicate better performance. In addition, we used metrics that can handle highly imbalanced data and enable accurate anomaly detection, such as, Balanced Accuracy (BA), AUC, True Positive Rate (TPR), and False Negative Rate (FPR) (Kale et al., 2022; Trauer et al., 2021) for the reopening task (see Section 2.1.2 for details of each measure).

6.4 Experimental Results

In this section, we report the evaluation results to answer the research questions. Table 6.1 and Table 6.2 show the evaluation results for pull request evaluation and reopening achieved by randomized baseline, existing approach, and our approach in four programming languages. The evaluation metrics for acceptance include Accuracy (Acc), Precision (P), Recall (R), F1-Score (F1), and area under the receiver operating characteristic curve (AUC). The evaluation metric for latency is Macro-Averaged Mean Absolute Error (MMAE) while the evaluation metrics for reopening include Balanced Accuracy (BA), AUC, True Positive Rate (TPR), and False Positive Rate (FPR). In addition, we also report the results with percentage improvement (Percent Imp) and the performance scores on average.

6.4.1 Results for RQ1

For the pull request evaluation, the analysis of all measures (i.e., Accuracy, Precision, Recall, F1-Score, AUC, and MMAE) shown in Table 6.1 suggests that the predictive results obtained with our approach (Our), are better than those achieved by using the randomized baseline (Randomized) in all cases (24/24) consistently. Our approach improves between 43.74% (in Python) to 65.21% (in Java) in terms of Accuracy, 4.71% (in Ruby) to 60.44% (in Java) in terms of Precision, 48.31% (in Python) to 65.17% (in Java) in terms of Recall, 31.57% (in R) to 62.83% (in Java) in terms of F1-Score, 29.75% (in R) to 81.85% (in Java) in terms of AUC, and 24.79% (in R) to 28.84% (in Java) in terms of MMAE over the baseline.

For the pull request reopening task, our approach outperforms the randomized baseline in most cases (14/16) in terms of Balanced Accuracy, AUC, TPR, and FPR. Our approach improves over the baseline between 17.92% (in Java) to 28.21% (in Ruby) in terms of Balanced Accuracy, 24.71% (in Java) to 46.39% (in Python) in terms of AUC, and 27.15% (in Ruby) to 52.75% (in R) in terms of TPR, while the results for FPR are mixed, with some cases showing improvement and others showing a decline compared to the baseline. Specifically, our approach improves FPR by 19.96% (in Python) and 29.27% (in Ruby) over the baseline, while it is unable to improve in R and Java. Our approach achieves the best performance in Ruby, as it consistently outperforms the baseline in all evaluation measures.

On average, our approach has significantly better performance than the baselines in all measures. For the pull request acceptance task, our approach obtains 0.762 Accuracy, 0.878 Precision, 0.791 Recall, 0.832 F1-Score, and 0.749 AUC, while for the latency task, it gains 1.163 MMAE. For the pull request reopening task, our approach achieves 0.618 Balanced Accuracy, 0.689 AUC, 0.694 TPR, and 0.458 FPR, compared to the baselines which achieve 0.500 Accuracy, 0.753 Precision, 0.500 Recall, 0.595 F1-Score, and 0.500 AUC for the acceptance task, and 1.600 MMAE for the latency task while it gains 0.500 Balanced Accuracy, 0.500 AUC, 0.500 TPR, and 0.500 FPR for the reopening task.

Our proposed approach outperforms the randomized baseline in all four programming languages, thus our approach is suitable for predicting pull request evaluation and reopening at the submission time.

Table 6.1: Performance comparison between our approach (Our) and the randomized baseline (Randomized) for predicting the pull request evaluation and reopening, reported as Accuracy (Acc), Precision (P), Recall (R), F1-Score (F1), AUC, Macro-Averaged Absolute Error (MMAE), Balanced Accuracy (BA), True Positive Rate (TPR), and False Positive Rate (FPR) along with percentage improvement (Percent Imp)

Language	**Approach**	**Acceptance**					**Latency**	**Reopening**			
		Acc	**P**	**R**	**F1**	**AUC**	**MMAE**	**BA**	**AUC**	**TPR**	**FPR**
Python	Randomized	0.500	0.771	0.500	0.607	0.500	1.599	0.500	0.500	0.500	0.500
	Our	**0.719**	**0.874**	**0.742**	**0.803**	**0.716**	**1.171**	**0.639**	**0.732**	**0.679**	**0.400**
	Percent Imp	43.74	13.40	48.31	32.30	43.18	26.80	27.81	46.39	35.66	19.96
R	Randomized	0.500	0.835	0.500	0.625	0.500	1.600	0.500	0.500	0.501	**0.500**
	Our	**0.721**	**0.874**	**0.777**	**0.823**	**0.649**	**1.204**	**0.600**	**0.715**	**0.765**	0.564
	Percent Imp	44.11	04.75	55.38	31.57	29.75	24.79	20.00	42.94	52.75	-12.79
Java	Randomized	0.500	0.523	0.500	0.511	0.500	1.600	0.500	0.500	0.501	**0.500**
	Our	**0.826**	**0.839**	**0.826**	**0.832**	**0.909**	**1.139**	**0.590**	**0.624**	**0.700**	0.515
	Percent Imp	65.21	60.44	65.17	62.83	81.85	28.84	17.92	24.71	38.87	-03.06
Ruby	Randomized	0.500	0.883	0.500	0.638	0.500	1.600	0.500	0.500	0.500	0.500
	Our	**0.781**	**0.925**	**0.819**	**0.868**	**0.720**	**1.141**	**0.641**	**0.684**	**0.635**	**0.354**
	Percent Imp	56.15	04.71	63.70	36.00	44.01	28.69	28.21	36.90	27.15	29.27
AVERAGE	Randomized	0.500	0.753	0.500	0.595	0.500	1.600	0.500	0.500	0.500	0.500
	Our	**0.762**	**0.878**	**0.791**	**0.832**	**0.749**	**1.163**	**0.618**	**0.689**	**0.694**	**0.458**
	Percent Imp	52.30	20.83	58.14	40.68	49.70	27.28	23.49	37.74	38.60	08.34

6.4.2 Results for RQ2

We compare the performance achieved from our approach (Our) against the existing approach (Existing) as shown in Table 6.2. For the pull request evaluation task, the analysis of all measures (i.e., Accuracy, Precision, Recall, F1-Score, AUC, and MMAE) suggests that our approach achieves better performance in most cases (23/24) compared to the existing approach. Our approach improves between 4.33% (in Java) to 21.36% (in R) in terms of Accuracy, 0.41% (declining in Python) to 3.01% (in Java) in terms of Precision, 5.97% (in Java) to 27.73% (in R) in terms of Recall, 3.68% (in Java) to 15.20% (in R) in terms of F1-Score, 1.13% (in Python) to 10.20% (in R) in terms of AUC, and 1.69% (in R) to 9.62% (in Ruby) in terms of MMAE.

For the pull request reopening task, our approach outperforms the existing approach in most cases (13/16) in terms of Balanced Accuracy, AUC, TPR, and FPR. Our approach improves over the baseline between 3.40% (in Python) to 15.00% (in Ruby) in terms of Balanced Accuracy, 3.89% (in Python) to 17.45% (in Ruby) in terms of AUC, and 0.00% (in R) to 92.00% (in Java) in terms of TPR, while the results for FPR are mixed, with some cases showing improvement and others showing a decline compared to the existing approach. Explicitly, our approach improves FPR by 9.63% (in Ruby) and 20.41% (in R) over the existing approach, while showing a decline in Python and Java. Ruby is also the programming language where our approach achieves the highest performance, as it consistently outperforms the existing approach in all evaluation measures.

On average, our approach obtains better performance in all measures, except FPR. The existing approach achieves an Accuracy of 0.701, Precision of 0.869, Recall of 0.709, F1-Score of 0.779, AUC of 0.713 for the acceptance task, while it gains an MMAE of 1.239 for the latency task and it obtains a Balanced Accuracy of 0.564, AUC of 0.630, TPR of 0.533, and FPR of 0.404. Overall, our approach shows better performance than the existing approach.

Our proposed approach outperforms the existing approach in all four programming languages. We can, thus, conclude that textual features extracted from the pre-trained models, our oversampling technique, shared learning, and deep learning classifiers improve the performance for prediction of pull request evaluation and pull request reopening.

It is worth noting that in our reopening experiments, we observed the FPR improvement in 14 out of 16 cases over the baseline, while we were unable to improve in two cases. Moreover, we observed the FPR improvement in 13 out of 16 cases over the existing approach, while we were unable to improve in three cases. However, it is crucial to consider that the importance of TPR or FPR may vary depending on the specific application and cost associated with each project. In our study, we have used AUC as the main evaluation metric, which provides a balanced measure between TPR and FPR. This approach allows us to account for the trade-off between sensitivity and specificity, and strike a balance in our analysis. Based on AUC, our results excel in all cases.

Apart from evaluating our approach's performance using standard measures, we conducted further experiments to assess the advantages of each proposed component compared to the existing approach. Our additional findings revealed the importance of textual features extracted from pre-trained embeddings for both pull request evaluation and pull request reopening predictions (see Section 6.5.3 for more details). Furthermore, our oversampling approach which combines SMOTE with VAE, proved to generate synthetic samples more effectively by mitigating outliers and noise compared to using SMOTE alone (see Section 6.5.2 for a comprehensive analysis). This effectively reduced the problem of overfitting during model training. Additionally, we ensured that our designed architecture was suitable for capturing the relationships between pull request outcomes. The results of a Chi-squared test of independence indicated the presence of relationships, enabling shared learning and validating our architecture (see Section 6.5.1 for detailed insights). The feature importance analysis also highlighted the inclusion of the reopening output as one of the most important features for the pull request evaluation stage. This finding reinforces the notion that the reopening output serves as a valuable predictor for evaluating pull requests and confirms the appropriateness of our architecture, where reopening prediction precedes evaluation prediction (see Section 6.5.3 for more information).

Finally, it is inevitable that our designed deep learning-based architecture played a main role to improve the predictive performance over the existing approach because it offered flexibility in designing complex architectures that capture outcome relationships and enable multi-output prediction. More precisely, the architectures for predicting reopening and evaluation are separated, with the reopening prediction being conducted before the evaluation prediction. Before finalizing this design, we explored several other designs, such as a single model architecture with shared learning among all three outputs, but found it to be insufficiently generalizable. Furthermore, our designed architecture offered an opportunity to address the issue of data imbalance specifically for reopening data, with our well-defined approach, SMOTE with VAE. The deep learning-based approach also presented feature learning abilities, allowing for intricate feature engineering tasks, for example, automatic feature engineering and textual feature extraction by pre-trained embedding models. Therefore, we can confidently conclude that our deep learning-based architecture significantly contributes to the improved predictive performance of pull request evaluation and reopening against the existing approach. Nevertheless, considering the aforementioned reasons, it is undeniable that all of our proposed components also play a vital role in encouraging the better predictive performance.

Table 6.2: Performance comparison between our approach (Our) and the existing approach (Existing) for predicting the pull request evaluation and reopening, reported as Accuracy (Acc), Precision (P), Recall (R), F1-Score (F1), AUC, Macro-Averaged Absolute Error (MMAE), Balanced Accuracy (BA), True Positive Rate (TPR), and False Positive Rate (FPR) along with percentage improvement (Percent Imp)

Language	**Approach**	**Acceptance**					**Latency**	**Reopening**			
		Acc	**P**	**R**	**F1**	**AUC**	**MMAE**	**BA**	**AUC**	**TPR**	**FPR**
	Existing	0.681	**0.878**	0.681	0.767	0.708	1.270	0.618	0.705	0.500	**0.264**
Python	Our	**0.719**	0.874	**0.742**	**0.803**	**0.716**	**1.171**	**0.639**	**0.732**	**0.679**	0.400
	Percent Imp	05.56	-00.41	08.97	04.66	01.13	07.82	03.40	03.89	35.71	-51.77
	Existing	0.594	0.865	0.609	0.714	0.589	1.224	0.527	0.649	**0.765**	0.709
R	Our	**0.721**	**0.874**	**0.777**	**0.823**	**0.649**	**1.204**	**0.600**	**0.715**	**0.765**	**0.564**
	Percent Imp	21.38	01.10	27.73	15.20	10.20	01.69	13.70	10.25	00.00	20.41
	Existing	0.792	0.815	0.779	0.797	0.872	1.198	0.554	0.584	0.362	**0.254**
Java	Our	**0.826**	**0.839**	**0.826**	**0.832**	**0.909**	**1.139**	**0.590**	**0.624**	**0.700**	0.515
	Percent Imp	04.33	03.01	05.97	04.50	04.31	04.96	06.50	06.94	92.00	-102.90
	Existing	0.737	0.920	0.769	0.838	0.682	1.262	0.557	0.583	0.506	0.391
Ruby	Our	**0.781**	**0.925**	**0.819**	**0.868**	**0.720**	**1.141**	**0.641**	**0.684**	**0.635**	**0.354**
	Percent Imp	06.00	00.53	06.47	03.68	05.49	09.62	15.00	17.45	25.58	09.63
	Existing	0.701	0.869	0.709	0.779	0.713	1.239	0.564	0.630	0.533	**0.404**
AVERAGE	Our	**0.762**	**0.878**	**0.791**	**0.832**	**0.749**	**1.163**	**0.618**	**0.689**	**0.694**	0.458
	Percent Imp	08.68	01.01	11.49	06.77	05.02	06.07	09.43	09.37	30.07	-13.32

6.5 Further Discussion of the Findings

The preceding section presented the results obtained from the research study based on the performance measures. In this section, we provide a more in-depth analysis and discussion of the findings. Through this analysis, we explore the implications of the results and their significance for the broader research question. Firstly, we explore relationships between the pull request outcomes. Furthermore, we state the advantage of oversampling using SMOTE combined with VAE compared to using only SMOTE. Additionally, we investigate the importance of features. Finally, we discuss the best configurations of our approach and the representative classifiers for the existing approach in each prediction task.

6.5.1 Relationships between the Pull Request Outcomes

As our approach utilized shared learning techniques for modeling, we tested the relationship between three different pull request outcomes: acceptance, latency, and reopening using the *Chi-squared test of independence* (Mchugh, 2013). The Chi-square test is a useful non-parametric statistic for testing hypotheses with nominal variables. The table reports the p-values for each combination of programming language and pull request outcome. A Chi-square test statistic (χ^2) is calculated as shown in Equation 6.1:

$$\chi^2 = \sum \frac{(O - E)^2}{E} \tag{6.1}$$

where O is the observed frequency and E represents the expected frequency for each cell while $\sum$ denotes the summation over all cells.

A significance level of 0.05 was used to determine statistical significance. The null hypothesis (H0) states that there is no relationship between the pull request outcomes, while the alternative hypothesis (H1) states that there is a relationship. To ensure reliable results, we applied certain criteria regarding the assumption about the sample size. More precisely, we required that the expected value in each cell is five or more in at least 80% of the cells, and no cell has an expected value less than three. Additionally, to ensure an adequate sample size, the sample size is at least equal to the number of cells multiplied by five (Mchugh, 2013) but not excessively large, as a large sample size can make the test overly sensitive to minor differences (Lin et al., 2013).

Table 6.3 shows the results of a Chi-squared test of independence between the pull request outcomes for different programming languages. The table provides evidences regarding the statistical significance, including p-values and significance codes, of the relationships observed. In this analysis, all combinations of pull request outcomes, including acceptance, latency, and reopening, were found to be statistically significant with a p-value less than 0.05. Consequently, we reject the null hypothesis, indicating that there are indeed relationships between pull request outcomes and programming languages. These findings further support previous studies and existing literature that have reported associations between pull request acceptance, latency, and reopening.

Table 6.3: Chi-Square Test of Independence between the pull request outcomes, reported as p-values and significance codes

Language		**Acceptance**	**Latency**	**Reopening**
Python	**Acceptance**	-	<0.001 (***)	0.042 (*)
	Latency	<0.001 (***)	-	<0.001 (***)
R	**Acceptance**	-	<0.001 (***)	<0.001 (***)
	Latency	<0.001 (***)	-	<0.001 (***)
Java	**Acceptance**	-	<0.001 (***)	<0.001 (***)
	Latency	<0.001 (***)	-	<0.001 (***)
Ruby	**Acceptance**	-	<0.001 (***)	<0.001 (***)
	Latency	<0.001 (***)	-	<0.001 (***)

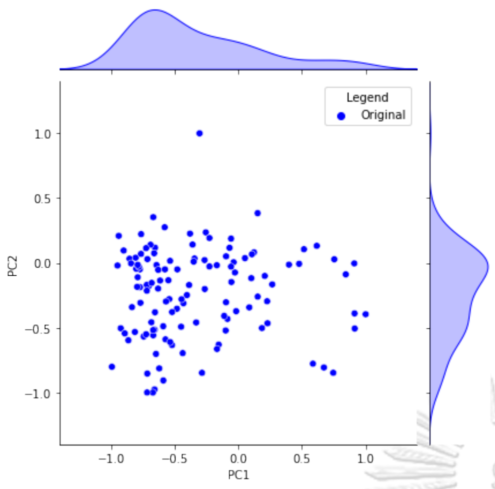
Sig. codes: <0.001 (***) 0.01 (**) 0.05 (*) 0.1 (.) 1 (_)

6.5.2 Comparison between the Oversampling Techniques

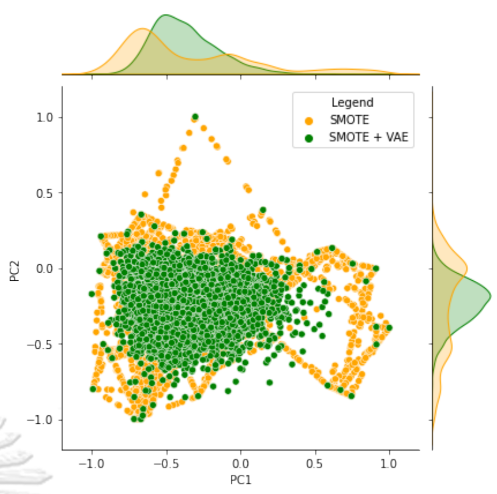
In the matter of the oversampling techniques, we will state the advantage of oversampling using SMOTE combined with VAE compared to using only SMOTE. To illustrate the effectiveness of these techniques, we present example cases utilizing PCA plot. Figure 6.1 and Figure 6.2 show 2-dimensional PCA plots of original reopening samples and synthetic reopening samples by SMOTE and SMOTE combined with VAE in the Python and Ruby languages. When comparing the synthetic samples generated by SMOTE (shown in orange) and SMOTE combined with VAE (shown in green) in Figure 6.1 (b) and Figure 6.2 (b), we can observe a clear difference in their distribution. While both methods generate synthetic samples, it is evident that SMOTE generates some noise and also synthesizes samples by considering outliers much. The problem that is most likely to occur is overfitting. This is because the model will try to fit the noise and outliers in the training data, which can lead to poor generalization performance on unseen data. On the other hand, the synthetic samples generated by SMOTE combined with VAE are much more evenly distributed and closely resemble the distribution of the original samples without considering the outliers. Therefore, we can conclude that oversampling using SMOTE combined with VAE is a better approach compared to using only SMOTE.

6.5.3 Feature Importance

We investigated which features serve as good predictors for our approach in both the pull request reopening stage and the pull request evaluation stage. To determine the importance of features, we employed a *sensitivity analysis*, one of the *gradient-based attribution methods* (Ancona et al., 2018, 2019; Nielsen et al., 2022). This method involves analyzing the gradients of the loss function with respect to the input features. It measures the extent to which the loss function changes when the input features are perturbed. The underlying idea is that if a feature is highly important, perturbing that feature should result in a significant change in the model's predictions and, consequently, in the gradients of the loss function. Conversely, if a feature has low importance, perturbing it should have minimal effect on the gradi-

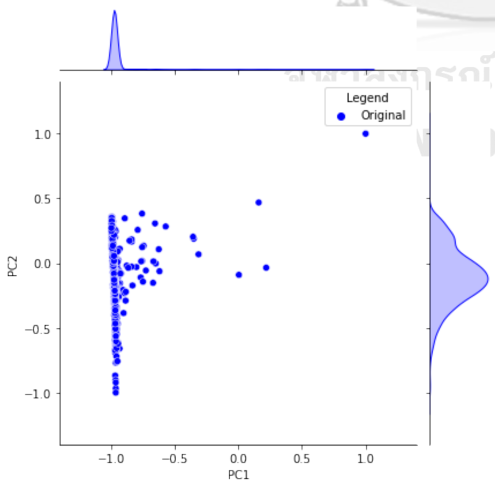


(a) Original reopening samples

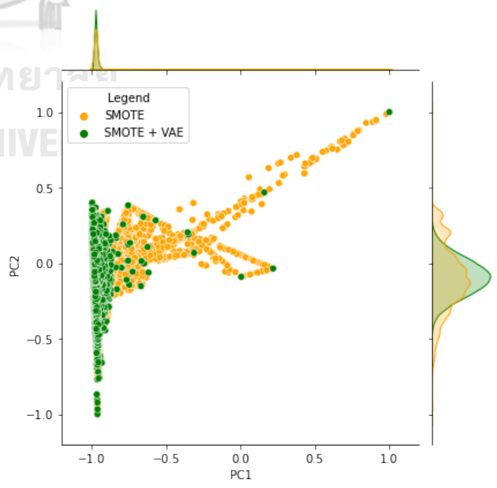


(b) Synthetic reopening samples

Figure 6.1: PCA plots of original reopening samples and synthetic reopening samples by SMOTE and SMOTE combined with VAE in the Python language



(a) Original reopening samples



(b) Synthetic reopening samples

Figure 6.2: PCA plots of original reopening samples and synthetic reopening samples by SMOTE and SMOTE combined with VAE in the Ruby language

ents. In short, the gradient-based method determines the contribution of each input feature to the output based on the gradients.

Formally, let's consider a deep neural network (DNN) that takes an input vector $x = [x_1, ..., x_N] \in \mathbb{R}^N$ and produces an output vector $S(x) = [S_1(x), ..., S_C(x)]$, where C represents the total number of output neurons. Given a specific target neuron c, the goal of an attribution method is to determine the contribution $R^c = [R^c_1, ..., R^c_N] \in \mathbb{R}^N$ of each input feature x_i to the output S_c. In the sensitivity analysis, attributions are constructed by taking the absolute value of the partial derivative of the target output S_c with respect to the inputs x_i as defined below:

$$R^c_i(x) = \left| \frac{\partial S_c(x)}{\partial x_i} \right| \tag{6.2}$$

The sensitivity analysis method, within the broader gradient-based approach, comprised several steps. Firstly, we created a new model using the trained weights, which captured the learned relationships between the input features and the model's predictions. Secondly, we defined the loss function to focus solely on the activations of the output layer. Lastly, we performed the sensitivity analysis by calculating and analyzing the gradients of the loss function with respect to the input features. These gradients quantified the sensitivity of the model's predictions to changes in each input feature. Taking the absolute values of these gradients, we estimated the importance of each feature. A larger absolute gradient magnitude for a particular feature indicated a higher sensitivity and, consequently, a greater importance in influencing the model's predictions. To obtain a comprehensive measure of feature importance, we calculated the average absolute gradient magnitude across the training data for each feature.

Table 6.4, Table 6.5, Table 6.6, and Table 6.7 present the top 10 most significant features and their corresponding normalized weights (Imp) for the reopening and evaluation stages of specific programming language. The weights were normalized to range from 0 (least important) to 1 (most important) to provide a relative measure of importance for each feature. In the Python and Java languages, textual features appear to be crucial in the reopening stage as they dominate in more than half of the features (refer to Table 6.4 and Table 6.6). For the R and Ruby languages, textual features play a primary role in both the reopening and evaluation stages (refer to Table 6.5 and Table 6.7). Across all four programming languages, it is evident that textual features constitute over half of the total top 10 features (45 out of 80). This leads to the conclusion that our proposed textual features are reliable predictors for both pull request reopening and pull request evaluation. Apart from the textual features, we can observe that the reopening probability, which is the output from the reopening stage and is utilized as one of the predictors for pull request evaluation, appears in the top-10 feature list for three languages which are Python, Java, and Ruby. This further signifies the importance of the reopening output in predicting pull request evaluation. It also validates the suitability of the architecture we designed, where we first predict pull request reopening and subsequently predict evaluation outputs.

Table 6.4: List of top 10 most important features with their normalized weight (Imp) in the reopening and evaluation stages for the Python language

Reopening		**Evaluation**	
Feature	**Imp**	**Feature**	**Imp**
reputation	1.00	contributor age	1.00
text 425	0.95	file rejected proportion	0.21
text 117	0.86	% of commits made by pull requests	0.14
# of events in pr	0.82	reopening probability	0.08
text 570	0.79	reputation	0.07
text 687	0.77	file rejected proportion	0.05
text 582	0.77	is assignee	0.05
text 563	0.75	text 117	0.05
text 128	0.73	text 160	0.05
text 111	0.69	text 66	0.05

Table 6.5: List of top 10 most important features with their normalized weight (Imp) in the reopening and evaluation stages for the R language

Reopening		Evaluation	
Feature	**Imp**	**Feature**	**Imp**
contributor age	1.00	text 52	1.00
has pr link	0.15	text 257	0.80
% commits by pulls	0.12	text 282	0.79
text 44	0.07	text 299	0.78
is core team	0.07	text 249	0.75
text 62	0.06	text 273	0.74
text 18	0.06	contributor age	0.73
text 99	0.06	text 212	0.73
is recent pr rejected	0.06	text 21	0.71
text 181	0.05	text 261	0.71

Table 6.6: List of top 10 most important features with their normalized weight (Imp) in the reopening and evaluation stages for the Java language

Reopening		Evaluation	
Feature	**Imp**	**Feature**	**Imp**
text 63	1.00	contributor age	1.00
text 67	0.94	% of commits made by pull requests	0.47
text 38	0.90	file rejected proportion	0.40
text 182	0.89	text 10	0.35
text 12	0.88	reopening probability	0.29
text 72	0.87	reputation	0.27
text 181	0.87	# of merged pr	0.20
text 73	0.85	# of rejected pr	0.19
text 262	0.74	has test	0.17
text 221	0.74	# of contributor commits	0.16

6.5.4 The Best Model Configuration of Our Approach

In this section, we provide an overview of the best model configuration for our approach, which is detailed in Appendix A. The configuration includes the utilization of pre-trained embeddings, Variational Autoencoder (VAE), and Deep Neural Network (DNN) components. To present the configuration clearly, we introduce

Table 6.7: List of top 10 most important features with their normalized weight (Imp) in the reopening and evaluation stages for the Ruby language

Reopening		**Evaluation**	
Feature	**Imp**	**Feature**	**Imp**
is assignee	1.00	file rejected proportion	1.00
% of commits made by pull requests	0.91	contributor age	0.65
has pr link	0.90	% of commits made by pull requests	0.63
has test	0.85	reputation	0.46
text 9	0.79	text 299	0.42
# of commits	0.78	text 732	0.41
text 23	0.77	reopening probability	0.32
# of deleted lines	0.75	text 342	0.29
text 35	0.74	text 105	0.29
text 48	0.73	text 676	0.26

Table A.1, Table A.3, Table A.5, and Table A.7 for the pull request reopening stage, and Table A.2, Table A.4, Table A.6, and Table A.8 for the pull request evaluation stage. These tables summarize the best configuration settings with detailed information such as the number of hidden units, activation functions, dropout rates, and normalization employed in each layer.

By presenting this comprehensive model configuration, we aim to provide a clear understanding of the key components and settings used in our approach for both the pull request reopening and evaluation stages. This information serves as a reference for replicating and further analyzing our approach, and also offers insights into the design choices made to achieve optimal predictive performance. For example, The tables indicate that the Linear activation function is predominantly used in the VAEs, while the ReLU activation function is primarily employed in the DNNs. Furthermore, the results show that BERT is the best pre-trained model for the reopening stage, while Word2vec is the best for the evaluation stage in Python. However, it is worth noting that while Word2vec and FastText were found to be the best pre-trained models for different stages of the approach in multiple programming languages, the pattern is not consistent across all languages. These variations suggest that the choice of the best pre-trained model depends on the specific programming language and the stage of the approach. Therefore, researchers and practitioners should carefully evaluate different models before deciding on the most suitable one for their specific use case.

6.5.5 The Best Classifiers for the Existing Approach

Table 6.8 presents the representative classifiers for the existing approach in each programming language. These classifiers were selected based on their performance in three distinct prediction tasks: acceptance, latency, and reopening, which were used as benchmarks for comparison with our approach. For example, in the Python cases, Random Forest (RF) emerged as the representative classifier for the acceptance task, while XGBoost (XGB) took the lead in both the latency and reopening tasks. Analyzing the results, it is evident that XGBoost showcased the best performance in both the acceptance and reopening tasks across multiple programming languages. On the other hand, Random Forest demonstrated favorable results in the majority of cases for the latency tasks. Interestingly, Decision Tree did not make it to the list of representative classifiers.

Table 6.8: List of the representative classifiers for the existing approach

Language	Acceptance	Latency	Reopening
Python	RF	XGB	XGB
R	XGB	RF	XGB
Java	XGB	RF	XGB
Ruby	XGB	RF	XGB

6.6 Threats to Validity

In this section, we will outline threats to external, internal, construct, and conclusion validity, and detail the steps taken to minimize their impact on our results.

6.6.1 External Validity

Our study provides a broad range of perspectives by analyzing 54 real-world well-known open-source projects across four popular programming languages on GitHub. However, our findings may not be representative of all programming languages and all kinds of software projects, especially in commercial settings. To address this limitation, we plan to expand our experiment to a more diverse range of projects and languages in the future.

6.6.2 Internal Validity

In order to ensure internal validity, we have taken several measures to minimize bias and errors throughout our study. One crucial aspect is the utilization of actual pull request outputs from real integrators, which provides us with authentic and reliable data. By incorporating real-world data, we aim to capture the practical aspects and challenges associated with pull request acceptance, latency, and reopening in open-source projects on GitHub. Furthermore, we have meticulously processed and analyzed only the information that was available at the time of the pull request submission. This was achieved by scraping the GitHub website, which allowed us to extract relevant features and avoid any potential information leakage. By strictly adhering to the information available during the pull request submission process, we maintain the integrity and accuracy of our predictions.

6.6.3 Construct Validity

We adopt standard evaluation metrics commonly used in classification tasks. These metrics have also been employed in prior software engineering research to assess the effectiveness of different approaches, enabling us to compare and validate our results. However, evaluating the reopening prediction presents a challenge due to highly imbalanced data, and there is limited prior work that addresses this issue. Therefore, we employ common evaluation metrics that have been used in other domains to assess our approach's performance on this task.

6.6.4 Conclusion Validity

We take a meticulous and cautious approach when drawing conclusions based on the extracted features from the studied project repositories. However, it should be noted that the latency may not always reflect the actual review and integration time of a pull request, as there may be other factors beyond the integration process such as the integrator having a heavy workload or lack of interaction with the contributor (de Lima Júnior et al., 2021). Additionally, the pull request reopening may not always indicate the actual reopening because it can occur due to accidental closure (Jiang et al., 2019).

CHAPTER VII

CONCLUSIONS

This chapter presents the conclusions of this research by providing a summary of the motivation, objectives, proposed approach, evaluation results, key findings, and potential avenues for future research.

7.1 Summary

Open-source software projects have adopted GitHub's pull-based model to allow contributors to make software changes in a flexible and efficient manner through a pull request. Contributors create pull requests for merging changes into the main project repository, and integrators review these requests to maintain quality and stability. However, the emergence of pull-based development increases the workload for integrators, particularly in large projects. Current studies have examined various factors and built predictive models using traditional machine learning for pull request acceptance and latency, with limited research on reopening prediction and timely integrator support. Reopened pull requests can introduce additional maintenance costs and create a burden for developers who are already busy. Furthermore, the challenges posed by text data and the imbalanced nature of reopened pull requests have not been adequately addressed. Additionally, there is a notable absence of models that offer timely predictions for integrators right after a pull request is submitted.

In this thesis, we propose a novel deep learning-based approach to early predict pull request acceptance, latency, and reopening in open-source projects hosted on GitHub. Our prediction is delivered at the time of pull request submission to enable integrators to plan their work more effectively. We design separate architectures for predicting reopening and evaluation, with the reopening prediction conducted prior to the evaluation prediction. Our approach makes use of both tabular features and textual features. We also leverage the state-of-the-art pre-trained models, the

SMOTE combined with VAE as the oversampling technique, shared learning between outputs, and deep neural networks to address the gaps and the challenges, and to improve the predictive performance for prediction of pull request evaluation and pull request reopening.

We conducted an extensive evaluation on several popular GitHub open-source software projects across four well-known programming languages, which revealed the superiority of our approach over random guessing and existing methods. On average, we achieved an Accuracy of 0.762, a Precision of 0.878, a Recall of 0.791, and an F1-Score of 0.832 in acceptance prediction, while we obtained an MMAE of 1.163 in latency prediction. For reopening prediction, we yielded a balanced accuracy of 0.618, an AUC of 0.689, and a TPR of 0.694. Our approach achieved significant improvements in Accuracy, Precision, Recall, F1-Score, MMAE, Balanced Accuracy, AUC, and TPR outperforming the existing method by 8.68%, 1.01%, 11.49%, 6.77%, 6.07%, 9.43%, 9.37%, and 30.07% on average. We can, thus, conclude that our proposed approach which incorporates combination of textual features extracted from the pre-trained model, the oversampling technique, shared learning, and deep learning classifier had an advantage for the prediction of pull request evaluation and pull request reopening.

7.2 Future work

In addition to our current findings, we believe that there is still room for improvement and further exploration in the area of pull request evaluation and reopening prediction. As such, we plan to validate our approach with a wider range of programming languages and larger projects, especially those in industrial settings. This will help us determine the generalizability of our approach and its suitability for use in real-world scenarios.

Furthermore, we aim to explore new sources of information that can better characterize pull requests. For instance, we plan to investigate the use of code changes as a source of information for enhancing the predictive performance of our approach, particularly for the reopening task. By incorporating code changes into our feature extraction process, we hope to capture more nuanced and relevant information that can help to better inform the prediction of pull request evaluation and reopening. Additionally, we will conduct a thorough investigation of the oversampling techniques for the reopening task. We will compare each technique in detail and analyze their in-depth effectiveness.

Finally, we plan to take the next step in the development of our approach by integrating it as a tool within the GitHub platform. This will allow us to gather feedback from real users and enable future analysis and refinement of the approach. By making our approach available to a wider audience, we hope to increase its impact and encourage further research in this area. We believe that our work has the potential to contribute to the development of more effective and efficient software engineering practices, which can ultimately lead to higher-quality software products and better outcomes for end users.

REFERENCES

Al-Zubaidi, W. H. A., Dam, H. K., Choetkiertikul, M., and Ghose, A. 2018. Multi-Objective Iteration Planning in Agile Development. Proceedings of Asia-Pacific Software Engineering Conference, APSEC 2018-Decem (2018): 484–493.

Ancona, M., Ceolini, E., Öztireli, C., and Gross, M. 2018. Towards better understanding of gradient-based attribution methods for deep neural networks. Proceedings of 6th International Conference on Learning Representations (ICLR 2018) .April (2018):

Ancona, M., Ceolini, E., Öztireli, C., and Gross, M. 2019. Gradient-Based Attribution Methods. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 11700 LNCS, pp. 169–191. Cham: Springer International Publishing.

Arias-Londoño, J. D., Gómez-García, J. A., and Godino-Llorente, J. I. 2019. Multimodal and multi-output deep learning architectures for the automatic assessment of voice quality using the GRB scale. (nov 2019):

Azeem, M. I., Peng, Q., and Wang, Q. 2020. Pull Request Prioritization Algorithm based on Acceptance and Response Probability. In Proceedings of IEEE 20th International Conference on Software Quality, Reliability, and Security (QRS 2020), pp. 231–242. Macau, China: Institute of Electrical and Electronics Engineers Inc.

Baccianella, S., Esuli, A., and Sebastiani, F. 2009. Evaluation Measures for Ordinal Regression. In 2009 Ninth International Conference on Intelligent Systems Design and Applications, pp. 283–287. Pisa, Italy.

Baevski, A., Zhou, H., Mohamed, A., and Auli, M. 2020. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. CoRR abs/ 2006.11477 (2020):

Bird, C. and Zimmermann, T. 2012. Assessing the Value of Branches with What-If Analysis. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12. New York, NY, USA: Association for Computing Machinery.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. 2009. The promises and perils of mining git. In Proceedings of 2009 6th IEEE International Working Conference on Mining Software Repositories, pp. 1–10. Vancouver, BC, Canada.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. 2016. Enriching word vectors with subword information. arXiv preprint arXiv:1607.04606 (2016):

Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. 1997. Syntactic clustering of the Web. Computer Networks and ISDN Systems 29.8 (1997): 1157–1166.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. 2002. SMOTE: Synthetic Minority over-Sampling Technique. J. Artif. Int. Res. 16.1 (jun 2002): 321–357.

Chen, D., Stolee, K., and Menzies, T. 2019. Replication can improve prior results: A github study of pull request acceptance. In Proceedings of IEEE International Conference on Program Comprehension, volume 2019-May, pp. 179–190. Montreal, QC, Canada: IEEE Computer Society.

Choetkiertikul, M., Dam, H. K., Tran, T., and Ghose, A. 2015. Characterization and prediction of issue-related risks in software projects. In IEEE International Working Conference on Mining Software Repositories, volume 2015-Augus, pp. 280–291. .

Choetkiertikul, M., Dam, H. K., Tran, T., Ghose, A., and Grundy, J. 2018. Predicting Delivery Capability in Iterative Software Development. IEEE Transactions on Software Engineering 44.6 (jun 2018): 551–573.

Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. : OpenReview.net.

Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. 2013. Leveraging transparency. IEEE Software 30.1 (2013): 37–43.

Dai, W., Ng, K., Severson, K., Huang, W., Anderson, F., and Stultz, C. 2019. Generative oversampling with a contrastive variational autoencoder. Proceedings of IEEE International Conference on Data Mining, ICDM 2019-November (2019): 101–109.

de Lima Júnior, M. L., Soares, D. M., Plastino, A., and Murta, L. 2018. Automatic assignment of integrators to pull requests: The importance of selecting appropriate attributes. Journal of Systems and Software 144 (2018): 181–196.

de Lima Júnior, M. L., Soares, D., Plastino, A., and Murta, L. 2021. Predicting the lifetime of pull requests in open-source projects. Journal of Software: Evolution and Process 33.6 (jun 2021):

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (oct 2018):

Geisser, S. 1993. Predictive Inference: An Introduction. Monographs on statistics and applied probability. Springer International Publishing, New York. ISBN 9781489944672. Available from: https://books.google.co.th/books?id=MKjZnQEACAAJ .

Gousios, G., Pinzger, M., and Deursen, A. V. 2014. An exploratory study of the pull-based software development model. In Proceedings of International Conference on Software Engineering, number 1 in ICSE 2014, pp. 345–355. Hyderabad, India: IEEE Computer Society.

Gousios, G., Zaidman, A., Storey, M.-A., and van Deursen, A. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, pp. 358–368. Florence, Italy.

Gousios, G., Storey, M. A., and Bacchelli, A. 2016. Work practices and challenges in pull-based development: The contributor's perspective. In Proceedings of International Conference on Software Engineering, volume 14-22-May-2016, pp. 285–296. Austin, TX, USA: IEEE Computer Society.

Harris, Z. S. 1954. Distributional Structure. WORD 10.2-3 (1954): 146–162.

Janiesch, C., Zschech, P., and Heinrich, K. 2021. Machine learning and deep learning. 31 (2021): 685–695.

Jiang, J., Mohamed, A., and Zhang, L. 2019. What are the Characteristics of Reopened Pull Requests? A Case Study on Open Source Projects in GitHub. IEEE Access 7 (2019): 102751–102761.

Jiang, J., teng Zheng, J., Yang, Y., and Zhang, L. 2020. CTCPPre: A prediction method for accepted pull requests in GitHub. Journal of Central South University 27.2 (feb 2020): 449–468.

Kale, R., Lu, Z., Fok, K. W., and Thing, V. L. L. 2022. A Hybrid Deep Learning Anomaly Detection Framework for Intrusion Detection. In 2022 IEEE 8th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS), pp. 137–142. Jinan, China.

Khan, M. Z. 2020. Hybrid ensemble learning technique for software defect prediction. International Journal of Modern Education and Computer Science 12.1 (2020): 1–10.

Kingma, D. P. and Welling, M. 2014. Auto-encoding variational bayes. Proceedings of 2nd International Conference on Learning Representations (ICLR 2014) (2014):

Knyazeva, I., Plotnikov, A., Medvedeva, T., and Makarenko, N. 2022. Multi-output Deep Learning Framework for Solar Atmospheric Parameters Inferring from Stokes Profiles. In Kryzhanovsky, B., Dunin-Barkowski, W., Redko, V., Tiumentsev, Y., and Klimov, V. V. (ed.), Advances in Neural Computation, Machine Learning, and Cognitive Research V, pp. 299–307. Cham: Springer International Publishing.

Lin, M., Lucas, H. C., and Shmueli, G. 2013. Too big to fail: Large samples and the p-value problem. Information Systems Research 24.4 (2013): 906–917.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (jul 2019):

Maddila, C., Upadrasta, S., Bansal, C., Nagappan, N., Gousios, G., and Deursen, A. 2022. Nudge: Accelerating Overdue Pull Requests Towards Completion. ACM Transactions on Software Engineering and Methodology (oct 2022):

Mchugh, M. L. 2013. The Chi-square test of independence Lessons in biostatistics. Biochemia Medica 23.2 (2013): 143–9.

McKee, S., Nelson, N., Sarma, A., and Dig, D. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME 2017) (2017): 467–478.

Mikolov, T. and Others. 2013. Distributed representations of words and phrases and their compositionality. Advances in Neural Information Processing Systems (2013): 1–9.

Mohamed, A., Zhang, L., Jiang, J., and Ktob, A. 2018. Predicting Which Pull Requests Will Get Reopened in GitHub. In Proceedings of Asia-Pacific

Software Engineering Conference (APSEC), volume 2018-Decem, pp. 375–385. : IEEE Computer Society.

Mohamed, A., Zhang, L., and Jiang, J. 2020. Cross-project reopened pull request prediction in github. In García-Castro, R. (ed.), Proceedings of 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, KSIR Virtual Conference Center, USA, July 9-19, 2020, pp. 435–438. USA: KSI Research Inc.

Nielsen, I. E., Dera, D., Rasool, G., Ramachandran, R. P., and Bouaynaya, N. C. 2022. Robust Explainability: A tutorial on gradient-based attribution methods for deep neural networks. IEEE Signal Processing Magazine 39.4 (2022): 73–84.

Nikhil Khadke, M. S., Ming Han Teh. 2012. Predicting Acceptance of GitHub Pull Requests. (2012):

Ortu, M., Destefanis, G., Graziotin, D., Marchesi, M., and Tonelli, R. 2020. How do you Propose Your Code Changes? Empirical Analysis of Affect Metrics of Pull Requests on GitHub. IEEE Access 8 (2020): 110897–110907.

Papamichail, M., Diamantopoulos, T., and Symeonidis, A. 2016. User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics. In Proceedings of 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 100–107. .

Rahman, M. M. and Roy, C. K. 2014. An insight into the pull requests of GitHub. In 11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings, pp. 364–367. : Association for Computing Machinery.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. 2016. SQuad: 100,000+ questions for machine comprehension of text. In EMNLP 2016 - Conference on Empirical Methods in Natural Language Processing, Proceedings, pp. 2383–2392. .

Salton, G. and Buckley, C. 1988. Term-weighting approaches in automatic text retrieval. Information Processing & Management 24.5 (jan 1988): 513–523.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. CoRR abs/1910.01108 (2019):

Saroop, A., Ghugare, P., Mathamsetty, S., and Vasani, V. 2021. Facial Emotion Recognition: A multi-task approach using deep learning. arXiv e-prints (oct 2021): arXiv:2110.15028.

Sarro, F., Petrozziello, A., and Harman, M. 2016. Multi-objective software effort estimation. In Proceedings of International Conference on Software Engineering, volume 14-22-May-, pp. 619–630. .

Shepperd, M. and MacDonell, S. 2012. Evaluating prediction systems in software project estimation. Information and Software Technology 54.8 (2012): 820–827.

Soares, D., Limeira, M., Murta, L., and Plastino, A. 2015. Acceptance factors of pull requests in open-source projects. In Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1541–1546. New York, NY, USA: Association for Computing Machinery.

Stack Overflow 2022. Stack Overflow Developer Survey 2022 [Online]. Available from: https://survey.stackoverflow.co/2022/ [2022,].

Tabassum, J., Maddela, M., Xu, W., and Ritter, A. 2020. Code and Named Entity Recognition in StackOverflow. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 4913–4926. Online: Association for Computational Linguistics.

Trauer, J., Pfingstl, S., Finsterer, M., and Zimmermann, M. 2021. Improving production efficiency with a digital twin based on anomaly detection. Sustainability (Switzerland) 13.18 (2021):

Tsay, J., Dabbish, L., and Herbsleb, J. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In Proceedings of International Conference on Software Engineering (ICSE 2014), number 1, pp. 356–366. Hyderabad, India: IEEE Computer Society.

Van Der Veen, E., Gousios, G., and Zaidman, A. 2015. Automatically prioritizing pull requests. In Proceedings of IEEE International Working Conference on Mining Software Repositories, volume 2015-August, pp. 357–361. : IEEE Computer Society.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, pp. 353–355. Brussels, Belgium: Association for Computational Linguistics.

Wattanakriengkrai, S., Srisermphoak, N., Sintoplertchaikul, S., Choetkiertikul, M., Ragkhitwetsagul, C., Sunetnanta, T., Hata, H., and Matsumoto, K. 2019. Automatic Classifying Self-Admitted Technical Debt Using N-Gram IDF. In Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), volume 2019-Decem, pp. 316–322. .

Xu, D., Shi, Y., Tsang, I. W., Ong, Y.-S., Gong, C., and Shen, X. 2020. Survey on Multi-Output Learning. IEEE Transactions on Neural Networks and Learning Systems 31.7 (2020): 2409–2429.

Yu, Y., Wang, H., Filkov, V., Devanbu, P., and Vasilescu, B. 2015. Wait For It: Determinants of Pull Request Evaluation Latency on GitHub. Proceedings of 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (2015): 367–371.

Zellers, R., Bisk, Y., Schwartz, R., and Choi, Y. 2018. SWAG: A large-scale adversarial dataset for grounded commonsense inference. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018, pp. 93–104. .

Zhang, J., Zhao, Y., Saleh, M., and Liu, P. J. 2019. PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. CoRR abs/ 1912.08777 (2019):

Zhang, X., Yu, Y., Gousios, G., and Rastogi, A. 2021a. Pull Request Decision Explained: An Empirical Overview. 49 (may 2021):

Zhang, X., Yu, Y., Wang, T., Rastogi, A., and Wang, H. 2021b. Pull Request Latency Explained: An Empirical Overview. 27 (aug 2021):

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. 2015. Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books. (jun 2015):

APPENDICES

APPENDIX A

THE MODEL CONFIGURATION

Table A.1: Model configuration of our approach in the reopening stage for the Python language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	Name	Layer	# Units	Act	Drop out	Norm?
Pre-trained	BERT	-	768	-	-	-
VAE	Encoder	Input	801	-	-	No
		Dense	128	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	64	Linear	0.1	Yes
		Output	32	Linear	-	No
	Decoder	Input	32	-	-	No
		Dense	64	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	128	Linear	0.1	Yes
		Output	801	Linear	-	No
DNN	-	Input	801	-	-	No
		Dense	32	ReLU	0.5	No
		Dense	16	ReLU	0.5	No
		Output	1	Softmax	-	No

Table A.2: Model configuration of our approach in the evaluation stage for the Python language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	**Name**	**Layer**	**# Units**	**Act**	**Drop out**	**Norm?**
Pre-trained	Word2Vec	-	200	-	-	-
DNN	-	Input	234	-	-	Yes
		Dense	512	ReLU	0.5	No
		Dense	1024	ReLU	0.5, 0.5	No
		Output	1, 5	SM, SM	-	No

Note that the output layer has two nodes for acceptance and latency, SM denotes the Softmax activation, and multiple numbers in the Dropput column represent stacked dropout layers.

Table A.3: Model configuration of our approach in the reopening stage for the R language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	**Name**	**Layer**	**# Units**	**Act**	**Drop out**	**Norm?**
Pre-trained	Word2Vec	-	200	-	-	-
VAE	Encoder	Input	233	-	-	No
		Dense	128	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	64	Linear	0.1	Yes
		Output	32	Linear	-	No
	Decoder	Input	32	-	-	No
		Dense	32	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	128	Linear	0.1	Yes
		Output	233	Linear	-	No
DNN	-	Input	233	-	-	Yes
		Dense	256	ReLU	-	No
		Dense	128	ReLU	-	No
		Dense	32	ReLU	-	No
		Output	1	Softmax	-	No

Table A.4: Model configuration of our approach in the evaluation stage for the R language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	**Name**	**Layer**	**# Units**	**Act**	**Drop out**	**Norm?**
Pre-trained	FastText	-	300	-	-	-
DNN	-	Input	334	-	-	Yes
		Dense	32	ReLU	-	No
		Dense	64	ReLU	-	No
		Dense	32	ReLU	0.25	No
		Output	1, 5	SM, SM	-	No

Note that the output layer has two nodes for acceptance and latency and SM denotes the Softmax activation.

Table A.5: Model configuration of our approach in the reopening stage for the Java language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	**Name**	**Layer**	**# Units**	**Act**	**Drop out**	**Norm?**
Pre-trained	Word2Vec	-	200	-	-	-
VAE	Encoder	Input	333	-	-	No
		Dense	256	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	32	Linear	0.1	Yes
		Output	64	Linear	-	No
	Decoder	Input	64	-	-	No
		Dense	32	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	256	Linear	0.1	Yes
		Output	333	Linear	-	No
DNN	-	Input	333	-	-	No
		Dense	128	ReLU	-	No
		Dense	256	ReLU	0.25	No
		Output	1	Softmax	-	No

Table A.6: Model configuration of our approach in the evaluation stage for the Java language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	**Name**	**Layer**	**# Units**	**Act**	**Drop out**	**Norm?**
Pre-trained	W2V	-	200	-	-	-
DNN	-	Input	234	-	-	Yes
		Dense	128	ReLU	0.25, 0.25, 0.5	Yes
		Output	1, 5	SM, SM	-	No

Note that the output layer has two nodes for acceptance and latency, SM denotes the Softmax activation, and multiple numbers in the Dropput column represent stacked dropout layers

Table A.7: Model configuration of our approach in the reopening stage for the Ruby language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	Name	Layer	# Units	Act	Drop out	Norm?
Pre-trained	Word2Vec	-	200	-	-	-
VAE	Encoder	Input	233	-	-	No
		Dense	128	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	64	Linear	0.1	Yes
		Output	32	Linear	-	No
	Decoder	Input	32	-	-	No
		Dense	32	Linear	0.1	Yes
		Dense	128	ReLU	0.1	Yes
		Dense	128	Linear	0.1	Yes
		Output	233	Linear	-	No
DNN	-	Input	233	-	-	No
		Dense	128	ReLU	0.25	No
		Dense	128	ReLU	0.25	No
		Dense	512	ReLU	0.25	No
		Output	1	Softmax	-	No

Table A.8: Model configuration of our approach in the evaluation stage for the Ruby language, reported with their model name, layers, number of units (# Units), activation function (Act), dropout rate (Dropout), and normalization (Norm?)

Model	**Name**	**Layer**	**# Units**	**Act**	**Drop out**	**Norm?**
Pre-trained	BERT	-	768	-	-	-
DNN	-	Input	802	-	-	Yes
		Dense	128	ReLU	0.25	Yes
		Dense	1024	ReLU	0.25	Yes
		Dense	512	ReLU	0.25, 0.5, 0.5	Yes
		Output	1, 5	SM, SM	-	No

Note that the output layer has two nodes for acceptance and latency, SM denotes the Softmax activation, and multiple numbers in the Dropput column represent stacked dropout layers.

BIOGRAPHY

NAME	Mr. Peerachai Banyongrakkul
DATE OF BIRTH	16 January 1998
PLACE OF BIRTH	Bangkok, Thailand
HOME ADDRESS	Bangkok, Thailand
INSTITUTIONS ATTENDED	Matthayom Watnairong English Program School, 2015
	High School Diploma
	Mahidol University, 2019
	Bachelor of Science (ICT)
	First Class Honours
	Chulalongkorn University, 2022
	Master of Science (Statistics)