AUTOMATED PRIORITY DETECTION IN SOFTWARE BUGS: A COMPREHENSIVE STUDY ON TRANSFORMER-BASED ENCODERS WITH CONTRASTIVE LEARNING, LARGE LANGUAGE MODELS AND VECTOR DATABASES FOR ENHANCED EFFICIENCY

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EYÜP HALIT YILMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY 2024

Approval of the thesis:

AUTOMATED PRIORITY DETECTION IN SOFTWARE BUGS: A COMPREHENSIVE STUDY ON TRANSFORMER-BASED ENCODERS WITH CONTRASTIVE LEARNING, LARGE LANGUAGE MODELS AND VECTOR DATABASES FOR ENHANCED EFFICIENCY

submitted by **EYÜP HALIT YILMAZ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. İsmail Hakkı Toroslu
Supervisor, **Computer Engineering, METU** _____

Assist. Prof. Dr. Ömer Köksal
Co-supervisor, **University of Doha for Science & Technology** _____

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering, METU _____

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering, METU _____

Assoc. Prof. Dr. Murat Özbayoğlu
TOBB Univ. of Economics and Technology _____

Date:25.01.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Eyüp Halit Yılmaz

Signature :

ABSTRACT

AUTOMATED PRIORITY DETECTION IN SOFTWARE BUGS: A COMPREHENSIVE STUDY ON TRANSFORMER-BASED ENCODERS WITH CONTRASTIVE LEARNING, LARGE LANGUAGE MODELS AND VECTOR DATABASES FOR ENHANCED EFFICIENCY

Yılmaz, Eyüp Halit
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. İsmail Hakkı Toroslu
Co-Supervisor: Assist. Prof. Dr. Ömer Köksal

January 2024, 53 pages

Software development processes include many challenges that require human effort and time investment. In time, many tools and techniques are developed to address these challenges and automate parts of software development and maintenance. Software bug reports are textual descriptions, often accompanied by code snippets and error logs, written by users or developers documenting operational failures of programs. These reports are later examined by the assigned developer to fix the bug. Automating the bug fixing pipeline helps determine the most suitable developer to assign to a given bug report, predict the bug fix time, estimate a priority level an so on. This thesis focuses on automated software bug report priority detection using state-of-the-art classification techniques. Widely successful transformer-based encoder classifiers are adapted to software domain via fine-tuning using open source datasets. Large Language Models (LLMs), on the other hand, are recently popularized transformer decoder networks specifically trained for text generation, which can be configured for priority class prediction. In order to accurately shape LLM output into desired for-

mat, Retrieval Augmented Generation (RAG) is used to condition the network to the downstream task and domain. Vector databases help store textual content in the bug reports according to cosine similarity and retrieve related instances during inference.

Keywords: Priority Detection, Contrastive Learning, LLMs, RAG, Vector Databases

ÖZ

YAZILIM HATALARINDA OTOMATİK ÖNCELİK TESPİTİ: ARTTIRILMIŞ VERİMLİLİK İÇİN KARŞILAŞTIRMALI ÖĞRENME, BÜYÜK DİL MODELLERİ VE VEKTÖR VERİTABANLARI İLE TRANSFORMATÖR TABANLI KODLAYICILAR ÜZERİNE KAPSAMLI BİR ÇALIŞMA

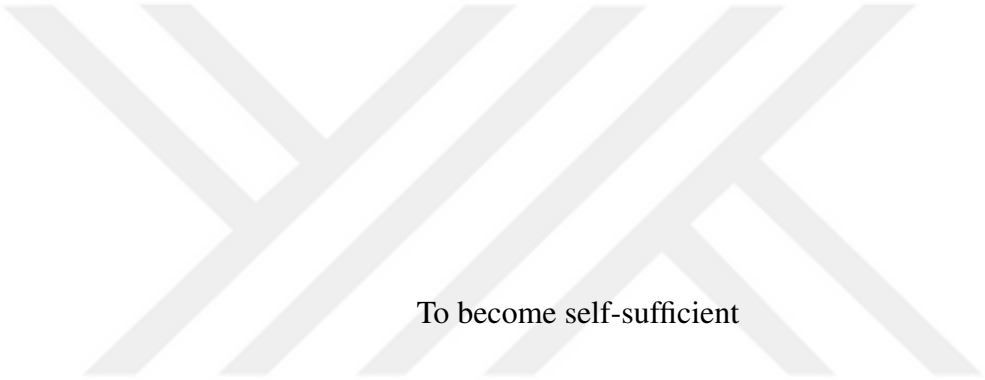
Yılmaz, Eyüp Halit
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. İsmail Hakkı Toroslu
Ortak Tez Yöneticisi: Dr. Öğr. Üyesi. Ömer Köksal

Ocak 2024 , 53 sayfa

Yazılım geliştirme süreçleri, emek ve zaman yatırımı gerektiren birçok zorluğu bünyesinde barındırmaktadır. Zamanla bu zorlukların üstesinden gelmek, yazılım geliştirme ve bakım süreçlerini otomatize edebilmek için birçok araç ve teknik geliştirilmiştir. Yazılım hata raporları, kullanıcılar veya geliştiriciler tarafından programların operasyonel hatalarını belgeleyen, genellikle kod parçacıkları ve hata mesajlarını içeren metinsel açıklamalardır. Bu raporlar daha sonra hataları düzeltmek için atanan geliştirici tarafından incelenir. Hata düzeltme süreçlerinin otomatikleştirilmesi, belirli bir hata raporuna atanacak en uygun geliştiricinin belirlenmesini, hata düzeltme süresinin tahmin edilmesini, öncelik düzeyinin tahmin edilmesini vb. içerir. Bu tez, en son teknoloji sınıflandırma tekniklerini kullanarak otomatik yazılım hata raporu öncelik tespitine odaklanmaktadır. Oldukça başarılı olan dönüştürücü tabanlı kodlayıcı sınıflandırıcılar, açık kaynak veri kümeleri kullanılarak ince ayar eğitimi kullanıla-

rak yazılım alanına uyarlanır. Öte yandan Büyük Dil Modelleri (LLM'ler), öncelik sınıfı tahmini için yapılandırılabilen, metin üretimi için özel olarak eğitilmiş, yakın zamanda popüler hale gelen dönüştürücü tabanlı kod çözücü ağlarıdır. LLM çıktısını istenen formatta doğru bir şekilde şekillendirmek için, ağı nihai sınıflandırma görevine ve etki alanına göre koşullandırmak için Bilgi İyileştirmeyle Geliştirilmiş Üretim (RAG) kullanılır. Vektör veritabanları, hata raporlarındaki metin içeriğinin kosinüs benzerliğine göre saklanmasına ve çıkarım sırasında ilgili örneklerin alınmasına yardımcı olur.

Anahtar Kelimeler: Öncelik Tespiti, Kıyaslamalı Öğrenme, Büyük Dil Modelleri, Bilgi İyileştirmeyle Geliştirilmiş Üretim, Vektör Veritabanları



To become self-sufficient

ACKNOWLEDGMENTS

I would like to thank my family for their love and support in my academic journey through which I encountered many obstacles and challenges. Their presence and guidance let me overcome those obstacles and challenges which helped me grow. I would like to give my thanks to my sister Beyza especially, who always made me feel special. My heartfelt gratitude goes to my sister Betül, who would understand me and inspire me under any circumstance and offer her help.

I also would like to thank my supervisor, Prof. Dr. İsmail Hakkı Toroslu without whose vision this work would not be possible. I would like to express my deepest appreciation to Assist. Prof. Dr. Ömer Köksal for his expertise in the software engineering field and sincere approach to me.

Finally, I gratefully admit my deepest appreciation for the "Autonymous" team and all its members. All the technical experience we earned by challenging ourselves and by trial and error grew stronger to a bond of the fellowship and countless bittersweet memories.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

List of Tables xiii

List of Figures xv

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation and Problem Definition 1

1.2 Contributions and Novelties 5

1.3 The Outline of the Thesis 6

2 ENABLING TECHNOLOGIES 7

2.1 Word Embeddings 7

2.2 Transformer Encoders and BERT 11

2.3 Large Language Models 13

2.4 Vector Databases 15

2.5 Retrieval Augmented Generation (RAG) 17

3 RELATED WORK ON AUTOMATED SOFTWARE BUG REPORT PROCESSING 19

3.1 Related Work 19

4 CONTRASTIVE LEARNING FOR TRANSFORMER ENCODERS 23

5 TRANSFORMER-BASED CLASSIFIER METHODS 27

6 DATA ACQUISITION AND ANALYSIS 31

6.1 Statistical Analysis 31

6.2 Experimental Design 35

7 RESULTS AND DISCUSSION 39

8 CONCLUSIONS 45

9 BIBLIOGRAPHY 47

List of Tables

TABLES

Table 6.1 Mozilla Dataset Difficulty Report 33

Table 6.2 App Reviews Dataset Difficulty Report 35

Table 6.3 Rotten Tomatoes Tomatometer Rating Difficulty Report 36

Table 7.1 Per-Class Classification Performances on Mozilla Core Dataset . . . 40

Table 7.2 Per-Class Classification Performances models on App Reviews Dataset 42

Table 7.3 Per-Class Classification Performances models on Rotten Tomatoes Tomatometer Rating Dataset 43

List of Figures

FIGURES

Figure 2.1 Illustration of the Continuous Bag-of-Words (CBOW) and Skip-Gram algorithms 9

Figure 2.2 Pretraining and fine-tuning stages of BERT. Illustration is taken from (1) 13

Figure 2.3 Encoder-decoder structure of the transformer network. Only the encoder network is utilized in text generation with a simple encoder layer. Illustration is taken from (2) 15

Figure 2.4 SBERT training procedure using Siamese networks. The model is trained to minimize the difference between sequence embedding vectors during training and cosine similarity is calculated during inference. Illustration is taken from (3) 16

Figure 4.1 The effect of the hyperparameter λ on the calculated loss values with the contrastive loss function 24

Figure 5.1 Framework for applying the chatbot for bug report priority detection 29

Figure 6.1 The distribution of the number of instances in each class for the Mozilla dataset 34

Figure 6.2 The distribution of the number of instances in each class for the App Reviews dataset 34

Figure 6.3 The distribution of the number of instances in each class for the Rotten Tomatoes dataset 37

LIST OF ABBREVIATIONS

ABBREVIATIONS	
BERT	Bidirectional Encoder Representations from Transformers
CNN	Convolutional Neural Network
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
RAG	Retrieval Augmented Generation
RLHF	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Network
S2R	Steps to Reproduce

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

Software development has become an integral part of many industrial applications that require any kind of automated logic to be deployed in the field. Although the requirements and specifications may vary, the general process of software development follow certain guidelines to help ensure consistency, reliability, and efficiency of the developed applications. An unavoidable part of software development process is unintended generation of bugs, failures of software implementations in syntax, logic or other modes of operation. The rapid proliferation of software applications has led to an exponential increase in bugs reported by developers, testers and users alike.

The ubiquity of digital systems in both industrial processes and daily activities has underscored the importance of software development, distribution, and maintenance. This widespread adoption has given rise to a burgeoning number of software bug reports, which meticulously document instances of system failure in various testing and production environments. These bug reports are systematically cataloged, complete with labels that denote bug priority, failure severity, assigned developer, implicated software components, and other pertinent information. These meticulously curated collections of bug reports serve as indispensable training datasets for automated classification algorithms, poised to revolutionize the efficiency of bug resolution processes.

Yet, the manual inspection and labeling of bug reports remain a laborious and time-consuming endeavor. An astonishing example reveals that labeling 7,401 bug reports required a staggering 10,884 individual inspections, spanning over 90 working days

(4). Furthermore, the process of developer assignment, often prone to human errors, can necessitate reassignment, a practice known as "bug tossing." Notably, in the open-source Eclipse project, it took an average of 40 days for initial developer assignment, with bug tossing extending the resolution timeline to an average of 100 days (5). The Mozilla project faced even greater challenges, with initial developer assignment averaging 180 days and bug tossing extending the resolution timeline to an average of 250 days (5). Therefore, there is a need for an efficient automated method that considers both contextual information from the bug reports as well as learned domain knowledge of software bug reports in order to accurately classify them. Yet, the manual inspection and labeling of bug reports remain a laborious and time-consuming endeavor. An astonishing example reveals that labeling 7,401 bug reports required a staggering 10,884 individual inspections, spanning over 90 working days (4). Furthermore, the process of developer assignment, often prone to human errors, can necessitate reassignment, a practice known as "bug tossing." Notably, in the open-source Eclipse project, it took an average of 40 days for initial developer assignment, with bug tossing extending the resolution timeline to an average of 100 days (5). The Mozilla project faced even greater challenges, with initial developer assignment averaging 180 days and bug tossing extending the resolution timeline to an average of 250 days (5). Therefore, there is a need for an efficient automated method that considers both contextual information from the bug reports as well as learned domain knowledge of software bug reports in order to accurately classify them. In recent years, there have been significant advancements in natural language processing (NLP) techniques applied to text analysis which can help automate several steps of the software development and maintenance processes (6; 7). The potential benefits of these NLP models in classifying software bugs are clear: they can accelerate the prioritization process, improve team productivity by directing resources towards high-impact issues, and ultimately lead to better quality software products.

Various practices for handling bugs has constituted a literature of sophisticated tracing systems, issue submit infrastructure and triaging techniques (5). As a result, there is a large and ever-increasing number of data collections containing software bug reports from community-driven open-source software projects and proposed algorithms to automate parts of the bug fixing pipeline. The target domain of software bug report

datasets vary according to the core problem they are engineered to solve; priority detection (8), automatic developer assignment (9), issue/feature classification are among the example use cases. The structure of the bug report depends on the downstream task but in general, there is a natural language text written by the user who experienced the bug describing the problem and giving instructions to replicate or reproduce the bug. Additional meta-data about the system specifications, software version, time and date information are also included in the reports occasionally.

Automated processing of software bug reports requires a combination of information extraction from the natural text component as well as sophisticated knowledge on the software domain to understand code snippets and error logs in the bug reports. Transformer-based deep learning models (1; 2) unlocked an era of highly capable natural language processing models that utilize the attention mechanism to assign contextual representation to input text. The models process textual input without domain restriction during the pretraining phase, which helps the underlying network to conceptualize real-world phenomena and store information in the network weights. Application of pretrained transformer-based models in automated bug report processing via fine-tuning is an emerging field with promising applications.

With emergence of successful context-aware vector representations of textual data, various needs in software development can be addressed via automated decision making mechanisms. Bug localization is one of such needs in software maintenance, which aims to identify the source code elements that are responsible for a given bug. However, existing bug localization techniques often rely on textual similarity between bug reports and source code, which may not capture the semantic relatedness between them. Ciborowska and Damevski (10) propose a novel approach to bug localization based on changesets, which are the sets of source code files that have been modified together in a version control system. They use BERT language encoder (1) to encode both bug reports and changesets into vector representations, and then compute the similarity between them using cosine distance. They also introduce several optimizations, including fine-tuning on software engineering data, applying a hashing technique to reduce the dimensionality of the vectors, and matching bug reports to individual changes within a changeset, to make BERT faster and more accurate for this task.

Another problem in bug-fixing pipeline is to retrieve the steps to reproduce (S2R) the same error encountered by the bug reporter in a different machine running the same software (11). Although it is critical that S2R is provided in detail to the developer, it is often overlooked and omitted by the reporters. Therefore, the natural text in the bug reports becomes the main source of information to extract S2R. Zhang et al. (12) utilize Reinforcement Learning and NLP techniques to train a model that can find the S2R through intelligent decision-making.

Recurrent Neural Networks (RNNs) combined with the Attention mechanism (2) constitute a competitive alternative in developer prediction task (13). The authors demonstrate that inclusion of bug report titles and descriptions increase the classification rank at k. They also examine the effect of the number of bug reports per developer with a thresholding approach and observe that more instances generally lead to higher performance.

A major problem in bug report priority detection is the imbalance in the number of samples in each priority class. Graph Neural Networks (GNN) provide a different modeling that can address the class imbalance (14). A heterogeneous graph representation that includes word-word, word-document and document-document connections is constructed from bug reports. Then, a 2-layer graph network learns to predict priority labels via a weighted supervised loss function. The weights are calculated according to the number of samples per class in the training data. Experimental results demonstrate the improvement of per-class prediction performance with help of GNNs over conventional learning algorithms even in existence of severe class imbalance.

Wang et al. propose CLeBPI (Contrastive Learning for Bug Priority Inference) as a novel approach to automatically assign priority levels to bug reports based on their semantic structure (15). The approach leverages a pre-trained language model and contrastive learning to learn contextual representations of bug reports without manual feature engineering. Experimental results demonstrate the superior performance of the CLeBPI algorithm to the existing methods on a public dataset, outperforming the baseline by a large margin.

CLeBPI introduces several improvements to the training process. The vanilla BERT encoder is modified with residual connections and dynamic masking (16) in domain-

specific pre-training. Bug report titles and descriptions are concatenated to obtain pre-training data. In fine-tuning, cosine distance between positive and negative pairs of bug reports are used for contrastive learning. Minor modifications such as token deletion, replacement or shuffling, which does not change semantics significantly, are used for positive pair construction.

Priority detection for software bug reports is a fundamental step in automated management of the bug-fixing pipeline. Although there are pioneering studies in the literature, the landscape of predictive algorithms with help of deep learning is rapidly growing and there is a lack of rigorous analysis on how newly emerging paradigms apply to this domain. This thesis focuses on the application of transformer-based deep neural networks for software bug report priority detection.

1.2 Contributions and Novelties

Our contributions are as follows:

- We fine-tune a transformer-based pretrained encoder (BERT) as a classifier and report its performance.
- We propose a contrastive learning procedure for the encoder network by defining a custom loss function.
- We propose a regularization term that helps reduce the negative effects of training data imbalance during fine-tuning.
- We apply LLMs, transformer decoder generative models, to the downstream task of software bug report priority detection with few-shot prompting.
- We report the effects of RAG (Retrieval Augmented Generation) to LLM classification performance.
- We utilize vector databases for similarity-based retrieval during RAG process.

By incorporating many different predictive models from a wide range of paradigms, we explore how state-of-the-art NLP techniques can be exploited for software bug

report priority detection. In order to analyze the generalization capabilities of the explored methods, we experiment with different settings and domains that share a similar problem structure.

1.3 The Outline of the Thesis

The outline of this thesis is as follows: In chapter 2, we present the enabling technologies that paved the way for the development and the application of the techniques described in this thesis. In chapter 3, we present the related work where a subset of the studies in the field of software bug report priority detection are referenced. In chapter 4, we describe our developed solution that capitalizes on the unique structure of the problem with a specifically tailored loss function. Chapter 5 describes the application process of LLMs and potential improvements by RAG systems in the domain of this work. We document the data acquisiton procedure and statistical analyses in chapter 6 and report the results along with our discussions in chapter 7. Chapter 8 provides the conclusions of the thesis.

CHAPTER 2

ENABLING TECHNOLOGIES

We present the background information about substantial technical advancements that made our work possible. Natural Language Processing (NLP) is the major field of artificial intelligence and deep learning literature that provides the ability to conduct informed automated decision making using textual data. Therefore, software bug report priority detection benefits substantially from developments in this particular branch which makes it necessary to establish a sound understanding of the underlying principles of the methods explored.

2.1 Word Embeddings

One major challenge in NLP field is to represent textual data in numerical form, which can be easily processed by computers in machine learning algorithms. In the conventional information retrieval systems, a term frequency-inverse document frequency (17) model is constructed from a collection of documents where each term is represented with a vector of size equal to the size of the document collection. For a given term t, the value at the index d of the representation vector is computed using a data collection D with formulae 2.1, 2.2, and 2.3.

$$\text{Tf-Idf}(t, d, D) = \text{Tf}(t, d) \times \text{Idf}(t, D) \tag{2.1}$$

$$\text{Tf}(t, d) = \frac{\text{number of times term } t \text{ appears in document } d}{\text{total number of terms in document } d} \tag{2.2}$$

$$\text{Idf}(t, D) = \log\left(\frac{\text{total number of documents in } D}{\text{number of documents containing term } t + 1}\right) \tag{2.3}$$

The motivation behind such a formulation comes from a fundamental observation about natural languages, named Zipf's Law (18). It is an empirical law which states that frequency of a word in a language is inversely proportional to the rank of the word. In essence, the most frequent words in a language carry little meaning, their use are mostly for grammatical reasons, and they are used in many places regardless of context. Whereas rarely occurring words, being associated with specific concepts, carry specific meanings usually dependent on the content and the topic of the document. Tf-Idf vector representation combines the term occurrence frequency with inverse document frequency (a measure of how common a given term is) to construct the corresponding vector.

Although quite useful in retrieval tasks and statistical machine learning models, there are some limitations to Tf-Idf vector representation. The size of the vector depends on how many documents are in a given collection and it consequently varies for different applications. The constructed vectors are usually very sparse, making it harder to run computation on even simple setups. Additionally, there are no semantic relations between word representations dependent on co-occurrence.

To improve the modeling capabilities and address these issues, researchers have proposed neural word embedding models that use a large corpus of text data as input and produce continuous, fixed-size vector representations for each unique word appearing in the corpus. Two well-known examples of such word embedding models are Word2Vec (19) and GloVe (Global Vectors) (20). Both these methods have shown promising results in various NLP tasks, including text classification, sentiment analysis, question answering, among others. In this section, we will provide a brief overview of Word2Vec and GloVe models.

Word2vec is a self-supervised learning model that learns word embeddings from large amounts of text data. The underlying idea behind this model is to capture the contextual information around each word and represent it as a vector in high-dimensional space, such that words with similar meanings or context are closer together than those having different meaning or context. Word2Vec models typically

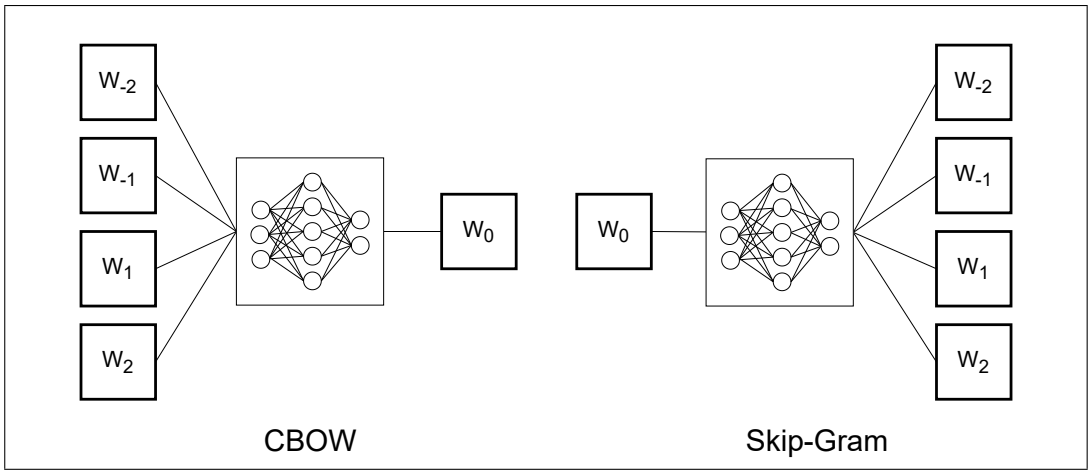


Figure 2.1: Illustration of the Continuous Bag-of-Words (CBOW) and Skip-Gram algorithms

employ two architectures: Continuous Bag-of-Words (CBOW) and Skip-Gram, illustrated in Figure 2.1. In the CBOW architecture, given the surrounding words of a target word w_{-2}, w_{-1}, w_1, w_2, the model tries to predict the target word w_0 based on co-occurrence patterns in the input corpus. This is done by training a neural network where network layers learn the embedding vectors for all unique words appearing in the dataset. In order to achieve this task, the model needs to compress enough information about words and co-occurrences which leads to a consistent representation in a vector space.

On the other hand, Skip-Gram architecture tries to capture the distributional similarity between words by training a model that predicts the neighboring words w_{-2}, w_{-1}, w_1, w_2 given a target word w_0. In this case, the neural network learns an embedding vector for w_0 that can help extract the words in the context frequently associated with the target word. During testing phase, the Skip-Gram architecture uses these pre-trained word embeddings to compute similarity scores between pairs of input words based on their distributional patterns in the corpus.

The resulting word vectors exhibit behaviors that are reminiscent of semantic relations between the corresponding vectors. Results of simple linear operations between the word vectors are in close proximity of the word vector for the suitable target word. A commonly cited example is $vector(\text{"}King\text{"}) - vector(\text{"}Man\text{"}) +$

$vector("Woman") \approx vector("Queen")$ (19). Word vectors capture the context shift between the words "man" and "woman" which corresponds to a close approximation of the contextual difference between the words "king" and "queen", as expected.

The use of neural networks in construction of word vectors is an innovative approach but it has a number of drawbacks. The training process is prone to errors arising from the stochastic nature of training and model size is a limiting factor for modeling capacity. GloVe (Global Vectors) (20) is another popular word embedding model proposed to improve upon the initial word vector construction method with use of statistical co-occurrence calculations. GloVe learns continuous vector representations for each unique word by optimizing a log-bilinear negative sampling objective function over large amounts of text data.

The basic idea behind this approach is to capture the local context around words and their global relationships with other words in the corpus, using word co-occurrence frequencies instead of the entire sparse matrix. In GloVe model, each unique word appearing in the dataset is represented as a vector in high-dimensional space with a fixed dimension size. The modeling performance of the word vectors depend on various hyper-parameters such as context window length and minimum frequency threshold. These vectors are learned by minimizing the loss function given in Equation 2.4 (20). The resulting logbilinear regression model is optimized using techniques like stochastic gradient descent (SGD) for optimization purposes.

$$\hat{J} = \sum_{i,j} f(X_{ij}) w_i^T \tilde{w}_j - \log(X_i) \tag{2.4}$$

Word embeddings solved a critical problem in NLP, transforming text into a format understandable and suitable for processing by computers. However, their application was limited by the fact that there is no straightforward way of obtaining an encoding of a text snippet consisting of multiple words that incorporated contextual information. A common approach it taking the average of all the vectors making up a sentence or word sequence and treat it as the overall encoding. This is an order-agnostic modeling of language and can be treated as a Bag-of-Words approach. Alternatively, Recurrent Neural Networks (RNNs) or Long Short-Term Memory (LSTM) models

process a sequence of words one by one to create an embedding by updating a randomly initialized vector at each computation step using the word embeddings.

Static vector embeddings for words with different meanings in different contexts get assigned the same vector calculated in the embedding construction phase. This prohibits a dynamic representation for words that reflect and change according to the context. The context window used in CBOW and Skip-Gram approaches are limited to only a few words which makes it hard to represent long-range dependencies. As a solution to these limitations, contextual embeddings emerged.

2.2 Transformer Encoders and BERT

BERT (Bidirectional Encoder Representations from Transformers) (1) is an application of the encoder architecture introduced in the Transformer network (2) with a few modifications. It employs WordPiece tokenization (21) for transforming the textual input to smaller chunks, sometimes smaller than a word, and assign them randomly initialized vectors. Later, a position embedding vector is added to these vectors to incorporate the information about the word order in the input. During training, these vectors can be updated via the gradient signal from the loss function.

In essence, BERT makes use of transformer encoders that work bidirectionally, to construct a representation of words using the context from both directions. This unique modeling enables incorporation of knowledge about the context from accompanying words for the given token. Additionally, the model not only creates representations for tokens, but also undergo a computation that yield a dynamic representation for the text sequence, allowing for more accurate applications.

A critical component in the underlying model is the use of the Attention mechanism (2). It allows the model to amplify the connections between a target word and related words in the sequence to focus on contextually related parts during training and prediction, which helps it construct a better understanding than traditional models that simply process words or word pairs in isolation. The core idea behind the Attention mechanism is to enable the model to adjust the weights assigned to each input element depending upon their importance in making predictions.

The attention mechanism works by computing a weighted sum of inputs where the weights are determined based on the relevance or significance of that particular input with respect to predicting an output. It involves three primary components, namely a query vector (Q), key vectors(K) and value vectors(V). The Q-vector represents the current token, or word, that the model processes at a certain time step, whereas K-vectors represent all tokens in the input from which relevant information is needed for the prediction. These vectors are then used to compute the representation vectors as described in the algorithm below.

Algorithm 1 Attention Mechanism

Input: Query Q, Key K, Value V

Output: Weighted sum of values Attention(Q, K, V)

function ATTENTION(Q, K, V)

    Score $\leftarrow$ softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ {Compute attention scores}

    Attention $\leftarrow$ Score $\cdot\, V$ {Weighted sum of values}

    return Attention

end function=0

Although the algorithmic complexity of the attention algorithm is $O(n^2)$ in terms of the sequence length n, it can be adjusted with attention masks in practice which helps ignore attention computation over unused tokens. The scaled dot-product attention mechanism helps the model learn important dependencies between frequently co-occurring tokens and incorporate this information in the contextual vector representations.

The training procedure for the BERT model has several key components, such as pre-training the model on a large corpus consisting of general purpose texts using masked language modeling (MLM). The MLM task requires the model to predict the words randomly masked out in the input sequence. To achieve this, the model has to learn which words in the context are most relevant when predicting the masked word words, much like the CBOW technique. There is an additional next sentence prediction task which asks whether a given two text sequences are immediate successors of each other as a single prediction score. However, this task is found to reduce the language understanding capability of the model and better be omitted (16).

The pre-trained models serve as the foundation for fine-tuning on specific tasks, illustrated in Figure 2.2, such as sentiment analysis, named entity recognition (NER), and text classification with higher accuracy and transfer learning.

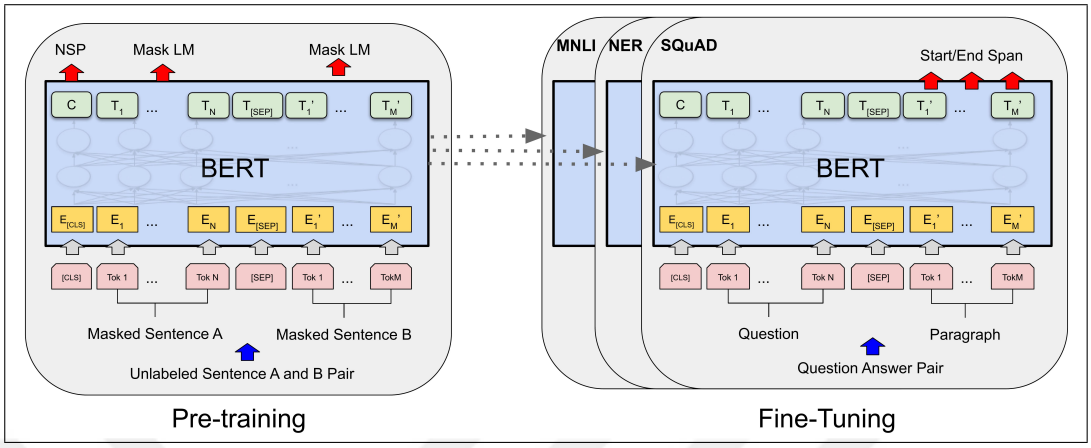


Figure 2.2: Pretraining and fine-tuning stages of BERT. Illustration is taken from (1)

With the newly introduced pretraining and fine-tuning paradigm, BERT opened a new chapter in NLP by allowing it to learn from large amounts of data efficiently while retaining the ability for high-level understanding across diverse tasks, making it an essential tool in many downstream applications like classification and ranking.

2.3 Large Language Models

Large Language Models (LLMs), popularized after the release of ChatGPT[1], are a family of neural networks based on the transformer decoder architecture, given in Figure 2.3, that are used for text generation tasks. After the introduction of highly capable text generation models using transformer decoders, first in Generative Pretrained Transformers (GPT) (22) and in successive studies GPT2 (23) and GPT3 (24), there appeared a substantial research interest for applications that improve and enhance the quality of the generated text and steer the direction towards a desired structure.

Text generation using transformer decoders involves the use of multi-head attention mechanism along with dedicated decoding algorithms and sampling processes. The input text, usually initiated with a special start-of-text token, is converted into vector

[1] https://chat.openai.com/

form by using a simple lookup embedding table. Then, stacks of decoder blocks consisting of multiple attention heads and residual connections to the layer normalization and feed forward network are utilized to compute the probability scores for the next token over the entire vocabulary. The probability vector is utilized by the decoding algorithm for sampling the next token which is later added to the input text. In the next step, the combination of the input text and model generation is utilized to calculate the probability scores which makes generation of subsequent tokens dependent on the intermediate model outputs. This is often called an auto regressive language modeling as opposed to the bidirectional language modeling employed in BERT.

During training, the model needs to develop a coherent understanding of the internal dynamics that make up a certain text sample to accurately predict what comes next. This enables a compression and storage of information about the training data, which is often collected from online sources and contain general knowledge about an unconstrained range of domains. At the end of the pretraining process, the resulting model can be used as a foundation for adaptation to a specific domain with fine-tuning, or can be conditioned to generate domain-related output via prompting (22; 23).

A consistent observation in transformer-based models is a positive correlation between the number of parameters a language model has, or the parameter size, and the downstream generation quality (25). This resulted in a generation of models that are highly capable at a wide range of tasks thanks to an increasingly higher model size, reaching billions (26) or even trillions (27) in number of parameters. However, the computational cost for training and deployment of these models increased as well, creating a barrier for entry for teams with limited development budgets. Moreover, the pretraining procedure used for training these models, the next token prediction task, does not enable an immediate deployment for many use cases. With the introduction of InstructGPT (28) and use of Reinforcement Learning from Human Feedback (RLHF), the medium of interaction with these models transferred to simple conversation, creating a more natural environment for wide-range use.

The immediate advantage transformer networks brought about is the ability to conduct parallel computing in training time with attention masking and batched processing of several inputs. It was not possible to deploy large scale training runs with

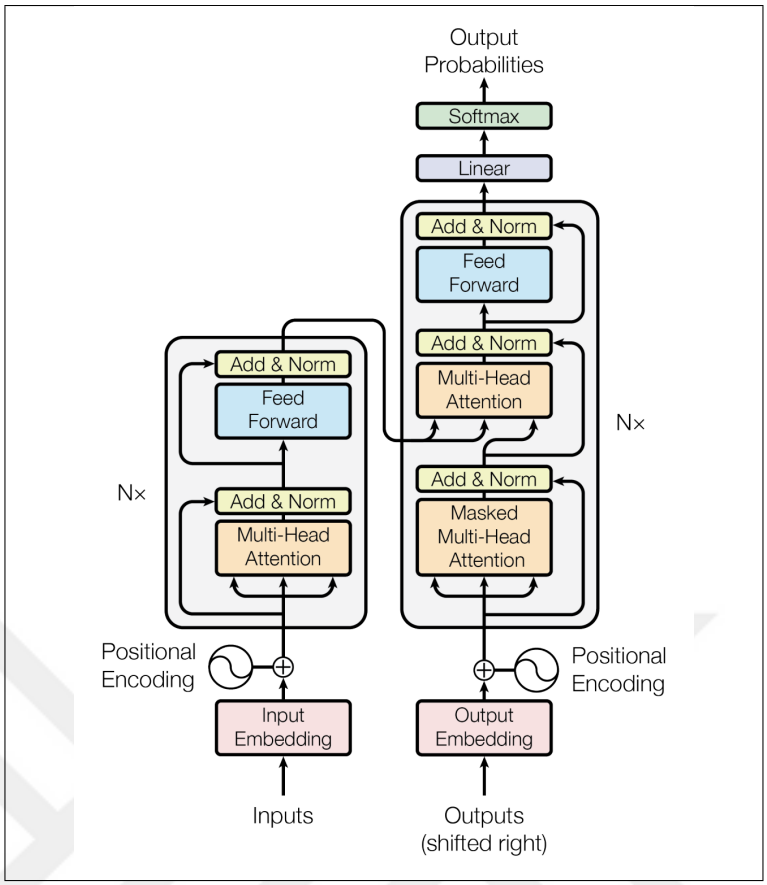


Figure 2.3: Encoder-decoder structure of the transformer network. Only the encoder network is utilized in text generation with a simple encoder layer. Illustration is taken from (2)

earlier methods such as RNNs (29) and LSTMs (30) since their sequential structure prohibit a parallel training procedure. Transformer decoders, on the other hand, make it possible to deploy many computation units, often GPUs, in parallel to train over billions of tokens in feasible time budgets.

2.4 Vector Databases

Popularization of neural network applications over different modalities of data, such as language, images, audio and video, established a common ground for high dimensional representations to be connected, both in terms of algebraic operations and semantic relations. Carefully constructed learning methods enabled coupling textual

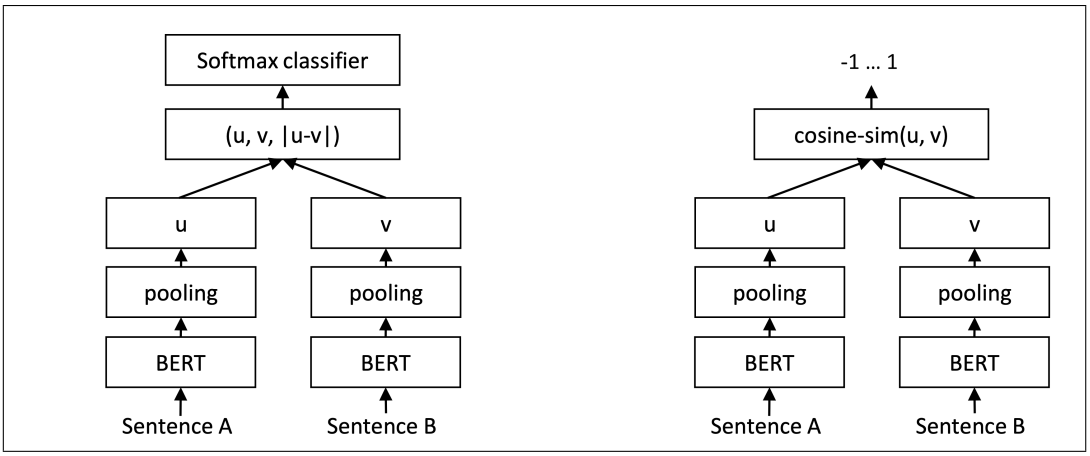


Figure 2.4: SBERT training procedure using Siamese networks. The model is trained to minimize the difference between sequence embedding vectors during training and cosine similarity is calculated during inference. Illustration is taken from (3)

descriptions and images (31) which can later be used for generating images that do not exist in the training set (32; 33).

Vector databases propose a new data storage paradigm that operates according to the principle that high dimensional vector representations for all modalities of data, text, audio, image, video and any other embedding, should be searchable via similarity measurements defined over vectors. This way, storage and retrieval operations can be improved with neural representations that discover hidden features during training.

A prerequisite for a neural embedding model in a vector database is to generate a semantically coherent cluster of embeddings for similar samples of data. For text inputs, a transformer encoder trained specifically to minimize a distance metric, e.g. cosine distance, over pairs of semantically similar texts would be used. To this end, researchers exploit Siamese networks for training over pairs of text sequences using metric learning and fine-tune a BERT model for sentence encoding (3). Figure 2.4 illustrates the training objective and inference structure of SBERT (3). Sentence A and Sentence B are encoded with Siamese BERT networks, networks that share their weights, to create embedding vectors in the same representation space.

The novelty of SBERT training procedure is the construction of training pairs, where similar text sequences are labeled as positive pairs and dissimilar text sequences are

labeled as negative pairs. Metric learning then assures the underlying embedding space is clustered into semantically meaningful segments and cosine similarity in test time yields a measure of how semantically related two inputs are.

Using SBERT as the encoder in a vector database provides an advantage in retrieval phase by searching over the entire training data collection according to documents' cosine similarity to the user query. General world knowledge obtained by the SBERT model during pretraining helps understand and construct semantic relations between retrieved samples and the input query that are otherwise would only be possible at a syntactic level with keyword-based retrieval algorithms.

2.5 Retrieval Augmented Generation (RAG)

LLM generated text has a simple objective, predicting the next word or token that comes after a number of given words making up the context. Many language models learn to mimic the contextual structure of sentences and text data and generate similar outputs at test time. However, their objective may not ensure that the information contained in the generated text is always correct, leading to confident yet factually wrong model outputs, which are often called hallucinations (34; 35; 36).

In order to reduce hallucinations at model output and ground the informative content in generated text on facts, Retrieval Augmented Generation (RAG) is proposed (37) as a middle step in generation procedure to provide more context to the LLM to essentially condition it to steer towards a factually correct output. RAG involves modifying the user prompt to construct a systematic structure that presents the factual information extracted from retrieved documents to the model.

It is observed that language models improve their performances at text generation tasks with help of few-shot demonstrations of expected outputs (24). Retrieved documents based on the user query, or a test sample for classification tasks, can be utilized to construct these few-shot demonstrations with ground truth labels. The limiting factor in this paradigm is the length of the context window, the maximum number of tokens that can be given to a certain LLM. System prompts that define a character and output style for the model, RAG-generated few-shot demonstrations and the test

sample all count towards the context window limit, which makes it possible to use only a certain number of samples in the prompt.

The inference time for each test sample is determined by the complexity of the RAG step, the size of the LLM and its quantization level, and the hardware acceleration solutions adopted in the pipeline. Using vector databases in the RAG step adds a computational overhead to the process but it is usually lighter than LLM generation since it is not a recurrent operation and the encoder network is orders of magnitude smaller than the LLM decoder.

CHAPTER 3

RELATED WORK ON AUTOMATED SOFTWARE BUG REPORT PROCESSING

The main challenge in processing a software bug report lies in the content of the report, often a mixture of natural language text and technical details, such as code snippets and error logs. Recent advancements in natural language processing (NLP) techniques provide a way forward, essentially compressing information about a domain in the weights of a neural network. With NLP models being able to analyze text data and learn representative deep features, it is now possible to exploit this information to classify bug reports into priority categories based on contextual information and semantic relationships between different bug reports. This has the potential to accelerate the prioritization process, improve team productivity by directing resources towards high-impact issues, and ultimately lead to better quality software products.

3.1 Related Work

Automatic software bug report assessment features a range of tasks, including priority detection, developer classification, fault prediction and many more. Researchers proposed (7) an ML-based approach to predict fault types using textual bug reports from 70+ projects with up to 69% macro average F1 score, and established a baseline for non-expert human performance in debugging tool selection. The models showed inter-project transferability and could aid inexperienced developers by saving time and resources during the debugging process. In another study, a comparative analysis (38) is conducted on three Multi-Objective Evolutionary Algorithms (MOEAs) - NSGA-II, NSGA-III and MOEA/D - to automatically generate finite state machines

(FSMs) based on bug reports written in natural language. The results show that the proposed approach significantly outperforms the baseline tool KLFA for real-world software programs with a greater number of bugs detected by NSGA-II, while also controlling model size and avoiding local optima through using all three objectives during evolution (size, over-approximation and under-approximation). In (39), Bidirectional Encoder Representation from Transformers (BERT) (1) is adapted to software security domain, outperforming baseline models on both large and small sample datasets; detecting 56 hidden vulnerabilities through deployment in Mozilla and RedHat projects. This demonstrates the practical value of deployed NLP models for automated software improvement processes. MaCa is proposed (40) as a method that leverages machine learning techniques for accurate classification of action words within bug reports associated with mobile apps, which significantly improves the success rates of existing approaches such as Yakusu and ReCDroid in reproducing bugs.

In many cases, data imbalance and data scarcity hinders downstream model performance. Various studies addressed this issue via different techniques such as oversampling, a well-known approach to generate synthetic data, or resampling to adjust the distribution of samples in the training set. Priority prediction using comment intensiveness features (8) and SMOTE-based (41) data balancing scheme on three open-source projects; Eclipse, Mozilla, and OpenOffice achieves 0.6078 Precision, 0.4927 Recall, 0.4465 F1-score, and 0.7836 Accuracy with five classification models: Multinomial Naïve Bayes, Support Vector Machines, Random Forest, Extra Trees, and eXtreme Gradient Boosting. The results show that CIS-SMOTE-based models outperform two advanced approaches, eApp, and cPur in terms of all performance measures. CLeBPI (Contrastive Learning for Bug Priority Inference) (15) leverages pre-trained language models and contrastive learning to tackle label imbalance problem by automatically learning contextual representations of bug reports without manual feature engineering, and show its effectiveness with an improvement in weighted average F1 score compared to four baseline approaches on a public dataset. Experiments with six state-of-the-art rebalancing methods (42) combined with five popular classification algorithms for security bug reports (SBR) predictions and demonstrate the potential of these techniques to improve performance by 267% in the best case and 75% on average, based on a comparative study using real-world projects.

Adaptive ranking (43) is used to profile developers through their commit messages using various features including domain knowledge encoding and learning-to-rank technique for recommending suitable developers to handle bug reports. The model was evaluated on around 22,000 bug reports from four large scale open-source Java projects and it outperforms state-of-the-art methods in terms of the percentage of correctly recommended top developers within the rankings.

In (44), the authors propose an automatic approach to predict bug report priorities using convolutional neural networks (CNNs). They apply natural language processing techniques and software engineering domain-specific emotion analysis on textual information of bug reports, converting them into vectors based on their syntactic and semantic relationship. The proposed method outperforms baseline approaches by more than 24% in terms of average F1-score across multiple open source projects. For a similar problem, bug report severity detection, a deep neural network-based automatic approach is proposed (45) to predict bug report severity levels, which outperforms state-of-the-art approaches by an average F-measure improvement of 7.90%. The study employs natural language processing techniques for text preprocessing and assigns emotion scores before passing the constructed vector and score through a deep neural network classifier for prediction purposes.

The existing body of literature, while insightful, reveals room for substantial improvement in the realm of automated bug report processing. These studies have historically focused on distinct facets of the bug resolution process, which can be broadly categorized into the following areas:

- **Developer Prediction:** This task focuses on predicting the most suitable developer to address a given bug report. By automating the assignment process, a developer prediction algorithm can relax time requirements of the bug fixing process.
- **Bug Priority Detection:** In this category, researchers endeavor to automate the assessment of bug report priorities, ensuring that critical issues receive prompt attention.
- **Bug Severity Detection:** Researchers in this field strive to develop automated

mechanisms for identifying the severity of reported bugs, aiding in triage and allocation of resources.

- **Bug Fix Time Requirement Detection:** This area of study seeks to predict the time required to resolve a bug, facilitating better project management and resource allocation.

While these research endeavors have undeniably contributed to the automation of bug report processing, this thesis focuses on the classification of bug report priority by introducing a novel perspective and experimenting with state-of-the-art predictive technologies. It is important to note that bug report priority classification is distinctive in its own right. Unlike many conventional classification tasks, the priority levels of bug reports are inherently interrelated, with each level signifying a specific degree of urgency or impact. In this context, our research explores the integration of state-of-the-art language models, such as BERT, and innovative contrastive learning paradigms to enhance the accuracy of bug report priority classification. We introduce a loss function that accounts for data imbalance and classification error magnitude, addressing the unique challenges posed by this classification task.

The landscape of deep neural networks for text classification is vast and rapidly changing with new models entering the scene each day. The question of which paradigm is the best suitable for an efficient automated priority detection method is still standing. Adoption of the widely successful transformer architecture into software bug report priority detection requires experimentation of different paradigms with various contemporary deep learning techniques, such as transformer encoder classifiers, vector databases, Large Language Models and Retrieval Augmented Generation (RAG). We present a meticulous analysis on the performance of models from different paradigms.

CHAPTER 4

CONTRASTIVE LEARNING FOR TRANSFORMER ENCODERS

Software bug report priority detection task has a unique feature in its formulation as a classification problem that can be exploited with a dedicated learning procedure. Each software bug report is labeled with a priority level, i.e. P1, P2, P3, P4, or P5, which indicate an ordering between the bug reports in terms of priority. In a deep representation learned by a neural network, P1 and P2 labeled bug reports would be expected to map to closer embeddings to each other than P1 and P5 labeled bug reports. In order to enforce such a modeling, we propose a contrastive learning loss to incorporate this order information into the training process.

The essential observation is as follows: During training, when the model prediction class for a given bug report differs greatly than the ground truth label, for instance model prediction is P1 and the ground truth label is P5, the loss function should generate a greater error signal than a closer prediction, e.g. when the model prediction is P1 and the ground truth label is P2. This distinction is not present in the commonly used cross-entropy loss where each class is considered independently. In the case that the classes are dependent with each class representing different levels of priority, contrastive learning with a specifically tailored loss function would regularize the embedding space with respect to class similarity. So, we modify the training loss function as given in 4.1, 4.2, and 4.3.

$$\text{Contrastive Loss} = \text{Cross-Entropy Loss} + \lambda \times \text{Mean Squared Error} \tag{4.1}$$

$$\text{Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) \tag{4.2}$$

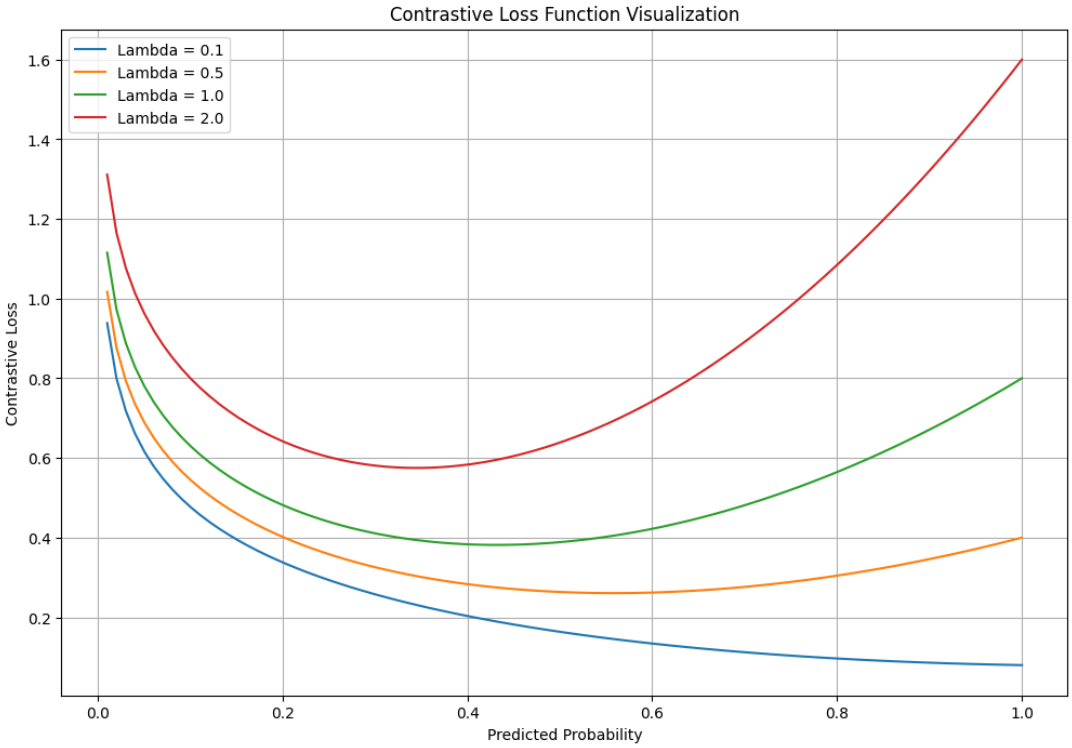


Figure 4.1: The effect of the hyperparameter λ on the calculated loss values with the contrastive loss function

$$\text{Mean Squared Error} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{4.3}$$

where N is the number of classes, y_i is the ground truth probability for class i, $\hat{y}_i$ is the predicted probability for class i, and λ is a hyperparameter that controls the weight of the mean squared error term relative to the cross-entropy loss. Essentially, we propose to regulate the model with mean squared error (MSE) as the additional regulatory term to the cross-entropy loss function. The effect of the hyperparameter λ is illustrated in Figure 4.1, we set $\lambda = 0.5$ in our experiments.

MSE is a widely adopted loss function that generates a strong gradient signal for regularization. The square of the difference between the ground truth label and the prediction ensures a non-negative loss value in any case with a parabolic increase with respect to the difference. To minimize this loss, the trained network should bring closer sequence embeddings of samples with similar priority labels and increase

the distance between embeddings of different class samples in the high dimensional representation space.

The contrastive loss function aims to combine the best of two worlds, the cross entropy term optimizes the classification performance while the mean squared error term regularizes the learning procedure with respect to the distance between the predicted class and the ground truth. The adjustable weight of the regularization term, λ, creates a trade-off between the cross-entropy loss and MSE to determine how much the training procedure prioritizes classification objective over the regularization objective.

Contrastive loss diverges from regular classification losses when the problem structure enables the definition of an ordering between classes. Such problems are similar in nature to ranking problems which are addressed with regression models where the last layer of the network consists of only a single neuron. Model output consists of a single real number, instead of a vector. The resulting number is treated as a prediction score, and the difference between the ground score and the prediction score constitutes the basis for loss calculation. However, regression models are not directly applicable in software bug report priority detection because the priority classes indicate discrete levels of priority, labeled by human annotators. So, the contrastive loss creates a modeling for classification tasks that share a similar structure with regression problems but with discrete levels.

Another serious problem in software bug report priority detection is severe imbalance in training data. We modify the contrastive loss function with an additional balancing term that exaggerates the loss values for poorly represented classes. We calculate the class distributions beforehand on the training dataset and use the percentage values for a weighted average of the log-likelihood predictions as given in 4.4.

$$\text{Imbalance Loss} = \frac{1}{N}\sum_{i=1}^{N} w_j \cdot y_i \cdot log(p(y^{(i)}|x)) \tag{4.4}$$

Here, w_j corresponds to one minus the percentage class weight calculated from the training dataset, to ensure a negative correlation between the class size and calculated loss value. We apply a mask to filter out correct predictions in every batch and apply

the imbalance term only to wrong predictions. Essentially, the proposed learning method amplifies the learning signal for instances from rare classes when the model makes mistakes. The total loss then becomes a simple addition of the contrastive loss and the imbalance loss, given in 4.5.

$$\text{Total Loss} = \text{Contrastive Loss} + \text{Imbalance Loss} \tag{4.5}$$

The imbalance loss term becomes relevant only when there is a substantial difference between the number of samples in different classes, as often seen in software bug report priority detection. It converges to a regular classification loss, albeit multiplied with a constant scalar, in the case that the training dataset is perfectly balanced.

CHAPTER 5

TRANSFORMER-BASED CLASSIFIER METHODS

The technical capabilities of neural language models have increased quite remarkably in many downstream applications. With the release of ChatGPT[1] and its open source counterparts, such as LLaMA (46), Llama 2 (47), and Mistral7B (48), the interface for human interaction with large language models (LLMs) transformed to natural conversation. The method that enabled this transformation is commonly referred to as instruction tuning, or instruction fine-tuning. Reinforcement Learning from Human Feedback (RLHF) (28) provides a preference-based learning procedure in contrast to regular self-supervised fine-tuning for controllable instruction fine-tuning. RLHF is a human-in-the-loop alternative in settings with no training data available.

Ubiquitous data generation capabilities of LLMs triggered an exponential growth in the number of instruction fine-tuned models. Mistral7B (48), OpenChat (49) and many others enabled general purpose chatbot applications to be run locally in consumer-grade hardware. The training domain of these models are usually not restricted, i.e. they are general purpose chat models trained with scraped data from online sources.

Despite their impressive capabilities, LLMs still face some challenges that need to be addressed before they can effectively support bug report priority detection tasks. In order to ensure that model outputs follow a desired format and avoid harmful or dishonest outputs, researchers deal with what is called the alignment problem. As a result, many techniques emerge that aim to tailor model response according to the needs of the downstream task. Prompt engineering (50; 51) is a method that tries to maximize the occurrence of the desired tokens at model output by changing the

[1] https://chat.openai.com/

structure of the input to fit certain criteria preferred by the model. These criteria might be set by the pretraining data, hyperparameters of text generation, and other factors that are not readily available for examination. Therefore, each downstream task and application needs a dedicated engineering effort to increase performance.

An emergent problem in LLM applications is hallucination, i.e. the tendency of the model to generate confident yet incorrect factual outputs. To mitigate this issue, Retrieval Augmented Generation (RAG) (37) adds a retrieval component between the user input and the LLM (52). RAG first applies a search using the user query and provides the retrieved information to LLM to assist the generation process. Effectively, RAG constitutes a means to reduce hallucination (53) by modifying the LLM prompt to include factual information and letting the model depend on the retrieved documents.

With the increasing embedding capabilities of neural models, the idea of transforming a storage of documents to a storage of embeddings became feasible. Vector databases, such as ChromaDB[2], are developed to provide a storage medium that operates on the semantic relationship between vector embeddings according to the contents of the data used to generate the embedding. For natural language inputs, a language model first generates a sequence embedding for the given input, such as a document, then the vector database stores the document based on a similarity measure of its embedding and existing embeddings in the database. The input data can be in any form acceptable by the embedding model, such as audio, image, or text. This provides an advantage in the retrieval step where one can apply a query to the system in any form available.

We harness these technologies in software bug report priority detection with certain modifications. We apply few-shot inference with $k = 15$ demonstrations to the LLM in the system prompt to increase model performance by conditioning the generation process (24). Figure 5.1 illustrates the framework we use for priority detection. We use OpenChat (49) in our experiments.

The illustrated framework depicts few-shot LLM generation in its vanilla form, that is, the demonstration samples are drawn from the training set according to a uniform random distribution. At each inference step, random sampling generates a new

[2] https://www.trychroma.com/

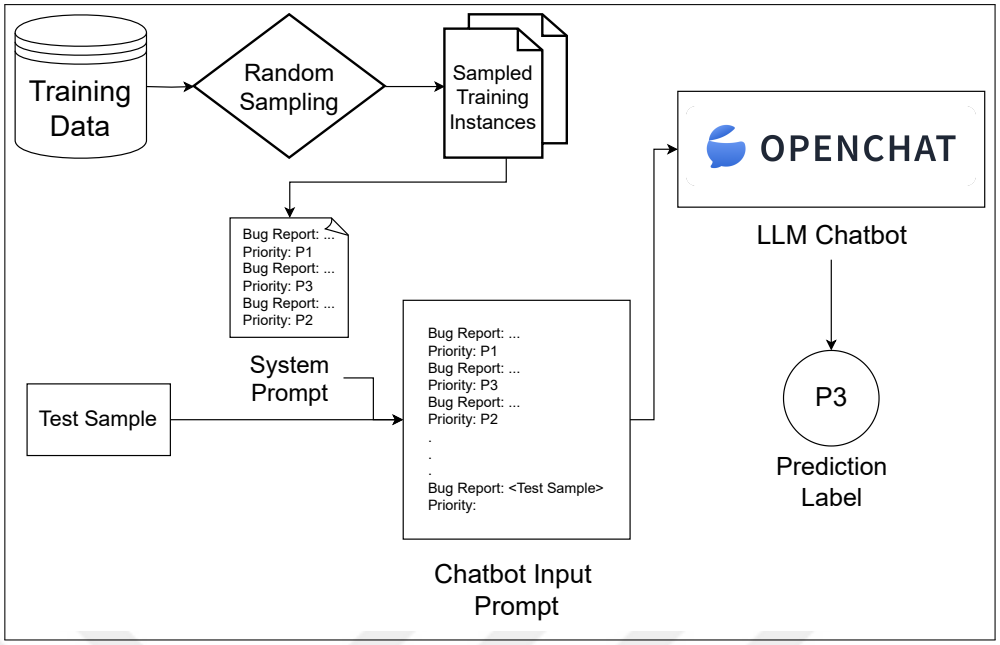


Figure 5.1: Framework for applying the chatbot for bug report priority detection

prompt to variate the conditioning and eliminate any potential bias. However, the class distributions in the training data may introduce a bias where dominant classes would appear more in the system prompt.

The output of the LLM is decoded into text via greedy sampling as given in 5.1. The generated token at each step is the most probable in the softmax output of the transformer decoder. There may be multiple tokens to generate the required class label. In the case that the output does not match any class labels, we re-run the generation process with a different seed.

$$p(t_1, \ldots, t_{T-1}|X) = \prod_{i=0}^{n} \max_{j=1}^{|V|} P(v_j|v_{\leq i})y = argmax_{x \in V^T} p(x|X) \tag{5.1}$$

In order to improve the quality of the prompt, we employ RAG with help of a vector database and retrieve most similar $k = 15$ training instances to the test instance. We use ChromaDB to store the bug reports in the training set which runs SBERT (3) under the hood to calculate the cosine similarity between the test instance and training instances. The few-shot samples are constructed using the most similar instances to the test sample.

The SBERT transformer encoder is a pretrained model that is trained over pairs of text documents with an objective to shape the embedding space according to the cosine similarity. Software bug reports in the training dataset are encoded into vector representations with the pretrained model and stored in the vector database. The retrieval component in RAG system therefore uses a transformer encoder for a similarity-based retrieval operation for the generation step. It is expected that the retrieved instances have similar labels and condition the model to generate the correct prediction label.

CHAPTER 6

DATA ACQUISITION AND ANALYSIS

In our study, we focus on the bug reports collected from the open source Mozilla project[1] that contain the textual contents of the bug reports, their priority levels, and additional meta-data (54). We utilize the title, description and priority label fields and filter out instances with less than three words and instances without assigned priority labels.

In order to assess the generalization capabilities of the models we employ in this work, we explore datasets from other domains. Software bug priority detection requires domain-specific knowledge from the models and a modeling of the problem that captures the order information between classes. Higher priority levels can be quantitatively separated from lower priority levels which defines a distinguishing aspect to the problem. We make sure that this property is also observed in all the datasets. So, we choose app reviews and movie reviews domains, where users assign scores to the products, apps and movies, from a range of 1-5. This way, contrastive learning can be applied and the embedding space can be shaped accordingly.

6.1 Statistical Analysis

Frequently, the success of a predictive model is determined by the statistical properties and the quality of the training data. Therefore, it becomes critical to conduct statistical analyses on the datasets. However, which features contribute most to the downstream task performance is a difficult question to answer. There are efforts in the literature (55) to determine a standard way of measuring the classification difficulty of

[1] https://www.mozilla.org/en-US/

textual datasets by combining different parameters and statistics. Evolutionary Data Measures (EDM) (56) exploits evolutionary algorithms to select among 48 different statistics according to their contribution to the classification score on textual datasets from a diverse range of domains. They select a number of crucial statistics observed in classification datasets that are exceptionally correlated with difficulty. The open-sourced tool [2] measures the dataset difficulty. We apply it to the datasets used in this thesis and report the results.

EDM works by applying term frequency-inverse document frequency (tf-idf) feature extraction using the white space character as the separator in the input text data and a combination of class distribution and vocabulary-based occurrence measurements. The textual contents of the bug reports along with their titles are used as the data whereas the priority labels are used to calculate class distributions.

The difficulty report for the Mozilla dataset is given in Table 6.1. The severe imbalance problem is apparent in the minimum number of instances in a class (P5 priority class in this dataset) which hinders the model's ability to learn the descriptive features of this class. The vocabulary of the dataset is diverse, indicated by the 0.068 score and class diversity is beneficial for the classifiers. However, high class imbalance and Hellinger similarity yields an overall HIGH difficulty score of 4.127.

In order to visualize the class imbalance in the Mozilla dataset, label distributions in terms of percentages of each class are visualized in Figure 6.1. It is apparent that P3 class dominates the dataset, more than half of the dataset consists of P3 instances, which is possibly due to a bias in human behavior to assign an average priority label to a particular bug report. P4 and P5 classes are very rare, constituting 5.3% of the entire dataset combined, which would make it harder for models to learn deep features that correlate with these classes.

In order to assess the generalization capabilities of the methods we employ to other domains, we select two new datasets from similar tasks. We make sure the labels in the classification tasks contain an order information with respect to each other just as priority labels do. To this regard, we use an app review dataset (57) and Rotten Tomatoes movie reviews dataset (58). The target labels are the ratings given by

[2] https://github.com/Wluper/edm

Table 6.1: Mozilla Dataset Difficulty Report

Metric	**Value**	**Comment**
Dataset Size	40565	-
Vocab Size	288751	-
Number of Classes	5	-
Mean Items Per Class	8113.0	-
Min. Items in a Class	689	EXTREMELY LOW
Average Sentence Length	862.117	-
Distinct Words : Total Words	0.068	GOOD
Class Imbalance	0.802	SOMEWHAT HIGH
Class Diversity	1.102	GOOD
Max. Hellinger Similarity	0.731	HIGH
Mutual Information	1.424	GOOD
Difficulty	4.127	HIGH

users and audiences on a 1-5 rating scale. Naturally, there is an ordering information between classes similar to priority labels in software bug report priority detection which can be exploited via contrastive learning.

We apply the same EDM difficulty assessment procedure to the app reviews and movie reviews datasets and present their results in tables 6.2 and 6.3 respectively. The class imbalance problem is not observed in the app reviews dataset, since we sample equal number of instances for each class from raw data. This approach lowers the computational budget of the training process and ensures the dataset is balanced.

The difficulty report shows that there is a slight problem with class diversity. This means that the app review contents share token similarities, probably due to domain-related terms. Maximum Hellinger similarity is high, indicating a lower distance between word distributions in each class, a similar observation to class diversity. Mutual information between classes is not problematically high, providing a way to distinguish different labels. The overall assessment for classification difficulty is SOMEWHAT HIGH for this dataset with 3.805 difficulty score.

We plot the class distributions as percentages in Figure 6.2 for the sake of completeness. There is a perfect balance in the dataset thanks to selective downsampling.

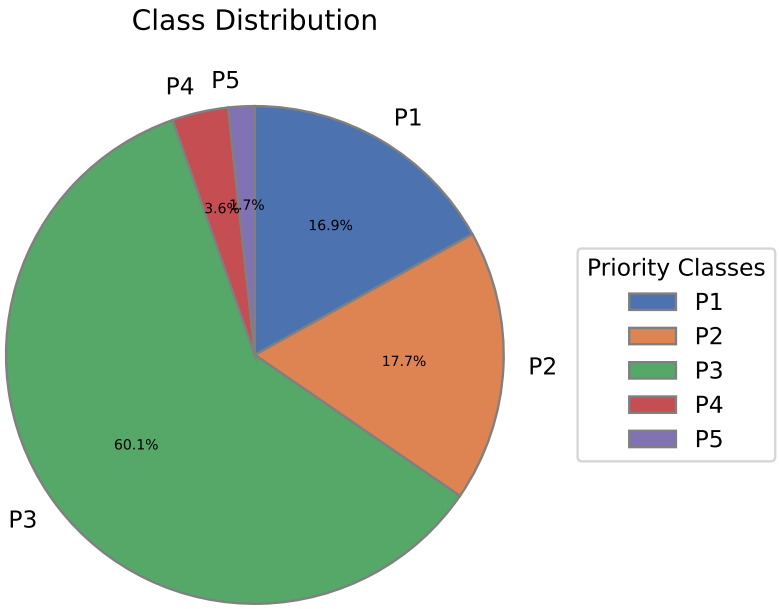


Figure 6.1: The distribution of the number of instances in each class for the Mozilla dataset

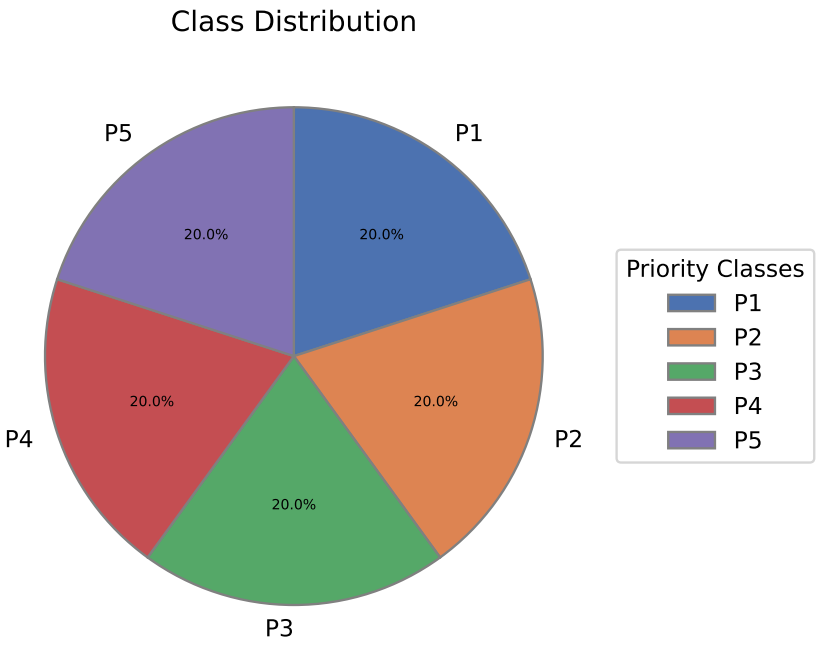


Figure 6.2: The distribution of the number of instances in each class for the App Reviews dataset

Table 6.2: App Reviews Dataset Difficulty Report

Metric	**Value**	**Comment**
Dataset Size	30000	-
Vocab Size	41619	-
Number of Classes	5	-
Mean Items Per Class	6000.00	-
Min. Items in a Class	6000	GOOD
Average Sentence Length	180.945	-
Distinct Words : Total Words	0.048	GOOD
Class Imbalance	0.0	GOOD
Class Diversity	1.609	SOMEWHAT HIGH
Max. Hellinger Similarity	0.804	HIGH
Mutual Information	1.344	GOOD
Difficulty	3.805	SOMEWHAT HIGH

Rotten Tomatoes movie reviews dataset is relatively small in size compared to the other datasets. The number of instances in the smallest class is slightly low. The class imbalance is not high but class diversity and maximum Hellinger similarity drive the difficulty level up. The difficulty score for this dataset is 4.148 and the assessment is HIGH. We expect classifiers to struggle with achieving high classification scores.

The class distribution plot for Rotten Tomatoes dataset is given in Figure **??**. P5 class is visibly scarce and the most dominant class, P4, is not a severely dominant class. Contrastive learning might help improve the underrepresented P5 class but class imbalance is not a critic problem, as indicated by the difficulty report and class distribution visualization.

6.2 Experimental Design

We apply 5-fold cross validation in all our experiments by splitting the datasets as 80% train - 20% test. The test splits do not share common samples, i.e. there is no overlap between test splits at each fold to ensure that the performance measurements (or observations) are independent of each other. The average score of 5 runs are

Table 6.3: Rotten Tomatoes Tomatometer Rating Difficulty Report

Metric	**Value**	**Comment**
Dataset Size	7291	-
Vocab Size	24407	-
Number of Classes	5	-
Mean Items Per Class	1458.2	-
Min. Items in a Class	390	SLIGHTLY LOW
Average Sentence Length	160.495	-
Distinct Words : Total Words	0.137	HIGH
Class Imbalance	0.579	GOOD
Class Diversity	1.379	SOMEWHAT HIGH
Max. Hellinger Similarity	0.693	SOMEWHAT HIGH
Mutual Information	1.360	GOOD
Difficulty	4.148	HIGH

reported in the results.

We use Precision, Recall and F1 scores, given in equation 6.1, 6.2, and 6.3 respectively, to calculate classification performance of models. True positives for a given class is the number of correctly predicted instances in test time, i.e. when the ground truth label and the predicted label matches. False positives occur when the classifier assigns the label of the given class but the ground truth label differs. False negatives indicate the number of instances for a given class assigned labels of a different class by the prediction model.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{6.1}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{6.2}$$

$$\text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.3}$$

Precision metric measures the model's classification performance by calculating the

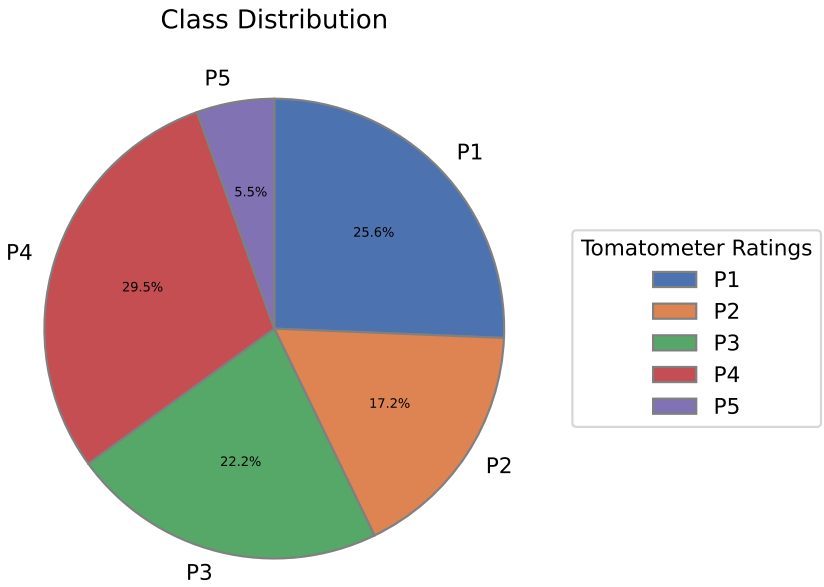


Figure 6.3: The distribution of the number of instances in each class for the Rotten Tomatoes dataset

ratio of correctly predicted instances out of all the instances assigned the label corresponding to the specific class. High Precision scores mean that the majority of the instances assigned the class label actually belong to that class. This is why Precision is also known as the false alarm rate.

Recall metric measures the portion of the instances belonging to a certain class correctly predicted by the model. Recall scores approach 100% when all the instances of the class are assigned the corresponding label in the test time, regardless of the number of false alarms. There is an inherent trade-off between Precision and Recall metrics since the model can maximize Precision by assigning predicting one correct sample and discarding the rest, which would drive Recall score to 0% or the model can maximize recall by assigning the class label to all the instances while increasing the number of false alarms dramatically and harming Precision scores greatly. Therefore, the model's ability to strike a balance between Precision and Recall becomes important, which can be achieved by intelligent decision-making with an understanding of the underlying problem structure.

As a measure of this trade-off, the geometric mean of the two performance metrics, re-

ferred to as the F1 Score, is developed. It is a frequently used performance metric that calculates the overall classification performance of classifiers with equal importance to Precision and Recall. A higher F1 Score essentially means a better classification quality if it is not dominated by either one of the performance metrics. Therefore, we report all three scores in our experiments and comment on the classification performances.

We refrain from reporting the average score of all classes as the classification score because a poorly trained classifier that only predicts the dominant class at test time may cause an unfairly optimistic calculation of performance. Instead, we report the performance of each class separately and comment on the performance differences between each class. The 5-fold averaging occurs between different folds for each separate class and minimizes performance differences due to stochastic effects, such as data splits and neural network initialization.

CHAPTER 7

RESULTS AND DISCUSSION

We present the average of 5-fold experiment results for software bug report priority detection for all methods in Table 7.1. There are a number of observations regarding different aspects of the problem.

Contrastive BERT outperforms other methods in most cases. With respect to all three performance metrics and in five different classes, Contrastive BERT approach outperforms standard BERT, OpenChat LLM and RAG approaches in 12 out of 15 cases (5 classes and 3 metrics). RAG improves precision for P3, the most dominant class, without a better F1 score, which may indicate an overfitting problem due to the dominance of the P3 class in the training set. This observation indicates that a carefully tailored classifier model can outperform models that are orders of magnitude larger in number of parameters. Incorporating the domain information about order relation between classes improves modeling capability compared to the standard BERT model.

P3 class performance is higher than other classes for all methods. Severe imbalance in the dataset causes model predictions to prefer the dominant class both for fine-tuning approaches and few-shot LLM outputs. Imbalance loss term in Contrastive BERT approach aims to mitigate the negative effects of dataset imbalance by calculating class distributions in the training set. Recall values are higher than precision values in BERT-based methods which indicates a greater number of false alarms in P3 class. This is a clear demonstration of the trade-off between precision and recall where the model overvalues the dominant class and assigns P3 label to many instances to increase recall while decreasing precision scores.

Table 7.1: Per-Class Classification Performances on Mozilla Core Dataset

	BERT					Contrastive BERT				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
Precision	40.32	35.73	74.25	10.70	15.40	**45.61**	**37.30**	76.12	**14.31**	**20.70**
Recall	34.18	31.92	**82.18**	6.27	6.42	**37.51**	**35.35**	82.07	10.40	**10.60**
F1-Score	36.76	33.52	77.93	7.78	8.54	**40.35**	**35.99**	**78.85**	**11.46**	**12.81**
	OpenChat					RAG+OpenChat				
Precision	18.46	17.42	61.54	3.35	2.05	15.81	9.92	**83.62**	5.89	4.35
Recall	14.41	30.90	46.11	9.40	0.39	31.51	16.98	67.28	**12.79**	0.86
F1-Score	16.19	22.28	52.72	4.94	0.66	21.05	12.52	74.57	8.07	1.44

LLM transformer decoder underperforms in classification. The OpenChat pre-trained LLM, based on transformer decoder architecture for text generation, fails to generalize from its training domain to software domain without further fine-tuning training. Few-shot prompting with training instances used for demonstrations acquired through random sampling mostly consist of the dominant class samples which leads to the highest scores for the P3 class. However, F1 score of the OpenChat model is the minimum among all the methods, which means vanilla caht models are not suitable as out-of-the-box classifiers for domain-specific applications.

RAG improves LLM performance in most cases. The integration of a vector database for RAG assistance in LLM generation increases classification performance, specifically according to the F1 score metric. Similarity-based retrieval system utilizes a transformer encoder architecture to generate semantic embeddings (3) and help prepare a more suitable prompt for the LLM in the few-shot setting (24).

P4 and P5 class performances are distinguishably lower than other classes. The training dataset does not include many P4 and P5 class samples which leads to under-representation during fine-tuning training for BERT-based methods. It is noteworthy that few-shot generative models also fail to accurately predict P4 and P5 labeled instances. This is possibly because the few-shot prompt is biased towards dominant classes and the LLM is conditioned in favor of these classes. RAG improves the

performance for these rare classes using a more selective sampling process.

In automated software bug report priority prediction, transformer decoder classifiers outperform generative LLM models. The task domain requires the model to have a sophisticated understanding of software management process which is better incorporated via parameter updates with fine-tuning. Contrastive learning provides a better exploitation of the ordering between classes which boosts classification performance for all classes.

We also test how each method performs in app reviews and movie reviews domains. For consistency in the notation, we report classes with same labels, i.e. P1, P2, ... P5. The mapping from app reviews to priority labels is straightforward, there is already a star rating with five classes. In movie reviews, we map the movie ratings into the 1-5 range linearly. Since an ordinal comparison between labels still holds, we apply the same learning methods as software bug report priority detection setup.

In Table 7.2, we observe that BERT-based methods outperform LLM approaches in app reviews star rating classification. The advantage of contrastive learning for BERT is not consistently observed for all classes, although the F1 Score is higher for three out of five classes for contrastive BERT. We note that the effect of imblance loss diminishes in this setting because the class distribution is balanced via undersampling. This reduces the difference from a regular fine-tuning process to a distinction with respect to different scores attached to review texts according to their writing tone and vocabulary. This is not the case for software bug reports since they usually contain code snippets, error logs and software-related terminology which requires domain-specific understanding from models to distinguish priority labels.

LLM performance in the app reviews domain improves with RAG in the majority of classes but fine-tuned transformer encoder classifiers outperform few-shot inference of transformer decoders. P1 and P5 class performances are noticeably higher for all models and specifically for LLMs. This can be attributed to a bias in user behavior that favors extremes, either a full praise or a full complaint - leading to more reviews with extreme ratings (P1 and P5), as opposed to neutral ones.

Table 7.3 shows classification results for movie reviews domain. LLM-based ap-

Table 7.2: Per-Class Classification Performances models on App Reviews Dataset

	BERT					Contrastive BERT				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
Precision	59.95	**40.87**	39.52	**50.36**	69.78	**61.08**	39.75	**40.34**	50.06	69.91
Recall	**56.07**	45.32	**45.33**	41.77	**66.29**	52.80	**46.76**	42.89	47.00	65.89
F1-Score	**57.88**	42.72	**41.96**	45.24	67.63	56.39	**42.91**	41.42	**48.36**	**67.75**
	OpenChat					RAG + OpenChat				
Precision	54.70	35.69	35.01	41.12	71.85	55.22	36.71	35.73	41.46	**77.61**
Recall	43.13	38.77	37.48	**51.41**	57.79	48.88	41.79	39.61	50.63	50.31
F1-Score	48.23	37.17	36.21	45.69	64.04	51.86	39.08	37.57	45.59	61.05

proaches outperform transformer encoder classifiers in this case, contrary to the observations in software and app reviews domains. Movie reviews domain may benefit from the influence of movies on popular culture and hence web scraped pretraining data used to train LLMs. We also note that transformer encoder classifiers may struggle due to the smaller size of the dataset used in fine-tuning.

RAG is not uniformly helpful, it boosts LLM performance according to F1 score in some classes while deteriorating it in extreme cases (P1 and P5). This might indicate that randomly sampled instances in the few-shot setting tend to favor the extremes while retrieval of similar samples shift the distribution towards a centralized shape (P3 achieves highest recall and P2 achieves highest F1 score).

When we compare the highest F1 scores observed in Table 7.1, 7.2, and 7.3 we see that software bug report priority detection performances are the lowest, apart from the dominant P3 class performance. The calculated classification difficulties for the utilized datasets are 4.127 for the Mozilla dataset, 3.805 for the App Reviews dataset, and 4.148 for Rotten Tomatoes dataset. Our performance observations are in line with the difficulty measurements. However, Rotten Tomatoes dataset has a considerably low imbalance score which helps models achieve higher performance scores even though the difficulty score is the highest. The domain of the dataset also helps models with a better general world knowledge in comparison to a domain with more expertise

Table 7.3: Per-Class Classification Performances models on Rotten Tomatoes Tomatometer Rating Dataset

	BERT					Contrastive BERT				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
Precision	50.15	54.47	34.15	31.59	48.67	50.65	56.28	34.36	30.80	49.24
Recall	35.73	43.74	44.34	29.57	55.74	36.46	38.39	41.81	35.23	56.14
F1-Score	41.73	48.52	38.58	30.55	51.96	42.40	45.64	37.72	32.87	52.46
	OpenChat					RAG + OpenChat				
Precision	**80.35**	45.49	37.12	**66.56**	**50.02**	43.36	**62.68**	**39.17**	31.48	48.88
Recall	**45.66**	**71.73**	60.84	21.88	**73.46**	42.86	57.49	**62.63**	**47.54**	23.13
F1-Score	**58.23**	55.67	46.11	32.93	**59.52**	43.11	**59.97**	**48.20**	**37.88**	31.41

required, software bug report priority detection.

CHAPTER 8

CONCLUSIONS

Software bug priority detection provides an automated assessment using the textual content of bug reports. We explore transformer encoder classifiers (BERT-based methods) using a standard cross-entropy loss and a contrastive learning loss and we experiment with Large Language Models (LLMs) using the transformer decoder architecture for text generation. Fine-tuning pretrained encoders tend to perform better than few-shot inference with LLMs even though there is an order of magnitude difference between the number of parameters and model size in favor of LLMs. Retrieval Augmented Generation (RAG) improves LLM performance where a vector database is used to retrieve similar samples to a given query. The vector database utilizes a pretrained transformer encoder (SBERT) under the hood, which lets the system harness the embedding space.

Contrastive learning provides a better way to incorporate domain knowledge into the model training process by leveraging the order information between classes. Wrong predictions that are further away from the ground truth are penalized more to regularize the model during training. Imbalance loss aims to negate the disadvantage of poorly represented classes by amplifying the loss values according to class distributions in the data. This way, the network can better focus on the minority classes that are underrepresented and harder to detect compared to instances belonging to the majority class.

Software bug reports domain requires expert-level knowledge from models which is hard to obtain in pretraining. Other more widely available domains, such as app reviews and movie reviews, highlight suitable applications of different models. Based on the nature of the problem and the structure of data, contrastive learning can help

mitigate predictive bias towards edge cases and extremes while few-shot LLM inference and RAG systems improve classification performance for other classes.

In the future, more advanced techniques such as ensemble learning, mixture of experts, and fine-tuning on LLMs with help of memory-efficient learning procedures such as Low Rank Adaptation (LoRA) (59) and QLoRA (60), may help improve the adaptation capabilities of neural models for the software bug report priority detection task. Furthermore, alternative representations for software bug reports that distinguish the code snippets and error logs and exploit them in a structured way can gain an advantage over regular NLP techniques.

The development of effective software bug priority detection models is critical for ensuring high-quality software products. Our work has shown that transformer encoder classifiers (BERT-based methods) perform well on this task, with fine-tuning pre-trained encoders proving to be more successful than few-shot inference using Large Language Models even though LLMs have significantly more parameters and larger model sizes compared to BERT. We also found that incorporating domain knowledge through contrastive learning improves performance by regularizing the models during training, while imbalance loss addresses class distribution issues for underrepresented classes. To improve LLM performance, vector databases and RAG systems are promising methods of incorporating domain-specific knowledge.

CHAPTER 9

BIBLIOGRAPHY

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[3] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019.

[4] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 392–401, IEEE Computer Society, 2013.

[5] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 111–120, 2009.

[6] C. E. Öztürk, E. H. Yilmaz, O. Köksal, and A. Koç, "Software module classification for commercial bug reports," in *2023 IEEE International Conference*

on Acoustics, Speech, and Signal Processing Workshops (ICASSPW), pp. 1–5, 2023.

[7] T. Hirsch and B. Hofer, "Using textual bug reports to predict the fault category of software bugs," *Array*, vol. 15, p. 100189, 2022.

[8] A. Dao and C. Yang, "Automated priority prediction for bug reports using comment intensiveness features and SMOTE data balancing," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 33, no. 3, pp. 415–433, 2023.

[9] T. W. W. Aung, Y. Wan, H. Huo, and Y. Sui, "Multi-triage: A multi-task learning framework for bug triage," *J. Syst. Softw.*, vol. 184, p. 111133, 2022.

[10] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 946–957, 2022.

[11] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 86–96, 2019.

[12] Z. Zhang, R. Winn, Y. Zhao, T. Yu, and W. G. Halfond, "Automatically reproducing android bug reports using natural language processing and reinforcement learning," *arXiv preprint arXiv:2301.07775*, 2023.

[13] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," in *ACM India Joint International Conference on Data Science and Management of Data*, Association for Computing Machinery, 2019.

[14] S. Fang, Y.-s. Tan, T. Zhang, Z. Xu, and H. Liu, "Effective prediction of bug-fixing priority via weighted graph convolutional networks," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 563–574, 2021.

[15] W.-Y. Wang, C.-H. Wu, and J. He, "Clebpi: Contrastive learning for bug priority inference," *Information and Software Technology*, p. 107302, 2023.

[16] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[17] C. Manning, P. Raghavan, and H. Schutze, "Term weighting, and the vector space model," *Introduction to information retrieval*, pp. 109–133, 2008.

[18] D. M. W. Powers, "Applications and explanations of Zipf's law," in *New Methods in Language Processing and Computational Natural Language Learning*, 1998.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[20] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[21] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[22] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training,"

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners,"

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[25] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[26] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[27] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *The Journal of Machine Learning Research*, vol. 23, no. 1, pp. 5232–5270, 2022.

[28] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27730–27744, 2022.

[29] D. E. Rumelhart and J. L. McClelland, "Learning internal representations by error propagation," 1987.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, *et al.*, "Learning transferable visual models from natural language supervision," in *International conference on machine learning*, pp. 8748–8763, PMLR, 2021.

[32] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*, pp. 8821–8831, PMLR, 2021.

[33] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10674–10685, IEEE, 2022.

[34] C. Rebuffel, M. Roberti, L. Soulier, G. Scoutheeten, R. Cancelliere, and P. Gallinari, "Controlling hallucinations at word level in data-to-text generation," *Data Min. Knowl. Discov.*, vol. 36, no. 1, pp. 318–354, 2022.

[35] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[36] B. Sun, Y. Li, F. Mi, F. Bie, Y. Li, and K. Li, "Towards fewer hallucinations in knowledge-grounded dialogue generation via augmentative and contrastive knowledge-dialogue," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1741–1750, 2023.

[37] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pp. 9459–9474, 2020.

[38] G. Guizzo, F. Califano, F. Sarro, F. Ferrucci, and M. Harman, "Inferring test models from user bug reports using multi-objective search," *Empirical Software Engineering*, vol. 28, no. 4, p. 95, 2023.

[39] X. Cao, T. Liu, J. Zhang, M. Feng, X. Zhang, W. Cao, H. Sun, and Y. Zhang, "Sbrpbert: A bert-based model for accurate security bug report prediction," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 129–134, 2022.

[40] H. Liu, M. Shen, J. Jin, and Y. Jiang, "Automated classification of actions in bug reports of mobile apps," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 128–140, 2020.

[41] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, p. 321–357, jun 2002.

[42] W. Zheng, Y. Xun, X. Wu, Z. Deng, X. Chen, and Y. Sui, "A comparative study of class rebalancing methods for security bug report classification," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1658–1670, 2021.

[43] B. Alkhazi, A. DiStasi, W. Aljedaani, H. Alrubaye, X. Ye, and M. W. Mkaouer,

"Learning to rank developers for bug report assignment," *Applied Soft Computing*, vol. 95, p. 106667, 2020.

[44] Q. Umer, H. Liu, and I. Illahi, "Cnn-based automatic prioritization of bug reports," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1341–1354, 2019.

[45] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46846–46857, 2019.

[46] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[47] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[48] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[49] G. Wang, S. Cheng, X. Zhan, X. Li, S. Song, and Y. Liu, "Openchat: Advancing open-source language models with mixed-quality data," *arXiv preprint arXiv:2309.11235*, 2023.

[50] T. Sorensen, J. Robinson, C. Rytting, A. Shaw, K. Rogers, A. Delorey, M. Khalil, N. Fulda, and D. Wingate, "An information-theoretic approach to prompt engineering without ground truth labels," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (S. Muresan, P. Nakov, and A. Villavicencio, eds.), (Dublin, Ireland), pp. 819–862, Association for Computational Linguistics, May 2022.

[51] B. Clavié, A. Ciceu, F. Naylor, G. Soulié, and T. Brightwell, "Large language models in the workplace: A case study on prompt engineering for job type classification," in *International Conference on Applications of Natural Language to Information Systems*, pp. 3–17, Springer, 2023.

[52] Y. Ahn, S.-G. Lee, J. Shim, and J. Park, "Retrieval-augmented response generation for knowledge-grounded conversation in the wild," *IEEE Access*, vol. 10, pp. 131374–131385, 2022.

[53] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval augmentation reduces hallucination in conversation," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 3784–3803, 2021.

[54] A. Lamkanfi, J. Pérez, and S. Demeyer, "The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 203–206, IEEE, 2013.

[55] X. Zhang, X. Li, and Y. Feng, "A classification performance measure considering the degree of classification difficulty," *Neurocomputing*, vol. 193, no. C, pp. 81–91, 2016.

[56] E. Collins, N. Rozanov, and B. Zhang, "Evolutionary data measures: Understanding the difficulty of text classification tasks," *CoNLL 2018*, p. 380, 2018.

[57] G. Grano, A. Di Sorbo, F. Mercaldo, C. A. Visaggio, G. Canfora, and S. Panichella, "Android apps and user feedback: a dataset for software evolution and quality improvement," in *Proceedings of the 2nd ACM SIGSOFT international workshop on app market analytics*, pp. 8–11, 2017.

[58] S. Leone, "Rotten tomatoes movies and critic reviews dataset," 2020.

[59] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, *et al.*, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2021.

[60] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *arXiv preprint arXiv:2305.14314*, 2023.