

CAPITAL UNIVERSITY OF SCIENCE AND
TECHNOLOGY, ISLAMABAD



Duplicate Bug Report Detection Using Hybrid Model

by

Nabiya Fatima

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the

Faculty of Computing

Department of Computer Science

2023

Copyright © 2023 by Nabiya Fatima

All rights reserved. No part of this thesis may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, by any information storage and retrieval system without the prior written permission of the author.

It is with great joy and pleasure to dedicate my thesis work to both my parents, without whom I would not have been able to make it this far in life. My parents have been there for me, through all the highs and lows, lending me with infinite support to carryout this journey. I'm truly grateful for the love and luxuries bestowed upon me by my parents. I would also like to dedicate this thesis to my supervisor for his cooperative and helpful guidance.



CERTIFICATE OF APPROVAL

Duplicate Bug Report Detection Using Hybrid Model

by

Nabiya Fatima

(MCS213027)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Ashfaq Ahmed	MY University
(b)	Internal Examiner	Dr. Umair Rafique	CUST
(c)	Supervisor	Dr. Syed Saqib Raza Rizvi	CUST



Dr. Syed Saqib Raza Rizvi

Thesis Supervisor

September, 2023

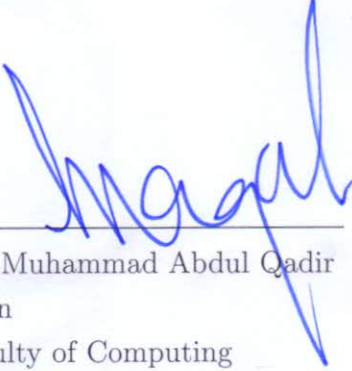


Dr. Abdul Basit Siddique

Head

Dept. of Computer Science

September, 2023



Dr. Muhammad Abdul Qadir

Dean

Faculty of Computing

September, 2023

Author's Declaration

I, **Nabiya Fatima** hereby state that my MS thesis titled “**Duplicate Bug Report Detection Using Hybrid Model**” is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.



(Nabiya Fatima)


Registration No: MCS213027

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled “**Duplicate Bug Report Detection Using Hybrid Model**” is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.



(Nabiya Fatima)

Registration No: MCS213027

Acknowledgement

I would like to acknowledge the blessings of Allah Almighty for helping me through all the hardships of my life. I would like to acknowledge the help and support of my family, who showered me with constant and never-ending support. Furthermore, I acknowledge the support and much needed guidance of my supervisor in conduction of this thesis.

(Nabiya Fatima)

Abstract

Duplicate bug report detection is an essential and challenging task, in large software projects where hundreds and thousands of bugs are reported each day and maintained through bug tracking systems. The process is necessary to avoid redundant work and to maintain the quality of the software. Duplication occurs when multiple users report the same bug. As the bug reports are written in natural language the same bug can be reported multiple times with varying descriptions, generating non-trivial duplicate bug reports. In order to avoid the redundant work an expert need to identify these duplicates and label them manually. This process requires extensive amount of efforts in terms of time and cost and directly impacts the bug fixing time. Earlier studies on Duplicate bug report detection uses Information Retrieval techniques. These techniques are fast and create sparse vectors for text representation. They however, lack to capture contextual and semantic meaning of words. However recent efforts to identify duplicate reports rely on advanced deep neural techniques. These approaches typically involve dense representations of bug reports, taking into account the semantic meaning of the words. Their higher precision and recall rates unfortunately, is accompanied by higher computational cost and processing time. There exists a need to develop new techniques with lower run time and higher precision. This work proposed a hybrid model that uses both IR and DL techniques to identify duplicate bug reports. The proposed approach utilizes both structured and unstructured data to retrieve the possible duplicates. It uses a time efficient technique, BM25 to filter out the bug reports and then a semantic reranking model is applied to rank the duplicates based on semantic similarities. We validate our approach on two benchmark datasets: Eclipse and Open Office. The results show a mean of 83.5% for the retrieval task, at Recall@20 and a MAP of 68.5% across both the datasets with an average of 5.65s PSPT.

Contents

Author’s Declaration	iv
Plagiarism Undertaking	v
Acknowledgement	vi
Abstract	vii
List of Figures	xi
List of Tables	xii
Abbreviations	xiii
1 Introduction	1
1.1 Bug Tracking	3
1.1.1 Issues in Bug Tracking	6
1.2 Problem Statement	10
1.3 Research Questions	11
1.4 Proposed Solution	11
1.5 Objective	12
1.6 Thesis Organization	12
2 Literature Review	13
2.1 Duplicate Bug Report Detection (DBRD)	13
2.2 Information Retrieval Approaches	14
2.2.1 Count Based Approaches	14
2.2.2 BM25 Approach	15
2.3 Machine Learning Approaches	17
2.3.1 CNN Based Approaches	17
2.3.2 BERT Based Approaches	18
2.3.3 Hybrid Approaches	19
2.3.4 Other Approaches	21
2.4 Discussion on DBRD Approaches	22

3	Research Methodology	27
3.1	Problem Identification	27
3.1.1	Literature Review	29
3.1.2	Research Gap Identification	30
3.1.3	Problem Formulation	30
3.1.4	Dataset Review and Selection	30
3.1.4.1	Eclipse dataset	30
3.1.4.2	OpenOffice dataset	31
3.1.4.3	Firefox	31
3.1.4.4	Kibana dataset	32
3.1.4.5	Selected datasets	32
3.1.5	Preliminary Preprocessing	33
3.1.6	Proposed Hybrid Model for DBRD	34
3.1.7	Preprocessing	34
3.1.7.1	Feature Selection	34
3.1.7.2	Natural Language Processing	36
3.1.7.3	Tokenization	37
3.1.7.4	Stemming	37
3.1.7.5	Stop words removal	37
3.1.7.6	Processed Feature Set	38
3.1.7.7	BM25 Indexing	39
3.1.7.8	Similarity Measure and Ranking	39
3.1.8	Semantic Reranking	41
3.1.8.1	SBERT	41
3.1.8.2	Fine Tuning SBERT Model	42
3.1.8.3	Data Preparation	42
3.1.8.4	Select SBERT model	43
3.1.8.5	Architecture	43
3.1.8.6	Loss Function	44
3.1.8.7	Similarity Measure	45
3.1.9	Evaluation metrics	46
3.1.9.1	Recall@K	46
3.1.9.2	Mean Average Precision (MAP)	48
4	Results and Discussion	49
4.1	Dataset Description	49
4.2	Duplicate Bug Report Retrieval Performance	50
4.3	Comparison with Previous Bug Report Retrieval Techniques	52
4.4	Effectiveness of the Approach in Terms of Response Time	54
4.5	Addressing Research Questions	57
5	Conclusion and Future Work	59
5.1	Limitations	59
5.1.1	Dependency on BM25 Model	59
5.1.2	Generalization Across Diverse Platforms	59

5.2	Conclusion	60
5.3	Future Work	60
5.3.1	Cross-domain Data	60
5.3.2	Optimizing Top-N Selection for Enhanced Duplicate Bug Report Detection	61
5.3.3	Integration with Bug Tracking Tools	61
	Bibliography	62

List of Figures

1.1	Bug life cycle [12]	3
1.2	Percentage of DBRs	8
1.3	No. of days required for analyzing the DBRs	8
1.4	IR model [23]	9
1.5	Deep Learning based NLP [25]	10
3.1	Proposed research methodology for retrieval of DBRs	28
3.2	Literature review hierarchy	29
3.3	Bugs title with tags	33
3.4	Bugs title without tags	33
3.5	Proposed approach for DBRD	35
3.6	Tokenization of title using NLTK	37
3.7	Stemming of tokenized title	38
3.8	Stop words removal of tokenized Title	38
3.9	Top-N similar bugs reports	41
3.10	Bugs triplet dataset	43
3.11	SBERT architecture	44
3.12	Triplet loss working	45
4.1	Recall@K on Eclipse dataset using BM25+SBERT	51
4.2	Recall@K on OpenOffice dataset using BM25+SBERT	51
4.3	MAP of Eclipse and OpenOffice datasets	52
4.4	Comparison of recall@K for BM25 and BM25+SBERT on Eclipse dataset	53
4.5	Comparison of recall@K for BM25 and BM25+SBERT on OpenOffice dataset	54
4.6	Recall rate at various values of K, and baseline approach in comparison to BM25 and BM25+SBERT.	55
4.7	Recall rate at various values of K, and all baselines in comparison to BM25 and BM25+SBERT.	55
4.8	PSPT of BM25, BERT, and BM25+SBERT	57

List of Tables

1.1	Generic bug report format[15]	5
1.2	Example of duplicate bug report [13]	7
1.3	Bugs data statistics [21]	7
2.1	Comparative Analysis of Existing Techniques	24
3.1	Eclipse Dataset statistics	31
3.2	OpenOffice Dataset statistics	31
3.3	Firefox Dataset statistics	32
3.4	Kibana Dataset statistics	32
3.5	Eclipse - Bug report format [52]	36
3.6	Example of TF-IDF calculation	40
3.7	Cosine similarity calculation example	47
4.1	Eclipse and Open Office datasets statistics	50
4.2	Train and test split	50
4.3	Recall@K for Eclipse dataset on all approaches	53
4.4	Recall@K for Open Office dataset on all approaches	54
4.5	PSPT for different sample sizes	56

Abbreviations

DBRD	Duplicate Bug Report Detection
DBR	Duplicate Bug Report
DL	Deep Learning
IR	Information Retrieval
ML	Machine Learning
NLP	Natural Language Processing
SQA	Software Quality Assurance

Chapter 1

Introduction

In the rapidly changing landscape of software development, ensuring quality software has become a principal concern. Nearly every industry relies on software for various purposes, including development, marketing, production, and support. A primary objective of software engineering is to develop high-quality software while adhering to predetermined cost and time constraints [1]. In the present time, software's have evolved into complex and sizable systems, and the expense to fix errors or faults in them is quite high [2].

Software engineering serves as the foundation upon which dependable and high-quality software is built [3]. It encompasses a systematic approach to designing, developing, testing, and maintaining software systems. By employing methodologies grounded in software engineering, development teams can harness structured and well-defined processes to create software that not only fulfills functional requirements but also adheres to principles of scalability, maintainability, and reusability [3].

Software Quality Assurance (SQA) emerges as a crucial discipline, encompassing a range of practices and processes aimed at delivering software that meets or exceeds user expectations. At the core of SQA lies the objective of achieving reliable, robust, and user-friendly software that aligns with industry standards and best practices [4]. SQA plays a pivotal role in identifying and mitigating potential risks throughout the software development life cycle.

One of the key strategies employed within SQA is software testing [5]. Software testing is a systematic process that involves evaluating a software application's behavior against specified requirements [4]. It serves as a means to verify that the software functions as intended and that it performs consistently across various scenarios [5]. By subjecting software to a battery of test cases, developers can identify deviations from the expected behavior and rectify issues before they reach end-users [6].

Among the diverse methodologies and techniques within software testing, Black Box Testing stands out as a foundational approach [7]. Black Box Testing involves examining the behavior of software externally, without knowledge of its code or internal structure [7]. Testers analyze the software's using inputs and examine outputs to assess whether the software responds correctly and as expected.

A natural consequence of thorough software testing is the identification of defects or bugs within the application [8]. As Black Box Testing focuses on evaluating the software from an end-user perspective, it is particularly effective in revealing functional discrepancies that may otherwise remain unnoticed [7]. Detected bugs can encompass a wide range of issues, from minor inconveniences to critical failures that impact the software's functionality and reliability [8].

In this context, effective bug reporting becomes a pivotal activity. Timely and comprehensive reporting of bugs, anomalies, and unexpected behaviors allows development teams to prioritize and address these issues promptly [9]. The bug reporting process serves as a bridge between the testing phase and the software development cycle, enabling developers to refine the software based on real-world usage scenarios [8].

Typically, bugs are reported by testers or users to explain the nature of the problem they encounter while using the software [10]. These bug reports help the developers to locate and fix the reported bug. In large software projects, several hundreds of bug reports are submitted to the Bug Tracking System (BTS) per day. Moreover, effective communication channels between testers and developers are crucial for a streamlined bug resolution process.

1.1 Bug Tracking

Bug Tracking refers to the process of identifying, monitoring, prioritizing and fixing bugs in software products [9]. Large software systems may have hundreds or thousands of bugs. To maintain the quality of the software, these bugs need to be identified, monitored, prioritized and resolved within a given time frame [11]. It is an iterative process as each new software update or version needs to be tested and debugged[9].

When a bug is detected, it is passed through different stages. Figure 1.1 shows how bug reports are addressed at each stage. At each stage of this cycle, the bug is handled as described below [12]:

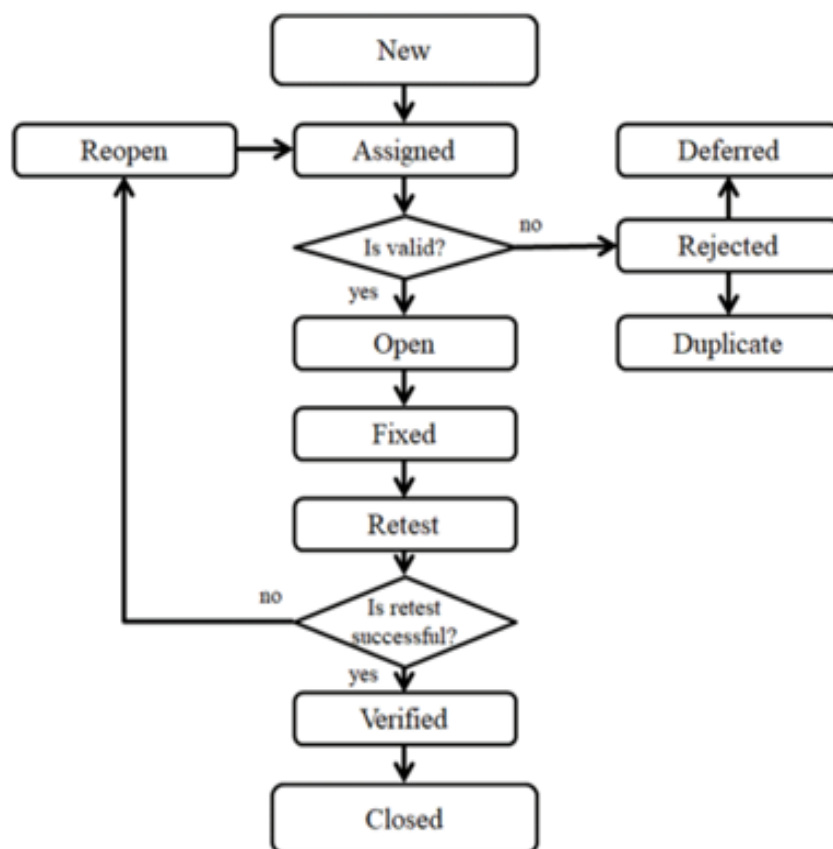


FIGURE 1.1: Bug life cycle [12]

- **New:** A new bug is initially discovered by a user or the QA team, and it is logged. Then, it is moved to the "Assigned" stage, where it is classified to determine how it will be handled.

- **Assigned:** At this stage, a tester logs the bug, and it undergoes triage to determine its priority and/or severity. If the bug is found to be unimportant, it is moved to the "Rejected" phase. However, if the bug is considered significant, it is assigned to the relevant developer according to its priority and severity.
- **Open:** At this stage, the bugs that are assigned to developers are analyzed and corrected. When the bug is fixed, it is moved to the "Fixed" stage.
- **Fixed:** At this stage, the developer sends the bug to the testing team after resolving it. Based on the results of the test, the bug may be moved to the "Reopen" or "Retest" stages, depending on whether further fixing or testing is required.
- **Reopen:** If a bug is still present even after the developer has fixed it, it is sent back to the "Assigned" stage, and the cycle is repeated.
- **Rejected:** If a defect is not genuine then the status is set to "Rejected".
- **Deferred:** When a bug is assigned the "Deferred" status, it means that it is expected to be fixed in the next releases. This status may be assigned due to various reasons such as low priority of the bug, lack of time for release, and minimal impact on the software.
- **Duplicate:** If a bug is reported more than once or if two bugs cause the same problem, one of them is marked as "Duplicate."
- **Retest:** At this stage, the tester performs a retest on the modified code provided by the developer to check if the defect has been resolved.
- **Verified:** After the developer has fixed the bug, the tester performs a retest. If the bug is no longer present, its status is changed to "Verified."
- **Closed:** If the tester test that the bug no longer exists, then its status is changed to "Closed".

The purpose of the bug life cycle is to easily communicate the current status of the bug to different users and to track the actual progress of bugs. In practice,

the process is not that simple. A reporter has to manually examine all the reports before reporting a new bug [13].

The general format for reporting a bug is shown in Table 1.1. Bug report typically contains both structured and unstructured information. Structured features encompass project-related details like priority, component, version, and status, among other features. Unstructured fields comprise text composed in natural language, encompassing features such as the title, which offers a concise bug description, and the detailed description, which provides developers with an explanation of the code malfunction, its causes, and how to reproduce it. Developers frequently depend on these unstructured features within bug reports for bug identification and resolution [14].

TABLE 1.1: Generic bug report format[15]

Feature	Description
Bug ID	112212
Product	General
Component	Preferences
Title	Language encodings in font preferences dialog not sorted
Status	Closed
Resolution	Fixed
Duplicate ID	1112
Priority	P4
Severity	Minor
Version	5.0
Platform	Macintosh
OS	10.12.0
Created	2020-03-11 11:53:00 -0500
Modified	2020-05-20 18:07:18 -0400
Description	Language encodings are listed in a seemingly random order.; The order be alphabetical (and therefore change with localization).; As a special case; User-Defined should be last.

These bugs are stored in a bug repository. These bugs are then assigned the bug to a developer who then determines the order in which to work on the assigned bug based upon its severity and priority. These bug reports are manually examined to identify if a new bug is a duplicate of the existing bug or not [15].

As the reports are written using natural language, a tester either relies on their knowledge of the bug repository or must perform a series of manual searches in order to find the duplicate bug. The former approach relies in the knowledge and expertise of the tester and the latter approach involves a lot of effort in terms of time and cost [16]. Both of these approaches could result in missed or false identification of the duplicate. Therefore, the challenge of identifying duplicate bugs is exacerbated by the reliance on natural language in bug reports.

1.1.1 Issues in Bug Tracking

Finding the DBR (duplicate bug report) is a challenging task. The repository contains two types of bug reports: Master reports and duplicate reports [17]. Each DBR must have a corresponding master report and have to make sure that they both address the same bug [17]. Duplication occurs when more than one user submits the bug report for the same problem [18, 19]. Due to the use of natural language to write bug reports, it is possible to describe the same bug in many different ways. Table 2 illustrates an example of DBRs (duplicate bug reports). To address the issue of duplicate bugs, it is necessary to classify these bugs as duplicates and link them to the master bug report. Manually identifying the duplicate bug report requires an extensive amount of effort both in terms of time and cost, and requires complete knowledge of the bugs [20].

An exploratory study [21], on nine different projects was performed. The study analyzes the total number of bugs reported in a specific project, the number of duplicates reported, the percentage of duplicates, and the time required to resolve the duplicate bug reports. Table 1.3 encapsulates the statistical details for each project, shedding light on the prevalence of duplicate bug reports and their impact on overall project timelines.

TABLE 1.2: Example of duplicate bug report [13]

Bug Id	Summary
85064	[Notes 2] No scrolling of document content by use of mouse wheel
85377	[CWS notes2] unable to scroll in a note with the mouse wheel
85502	Alt+ \downarrow letter \downarrow does not work in dialogs
85819	Alt- \downarrow key \downarrow no longer works as expected
85487	Connectivity: evoab2 needs to be changed to build against changed api
85496n	Connectivity fails to build (evoab2) in m4

TABLE 1.3: Bugs data statistics [21]

Project	#Bugs	#Duplicates	%Duplicates	Median - Resolving Time
Mozilla Core	205,069	44,691	21.8%	102.1 days
Firefox	115,814	35,814	30.9%	76.4 days
Thunderbird	32,551	12,501	38.4%	83.7 days
Eclipse Platform	85,156	14,404	16.9%	29.8 days
JDT	45,296	7,688	17.0%	23.0 days
Hadoop	12,855	1,861	14.5%	14.3 days
HDFS	12,779	1,659	13.0%	9.7 days
Cassandra	14,071	2,083	14.8%	8.6 days
MapReduce	7,019	977	13.9%	28.2 days

A graph illustrating the proportion of DBRs can be noted in Figure 1.2. It is evident that duplicate bug reports account for nearly 20% of the entire bug report count. The time dedicated to searching for and analyzing these duplicate bug reports is depicted in Figure 1.3.

Duplicate bug detection approaches can be divided into Information Retrieval (IR) techniques and Deep Learning (DL) based techniques. IR involves the process of representing, storing, and searching through a large dataset with the aim of extracting valuable insights and providing access to discover pertinent outcomes that fulfill the user's requirements in response to a user query [22]. The core structure of IR has the following steps.

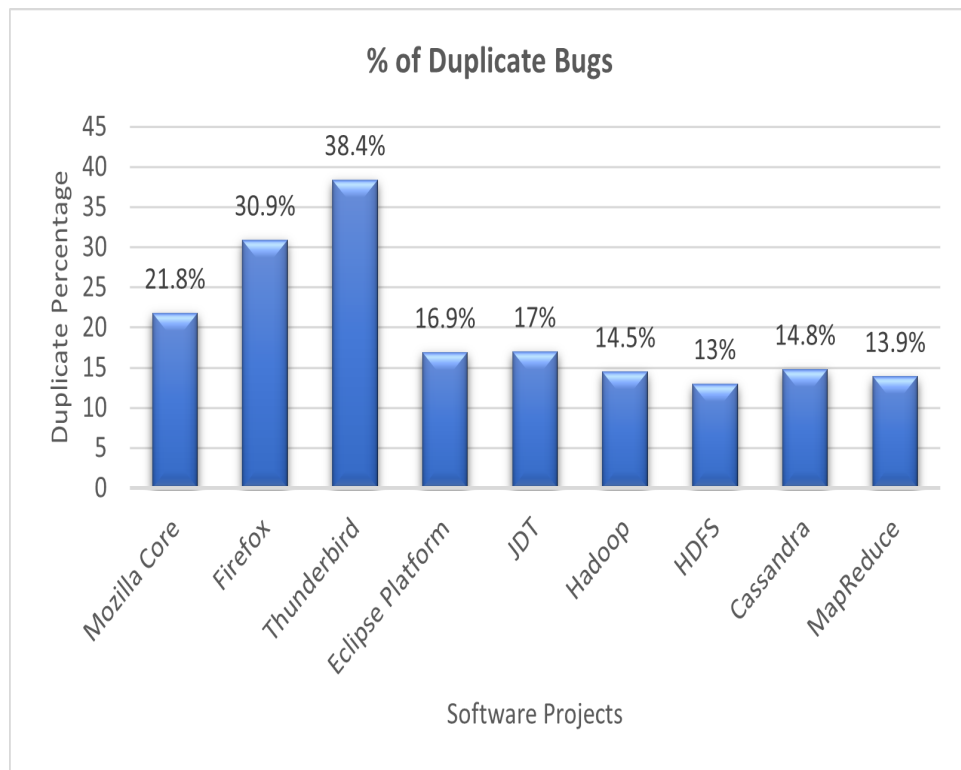


FIGURE 1.2: Percentage of DBRs

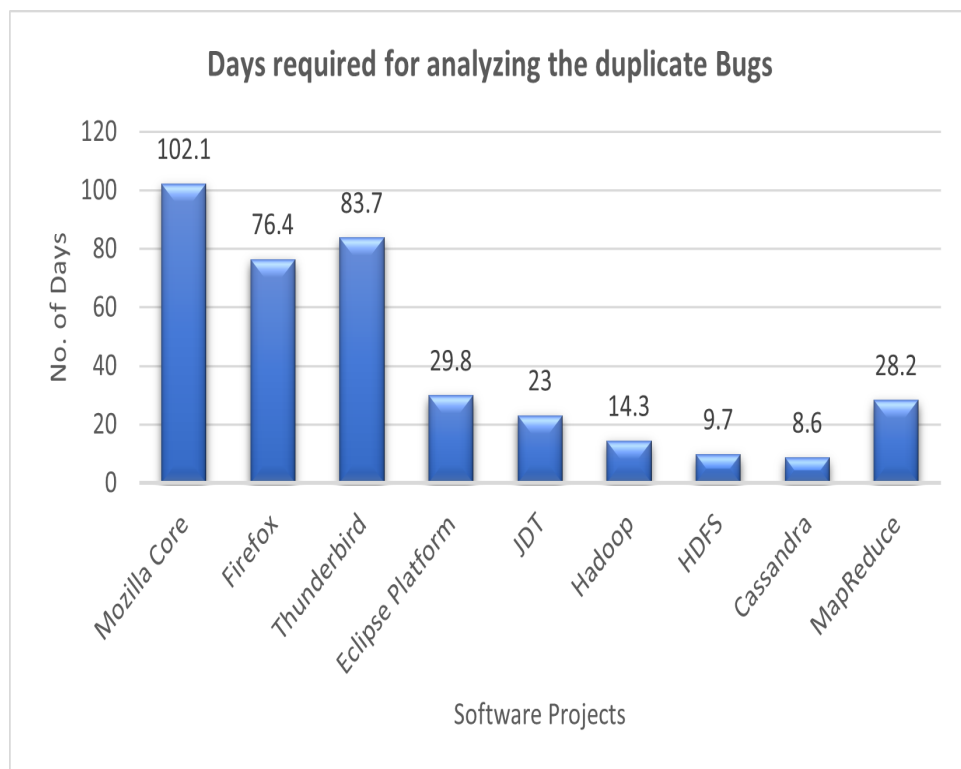


FIGURE 1.3: No. of days required for analyzing the DBRs

A user query that initializes the process, and a corpus of documents, constituting the data under consideration. Next is the preprocessing phase which encompasses tokenization, stemming, and stop word removal. The most important step is indexing which constitutes a pivotal phase where a comprehensive index is crafted for the terms within the documents. This index maps terms to the pertinent documents, serving as a catalyst for expediting the retrieval process. The last step is matching and ranking. This process yields relevance scores via query-document comparison, presenting ranked results to users [23]. Figure 1.4 shows a basic workflow for information retrieval model.

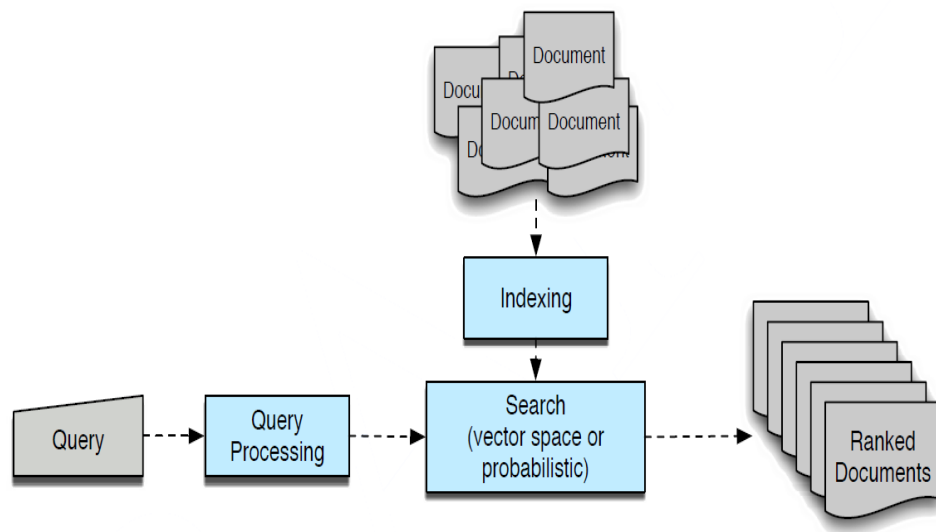


FIGURE 1.4: IR model [23]

In IR, the representation of text is done using Bag of Words (BOW), Term Frequency (TF), or Inverse Document Frequency (TFIDF). These techniques generate sparse vectors but fail to capture the semantic relation between words [24].

To overcome the problems of semantic search, deep learning has emerged as a prominent area of research, finding widespread application across various challenges within Natural Language Processing (NLP). Some of the notable deep learning models for natural language processing are Word2vec, GloVe, and BERT. These models are effective in capturing the semantic meaning of the document. They represent text using dense vectors. These dense vectors are then used for various tasks like topic modeling, classification, etc [25]. Figure 1.5 shows a basic workflow for NLP based deep learning model.

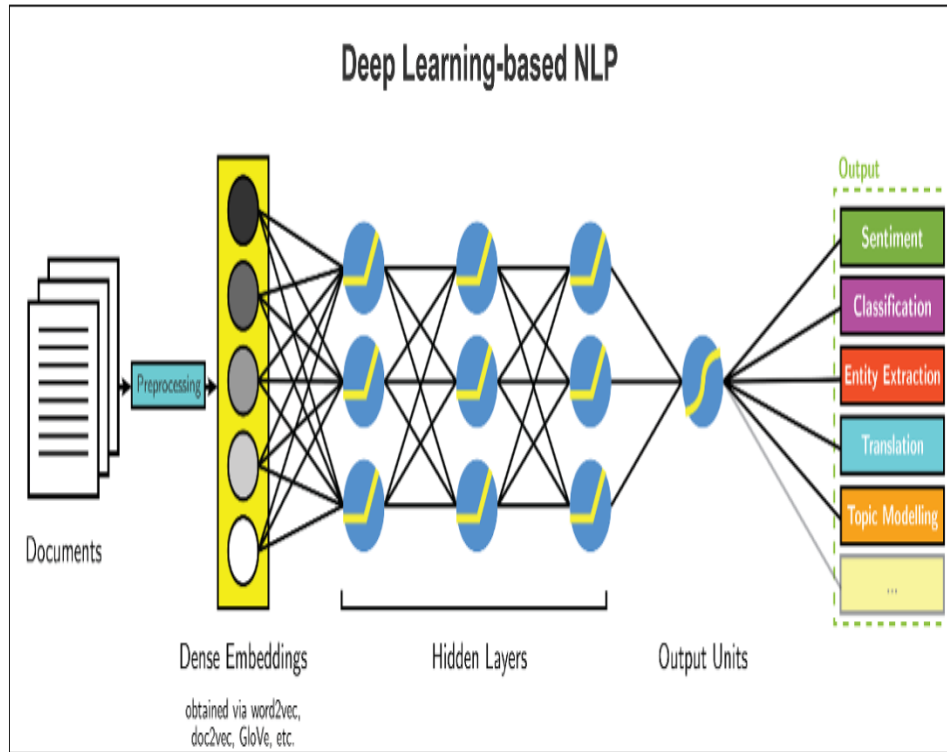


FIGURE 1.5: Deep Learning based NLP [25]

Consequently, the measure of semantic likeness between words is established based on spatial separation, often using metrics like cosine similarity or Euclidean distance [5]. Although these models are able to capture the semantic meaning between query and document they come at a cost of high computational complexity.

1.2 Problem Statement

Existing studies either use IR or DL models to find the duplicate bug report from the corpus. We have performed an in-depth analysis of the literature and identified the following gaps:

1. In most of the studies [26–31], traditional IR techniques like TF-IDF, and BM25 have been used to compute similarity between bug reports. These techniques are comparatively fast and generate sparse vectors for text representation but struggle to capture the semantic and contextual meaning of words in a document.

2. Recent studies [10, 13, 16, 32] have employed DL models such as CNN, BERT, and Word2Vec. These models excel in capturing semantic significance, yielding dense vectors. While these DL models have higher precision and recall rates, their computational cost and hence the response time is high. Hence, a comparative assessment should be done to replace the existing approaches with some new simpler techniques with low runtime complexity [29, 33, 34].

1.3 Research Questions

We have formulated the following research questions based on the problem statement described above.

1. How the proposed approach using both sparse and dense vectors be more efficient in retrieving duplicate bug reports?
2. How can the processing time of the proposed approach be reduced compared to previous DBRD approaches?

1.4 Proposed Solution

The proposed approach merges two distinct methodologies, IR and deep learning semantic model, with the objective of effectively detecting duplicate bug reports. IR models are fast as compared to DL models but fail to capture the semantic meaning. On the other hand, DL models are computationally expensive but are able to capture the semantic meaning between words. This integration seeks to capitalize on the strengths of both techniques. Initially, the solution employs a BM25, which is an IR technique for the identification of potential duplicate reports. Subsequently, to enhance the accuracy and relevance of the results, a

specialized semantic model is used to rerank the bug reports based on semantic similarity. This two-step process is designed to overcome the limitations inherent in singular approaches. As a result, the processing time required is notably reduced in comparison to employing only full semantic models.

1.5 Objective

- To propose a novel hybrid model that efficiently utilizes IR and DL models.
- Evaluate the effectiveness of the proposed technique on the benchmark dataset.
- Reduce the computational cost and hence the processing time for DBRD.

1.6 Thesis Organization

The subsequent sections of the study are structured as follows:

Chapter 1: Introduces the research subject and goals, addressing the problem statement and technological context to ensure effective outcome presentation.

Chapter 2: Presents Literature Review, exploring relevant prior research and identifying research gaps.

Chapter 3: Proposed a methodology for addressing the questions raised in Chapter 1.

Chapter 4: Presents the performance evaluation of outcomes achieved through the application of the proposed approach.

Chapter 5: Summarizes the research findings, draws conclusions, and outlines future directions for study.

Chapter 2

Literature Review

2.1 Duplicate Bug Report Detection (DBRD)

DBRD is a process of identifying DBRs within bug tracking repository. In a standard bug tracking system, over a thousand bugs are logged on a daily basis, as mentioned by Kukkar et al. [13] in a study on duplicate issues. The problem arises when the same bug is reported by multiple users. As bug reports are written in plain text, it can be challenging for the tester reporting the issue to recognize a DBR prior to submitting a new one [16].

In DBRD domain various approaches have been proposed for the classification and retrieval of DBRs. Approaches for identifying DBRs encompass both structured and unstructured data [35]. Structured data includes product, priority, component, resolution, bug severity, bug status, and version. On the other hand, unstructured data pertains to textual attributes, typically derived from fields like title and description, which form a textual document [14]. Hybrid-structured reports involve the combined utilization of both textual and categorical features. Occasionally, they also incorporate execution-related information like stack traces or logs [36].

DBRD approaches are categorized into IR and machine learning (ML) based solutions. The approaches used by different studies over the past years have been discussed below in detailed.

2.2 Information Retrieval Approaches

In past research, IR has been applied for the automated identification of DBRs. IR approaches allow users to quickly access information from large datasets. These techniques can scale to handle a wide range of data sources and types, making them suitable for DBRD. However, a notable drawback of employing IR techniques is its disregard for the contextual placement of words and its sole reliance on word frequency within the vector representation. The studies, that rely only on IR techniques, face limitations in capturing the semantic similarity [37]. Such limitations can lead to challenges in identifying duplicates that are semantically similar.

2.2.1 Count Based Approaches

Count-based approaches in NLP involve quantifying the frequency of terms in a corpus of text. These approaches are based on the idea that the frequency of certain words or terms can provide significant information about the content and characteristics of the text data [38].

The earlier research into the identification of DBRs was done by Runeson et al. [39]. This study employed a feature vector for documents using the Bag of Words (BoW) technique to characterize bug reports. The study employed cosine similarity between report vectors to detect potential duplicates by assessing content similarity. The accuracy attained in identifying duplicates within an exclusive dataset from Sony Ericsson Mobile Communications reached a maximum of 40%. Nonetheless, this approach maintains its status as a straightforward and computationally economical technique.

The techniques presented by Wang et al. [40] and Jalbert and Weimer [41] used TF-IDF to transform textual information into vectors. These vectors were then used to calculate similarity scores, facilitating the identification of DBRs. Wang et al. achieved detection accuracies of 71% for Eclipse and 82% for Firefox datasets.

In contrast, Jalbert and Weimer achieved an accuracy rate of 43% for the Mozilla dataset by incorporating bug severity, date fields, textual similarity, and graph data into their classifier. The technique can be further improved by separately handling non-natural text during the duplicate retrieval process.

Sun et al. [26] enhanced the earlier technique proposed by [40] using sparse vectors that rely on Inverse Document Frequency (IDF) to determine the importance of individual words within a bug report. Subsequently, they implemented a Support Vector Machine (SVM) to assess whether a pair of reports were duplicates or not. The outcomes of their investigations revealed accuracy rates ranging from 50% to 70% across three distinct datasets i.e. Eclipse, Firefox, and Open Office.

2.2.2 BM25 Approach

BM25 (Best Matching 25), is a ranking function used in (IR) and text mining. It is an improvement over the traditional TF-IDF weighting scheme and is specifically designed for ranking bug reports based on their similarity to a given query. There are several variants and adaptations of the BM25 ranking function, few notable variants are BM25+, BM25F, BM15 etc [42].

In 2011 Sun et al. [27] addresses the challenge of accurately identifying duplicates by using both textual (unstructured) and categorical (structured) features. It introduces a retrieval function (REP) that assesses the similarity between bug reports. The study extends BM25F for textual similarity measurement and optimizes REP using two-round stochastic gradient descent. The research demonstrates a relative enhancement in the recall rate@k of 10-27% and an improvement in Mean Average Precision (MAP) values of 17-23% when compared to the approach previously suggested by Sun and colleagues [26].

Aggarwal et al. [43] devised a contextual technique by creating a vocabulary for eight different word lists, derived from general software engineering literature and technical documentation of four different projects. The bug reports were analyzed by comparing their title and description fields. This comparison was carried out

using BM25F, which includes assessments for both individual words (unigram comparison) and pairs of words (bigram comparison). For categorical fields, the binary rating system was employed, assigning a value of 1 when there was a match and 0 otherwise. Subsequently, contextual attributes were generated by employing BM25F similarity scores based on word lists extracted from bug reports. The research results in a classification accuracy of 90% for identifying DBRs across various datasets, including Mozilla, OpenOffice, Eclipse, and Android.

Hindle et al. [28] introduced a technique to prevent duplicates by continuously querying bug repositories. Their approach involved an updating search mechanism that provided suggestions for potential duplicates. The study focused on combining TF-IDF, BM25, and cosine distance for effective bug report matching. Tests were conducted on 12 datasets including OpenOffice, Mozilla, Eclipse, and Android. Results demonstrated a significant 42% reduction in DBRs. However, the approach's reliance on individual words might limit its effectiveness due to synonyms and similar terms.

Behzad et al. [30] introduced an efficient feature extraction model to enhance Duplicate Bug Report Detection (DBRD). This approach integrates new textual features based on term frequency and Inverse Document Frequency (IDF) aggregation in both uni-gram and bi-gram forms. The model was evaluated on datasets from Android, Eclipse, Mozilla, and Open Office, demonstrating a significant improvement in DBRD with a 75% recall rate. The incorporation of these features enhances the ability of the model to accurately identify and validate duplicate bug reports, offering potential benefits for more effective bug tracking and resolution in software development.

Another study done by Neysiani et al. [31] uses Manhattan distance along with BM25 for bug reports similarity. The study serves the purpose of extracting different topics as contextual features. These features are then incorporated to enhance the effectiveness of bug report similarity detection in software systems. The technique was evaluated on four benchmark datasets, including Mozilla, OpenOffice, Android, and Eclipse. The proposed study achieved an accuracy ranging from 96% to 97% for the classification task.

2.3 Machine Learning Approaches

Recently, ML approaches have been effectively utilized for the identification of DBRs. They offer substantial advantages in terms of semantic understanding, feature extraction, and accuracy in DBRD. However, it's important to note that all those approaches that use deep learning models, inherit both their advantages and limitations. These models excel at capturing semantic nuances and extracting relevant features from textual data, but they also require substantial training data, computational resources, and domain-specific adaptation making them infeasible for smaller projects [37]. These considerations must be carefully weighed when applying them to real-world tasks in software development.

2.3.1 CNN Based Approaches

Deshmukh et al. [16] utilized a deep Siamese model that takes hybrid features as input. This architecture incorporates three types of networks: MLP (Multilayer Perceptron), CNN and BiLSTM (Bidirectional long short-term memory). It produces a compact vector that represents the bug report and utilizes a triplet loss function during training with the goal of increasing the similarity among duplicates while reducing it among non-duplicates. This method was put to the test on three extensive datasets: Eclipse, NetBeans, and OpenOffice, resulting in an accuracy ranging from 72% to 82% for the classification task and a retrieval rate spanning from 50% to 81%.

Another similar study conducted by Kukkar et al. [13] introduced a DL model based on CNN. This CNN model employed a Siamese structure to capture the syntactic and semantic relationships among words in bug report content by examining their preceding and following words. The words were encoded using Word2Vec, with each word represented in a vector of 300 dimensions. In their study, the classification tasks yielded an average accuracy ranging from 85% to 99%, while the retrieval tasks achieved a Recall@20 rate spanning from 79% to 94%. Six different datasets are used for evaluation of the proposed approach.

Xie [15] introduced a framework that combines structured data, including specific domain-related characteristics like component and bug severity, with Convolutional Neural Networks (CNN). The textual content within bug reports is represented using word embedding vectors, which are then fused with these domain-specific attributes. In order to distinguish whether a pair of bug reports are duplicates, the model conducts a classification at its final layer, determining the duplicate status of these bug reports. This classification process unfolds within the concealed layers of the CNN, extracting latent features from the bug reports. This approach's efficacy was tested across four datasets: Hadoop, Hdfs, MapReduce, and Spark, resulting in classification accuracy ranging from 82% to 94%.

In 2020 He et al. [10] extended the utilization of CNN by introducing a Dual-Channel CNN (DC-CNN) technique. An innovative method for representing pairs of bug reports was introduced, which involved crafting a dual-channel matrix by merging two single-channel matrices, each representing an individual bug report. These pairs of bug reports were then input into a CNN model to capture the contextual relationships between them and classify a pair of bug reports as duplicates or non-duplicate. The evaluation of this approach was conducted on three extensive datasets from three open-source projects, OpenOffice, Eclipse, and NetBeans, along with a larger combined dataset. The classification accuracy achieved was 94%, 96%, 95%, and 95%, respectively, for these datasets.

2.3.2 BERT Based Approaches

BERT (Bidirectional Encoder Representations from Transformers) stands as a pioneering pre-trained deep learning model for natural language processing (NLP), adept at comprehending contextual relationships and word meanings within text [44]. Building upon BERT's success, various adaptations and variants, including RoBERTa, GPT-3, and T5, have emerged, pushing the boundaries of NLP capabilities. These models have demonstrated remarkable performance across a diverse array of NLP applications, showcasing the continuous evolution and impact of transformer-based architectures in the field.

Rocha et al. [32] proposed a system for detecting DBRs based on semantic context using a Siamese architecture. This architecture consisted of two networks: BERT for processing textual features and MLP, LDA for handling categorical features and topic distributions respectively. The model was trained using a Quintet Loss function, optimizing the similarity of duplicates reports and minimizing it between unique bug reports. The study validated the approach on NetBeans, and Eclipse datasets, achieving an average recall rate of 85% for retrieval task and an AUROC of 84% for classification tasks. However, it's worth noting that using an attention-based model like BERT is computationally more expensive than many other methods.

Messaoud et al. [14] proposed a self-attention-based Neural Language Model for detecting duplicate reports. The proposed framework uses unstructured data of bugs to generate corresponding BERT's words representation with 300 dimensions. Next the self-attention layer is used to identify the contextual relationship between the words. The output obtained is then fed into MLP layer to get the corresponding duplicate or non-duplicate category. The model is validated on three popular projects Thunderbird, Mozilla, and Eclipse and achieved an average recall rate of 91% for classification task. The approach does not use any structured information in DBRD.

Wu et al. [45] introduced a novel approach, CTEDB (Combination of Term Extraction and DeBERTaV3), for detecting DBR. CTEDB employs technical term extraction based on Word2Vec and TextRank algorithms to identify terms. It then calculates contextual similarity using Word2Vec and SBERT models and utilizes the DeBERTaV3 model for DBR detection. Experimental results demonstrate that CTEDB achieve a classification accuracy of 98.44% on mozilla core.

2.3.3 Hybrid Approaches

Recently Jiang et al. [37] performed a study to determine if well-established DL based methods outperformed classic IR based methods in the task of DBRD. They proposed a novel technique, combining IR and DL models.

For a more comprehensive computation of textual similarity. The experimental results revealed that the DL based method alone did not achieve high performance in comparison to IR based methods. However, the combined approach significantly improved the MAP metric of classic IR based methods, with a median improvement ranging from 7.09% to 11.34% and a maximum improvement of 17.228% to 28.97%.

A similar study was conducted by Zang et al. [46]. They introduce Cupid, an approach that combines the traditional DBRD method REP proposed by [27] with the advanced ChatGPT language model. ChatGPT is initially utilized for extracting key bug report information, which is then input into REP for duplicate bug report identification. Cupid's performance was evaluated against three existing approaches across three datasets Spark, Hadoop, and Kibana, achieving new state-of-the-art results with Recall Rate@10 scores ranging from 0.59 to 0.67 on all datasets.

Chauhan et al. [47] presented a framework called DENATURE, designed for the detection of DBRs and the identification of bug types. Duplicate bugs were identified using IR method, while ML classification techniques were employed to categorize bug reports by their type (Bug or feature). Experimental results indicated that the proposed framework achieved prediction accuracy levels of up to 88.81% on the Eclipse dataset.

In addition, the dual-tier approach introduced by Akilan et al. [48] not only combined classification and clustering but also leveraged Latent Dirichlet Allocation (LDA) for clustering based on topics. The multimodal text representation techniques, including FastText (FT), Global Vectors for Word Representation (GloVe), and Word2Vec (W2V), were integrated into the model, enhancing its ability to capture diverse semantic relationships. Furthermore, a unified text similarity measure employing Cosine and Euclidean metrics was incorporated, enhancing its ability to capture diverse semantic relationships and providing a comprehensive evaluation of textual similarities. The model's effectiveness was tested on the Eclipse dataset, comprising over 80,000 bug reports, and yielded promising results with a Recall Rate (RR) of 67% for Top-N similar bugs.

2.3.4 Other Approaches

Banerjee et al. [33] employed three techniques: base text similarity approaches, time windows, and document-related factors. The study uses cosine similarity with group centroids and identifies the longest common subsequences. It compares either the title field or both the title and the summary fields of new reports with every prior report in the repository. Time windows and document-related factors play a crucial role in narrowing down the search space. The results were evaluated on the Eclipse, Firefox, and Open Office repositories, achieving a high initial recall of 70%.

Budhiraja et al [49] introduced the Deep Word Embedding Neural Network, model designed to ascertain whether a pair of bug reports constitutes duplicates. This model takes into account both the description and title of the bug reports, representing words using CBOW (Continuous Bag of Words) and Skip-Gram vectors. Classification is performed using a Sigmoid output layer and an MLP layer. The study reported a retrieval rate of 77% for duplicate identification in the retrieval task and achieved a classification accuracy of 94% when applied to the Open Office and Firefox datasets. Notably, the study does not incorporate the categorical features of bug reports.

Another study by Ebrahimi et al. [36] used stack traces from bug reports to detect duplicate bugs. The study employed a Hidden Markov Model (HMM) for the automatic detection and classification of DBRs. The approach was validated using Firefox and Gnome datasets resulting in an average recall rate of 71.5%. However, it's worth noting that the approach does not work for bug reports that don't have prior duplicates in the repository.

Mahfoodh et al. [50], introduced two distinct similarity metrics for the identification of duplicated bugs, employing the Word2Vec and natural language processing technique within the Tensorflow framework. An experimental comparison was conducted using bug report descriptions from eight different software components within the Mozilla Core dataset.

Various sentence types were selected from the duplicated bug category records to

evaluate and discuss the accuracy of each component. The study also incorporated an earlier method to calculate software risk values from duplicate records and predict bug-fix times for components not identified as duplicates by the Word2Vec approach. The study's findings demonstrated a maximum precision accuracy of 99.89% for components correctly identified as duplicates by the employed approach.

Neysiani and Babamir [35] conducted research focused on evaluating the most effective automated bug report detection methods, considering both information retrieval (IR) and machine learning (ML) solutions. They utilized various classifiers, including K-Nearest Neighbors (KNN) and Logistic Regression (LR), coupled with algorithms such as Support Vector Machine (SVM), Information Gain Ratio (IGR), Chi-Square (CS), Gini Index (GI), and Principal Component Analysis (PCA) for document representation. Their findings indicated that ML-based approaches outperformed IR-based methods, especially when hybrid-structured information was employed in all experiments. In terms of retrieval tasks, the ML-based methods successfully retrieved 51% of replicas. It's important to note that the study was exclusively evaluated using the Android dataset.

2.4 Discussion on DBRD Approaches

For many years IR based approaches were the state-of-art techniques for DBRD. These approaches used Bag of Words (BOW), Term Frequency-Inverse Document Frequency (TF-IDF), and BM25 models for representing bug reports as vectors [32]. They often rely on the statistical analysis of term frequencies within bug reports to identify potential duplicates. BOW, for instance, characterizes bug reports as document feature vectors and uses cosine similarity to measure the similarity between these vectors. While these methods exhibit computational efficiency, they have limitations in capturing contextual relationships among terms.

In contrast, DL based approaches represent a more recent trend in DBRD. These approaches harness the power of neural networks to learn complex patterns and

semantic relations from textual data.

Prominent DL models, including CNN, Dual-Channel CNN (DC-CNN), BERT with Siamese architecture, and HMM, have been applied to DBRD.

While CNN and models like BERT have shown promise in the domain of DBRD, they are not without their limitations. The computational demands of DL models like CNN and BERT are substantial. A study [37, 51] shows a comparison of IR and DL approaches. The results indicate that REP, introduced in 2011, surpasses more recent, sophisticated deep learning-based approaches, emphasizing its importance as a robust benchmark.

DBR contains both structured and unstructured data. Detecting duplicate reports is a multifaceted task that involves not only the lexical matching of terms but also a deeper understanding of the semantic context within textual data. IR techniques are well-suited for lexical search. However, to achieve a more comprehensive and nuanced understanding of textual content, DL techniques come into play. DL models, such as BERT, excel at capturing semantic relationships and contextual similarities between bug reports. Combining the lexical precision of IR with the semantic understanding of DL models will lead to more effective and precise identification of DBRs in software repositories.

TABLE 2.1: Comparative Analysis of Existing Techniques

Ref.	Dataset	Techniques	Result
[39]	Sony Ericsson Mobile Commu- nications	weight = 1 + log(frequency), Time window, Cosine Similarity,	RR@ k= 31% - 42%
[26]	Mozilla, OpenOffice, Firefox, Eclipse	weight= log2(frequency), Support Cosine Similarity, Vector Machine,	Recall= 100%, F1= 14.8%
[40]	Firefox, Eclipse, Mozilla	Cosine Similarity, TF-IDF	Recall rate @ k= 84% - 93%, 67 - 93%
[41]	Mozilla Firefox	weight= 3 + 2 log2 (frequency), Graphic Cluster Algorithm, Linear Regression, Cosine Similarity	RR @ k= 25% - 50%
[27]	Eclipse, Mozilla	Gradient descent, BM25F	RR @ k= 36% - 73%, 43% - 76%

- [28] Android, AppIn-ventor, Bazaar, Cyanogenmod, Eclipse, K9Mail, Mozilla, MyTrack, OpenSTack, Tempest, Osmand TF-IDF with 42% duplicate prevention Cosine Distance, BM25
- [31] Android, Mozilla, Eclipse, Openoffice Manhattan Distance Similarity, Decision Tree, Naïve Bayes, Neural Networks Accuracy= 99.47%, 96.58%, 97.14%, 96.87%
- [43] Android, Eclipse LDA, Labeled LDA, Cosine Similarity Recall rate @k=95.09%, 95.47%
- [10] NetBeans, Eclipse, OpenOffice, Combined DC-CNN, Word Embedding (Word2Vec) Classification: 94-95%
- [13] Mozilla, NetBeans, Eclipse, OpenOffice, Gnome, Firefox, Combined Siamese, CNN, Word Embedding (Word2Vec) Classification: 85-99 % Retrieval: 79-94%
- [15] Hdfs, Hadoop, Spark, MapReduce CNN, Context, Word Embedding (Glove, Word2Vec, Random) Classification: 82-94%

-
- [16] OpenOffice, Siamese, MLP, Classification: 72-
Eclipse, Net- Bi-LSTM, CNN, 82% Retrieval: 50-
Beans, Com- Glove 81%
bined
- [32] OpenOffice, LDA, BERT Classification: 84%
Eclipse, Net- Retrieval: 85%
Beans
- [35] Android KNN, PCA, LR, Classification: 97%
CS, IGR, GI Retrieval: 34-51%
- [49] Firefox, MLP, Word Em- Classification: 60-
OpenOffice bedding (CBOW, 94% Retrieval: 21-
SkipGram) 77%
- [34] Firefox, Eclipse Longest Com- RR @ k= 42% -
mon Subsequence 75%, 55% - 83%
(LCS), Match
size within gr
oup weight, Time
window
- [47] Spark, Hadoop, ChatGPT, REP RR @ 10= 59% -
Kibana 67%
- [47] Eclipse Logistic Regres- Auucracy 88.81%
sion, KNN, Naive
bayes, AdaBoost,
Decision Tree,
SVM
-

Chapter 3

Research Methodology

This chapter discusses the conducted research methodology for this study. It encompasses three phases. Phase 1 encompasses the conceptualization of the research idea. Phase 2 involves the planning of the research, including the selection of benchmark datasets. In Phase 3, the implementation, evaluation and outcomes of the research are discussed. Figure 3.1 shows the proposed research methodology.

3.1 Problem Identification

In large software projects, bugs are reported and maintained through specialized software generally called BTS. Duplication occurs when the same bug is reported by multiple users. The issue with bug reporting is to determine if a new bug report is a duplicate of an existing bug and to find the original or similar bugs from the bug repository. As the reports are written in natural language, a tester either relies on their knowledge of the bug repository or must perform a series of manual searches in order to find the DBRs. Manually identifying the DBR requires a considerable amount of effort both in terms of time and cost, and requires complete knowledge of the bugs [20]. Automating the process of DBRD becomes imperative to alleviate the time and effort in manual identification, offering an efficient solution for DBR. The problem of DBRD was discussed in detail in Chapter 1 under section 1.1.

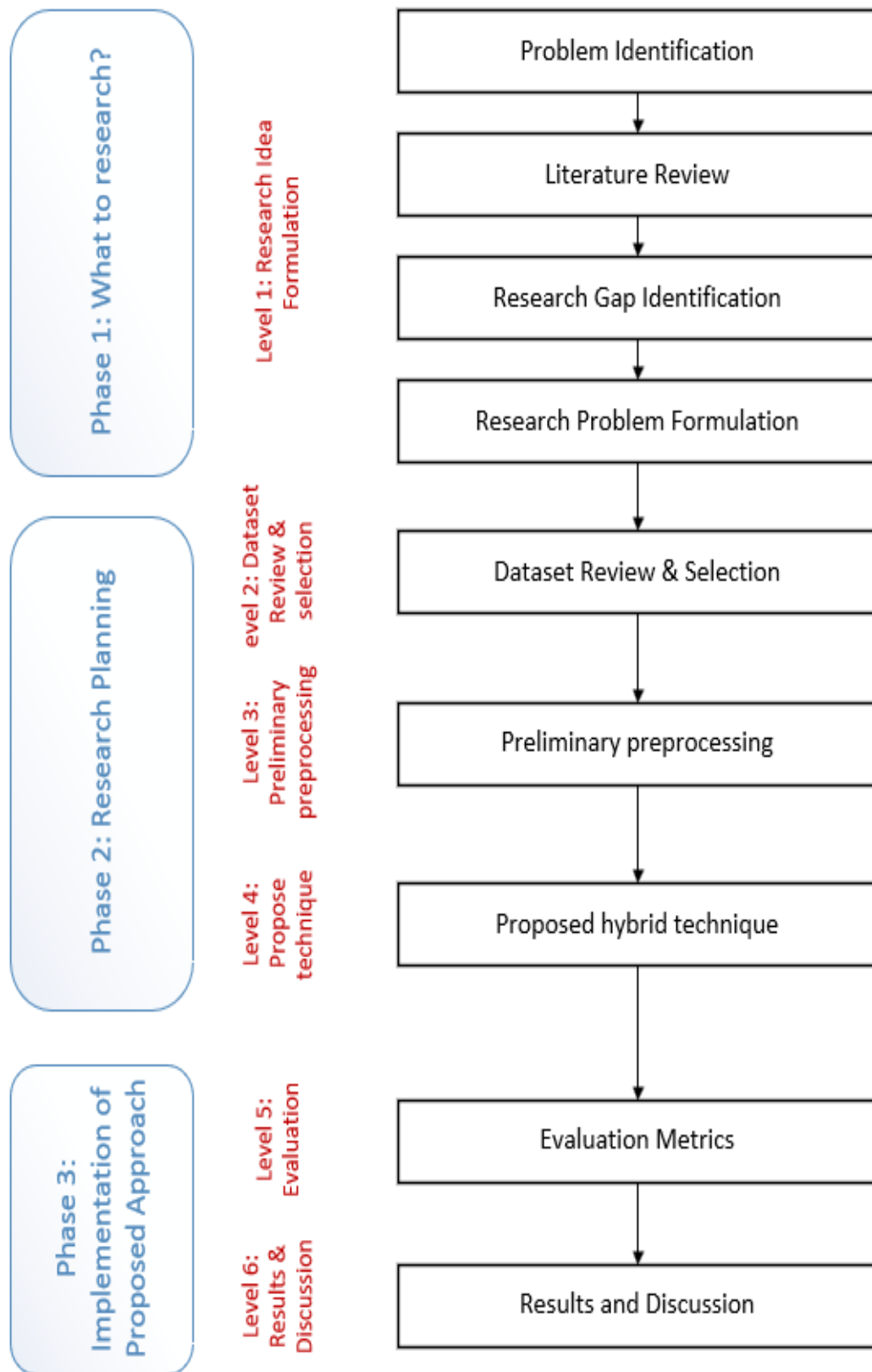


FIGURE 3.1: Proposed research methodology for retrieval of DBRs

3.1.1 Literature Review

The literature review is categorized into two primary sections: studies utilizing IR techniques for DBRD retrieval and studies employing ML based techniques. The IR techniques encompass count-based approaches, specifically Bag-of-Words (BoW), TF-IDF, and BM25. On the other hand, ML-based studies are categorized into four distinct groups: those employing the CNN model for DBRD, those utilizing the BERT model, those employing hybrid approaches combining IR and ML, and those exploring various other innovative techniques. Figure 3.2 illustrates the studies that have employed the aforementioned approaches.

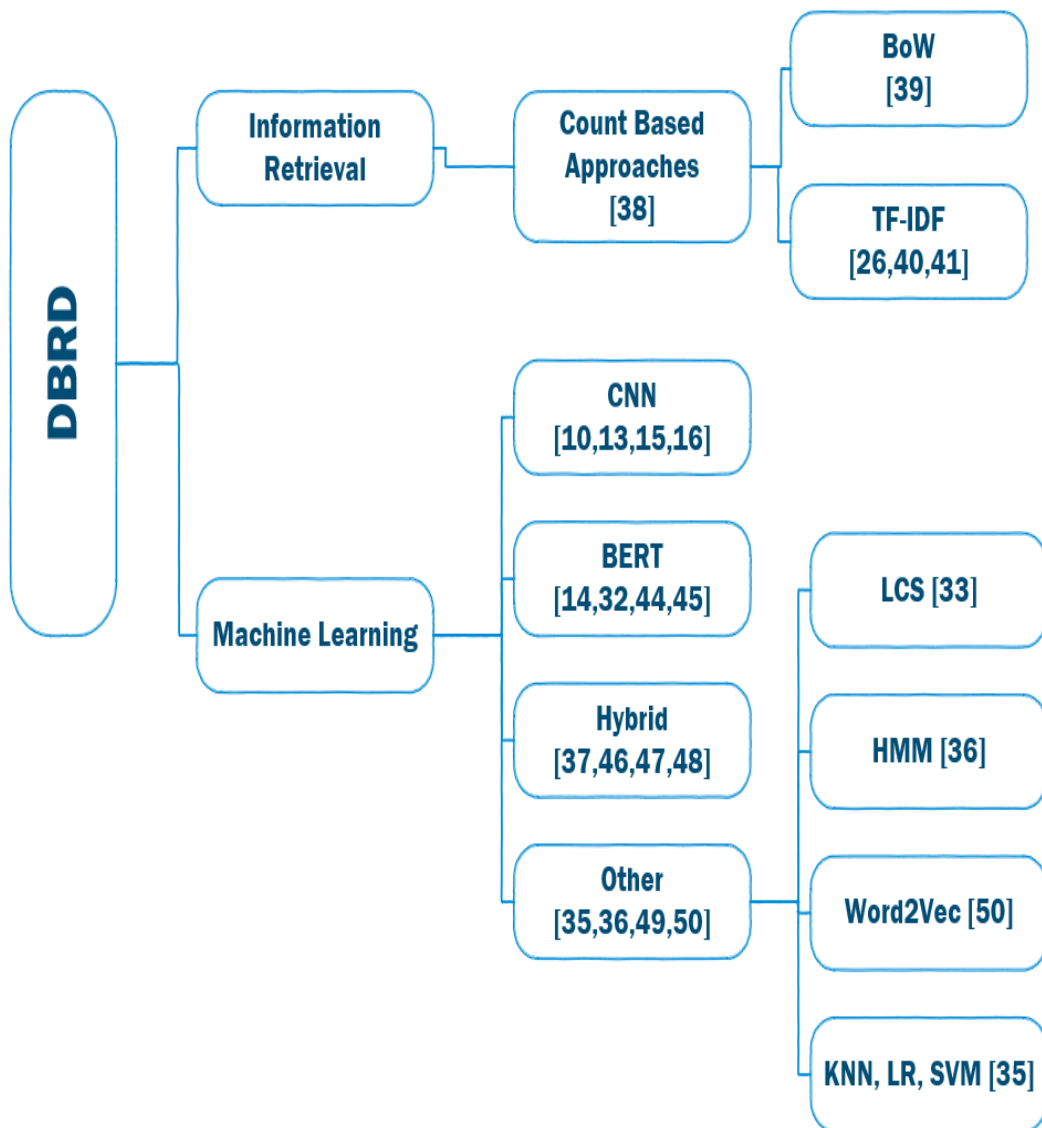


FIGURE 3.2: Literature review hierarchy

3.1.2 Research Gap Identification

From the literature, it is concluded that:

- IR techniques are fast as compared to DL techniques for lexical search, but do not capture contextual meaning.
- DL models like BERT are efficient in capturing the semantic similarity but are computationally expensive.

3.1.3 Problem Formulation

This study addresses the trade-off between traditional IR techniques, such as TF-IDF and BM25, known for their speed but limited semantic understanding, and DL models like CNN, BERT, and Word2Vec, which excel in capturing semantic nuances but come with high computational costs. Our aim is to conduct a comparative assessment to explore simpler techniques with lower processing time, seeking a balance between computational efficiency and semantic accuracy in DBRD.

3.1.4 Dataset Review and Selection

Each software has its own bug dataset that describes the software's specific nature. Consequently, the types of bugs, issues, and feature requests reported for one project may differ from those reported for another. Some of the famous datasets in the field of DBRD are discussed below.

3.1.4.1 Eclipse dataset

The Eclipse dataset is a widely recognized benchmark in the field of DBRD. It consists of a diverse range of bug reports extracted from the Eclipse open-source project. This dataset is characterized by its substantial size and extensive history, spanning various versions and components of the Eclipse platform. Researchers

favor the Eclipse dataset due to its rich and diverse bug reports, which encompass a wide spectrum of software development issues. Table 3.1 shows eclipse dataset statistics.

TABLE 3.1: Eclipse Dataset statistics

Eclipse Data statistics	
Total Bug Reports	110181
Duplicate Reports	29037
Components	927

3.1.4.2 OpenOffice dataset

The OpenOffice dataset is notable for its use in research related to DBRD. It originates from the Apache OpenOffice project, which provides a comprehensive office suite. This dataset is valued for its well-structured bug reports and their associated metadata, contributing to the robustness and reliability of experiments in DBRD research. Table 3.2 shows eclipse dataset statistics.

TABLE 3.2: OpenOffice Dataset statistics

OpenOffice Data statistics	
Total Bug Reports	111121
Duplicate Reports	19785
Components	15

3.1.4.3 Firefox

The Firefox dataset is another prominent dataset employed for DBRD studies. This dataset exhibits characteristics of real-world bug repositories and captures the complexities of managing a high-profile open-source software project. Table 3.3 shows eclipse dataset statistics. The Firefox dataset provides a diverse and comprehensive collection of bug reports, making it an ideal choice for evaluating and enhancing DBRD systems.

TABLE 3.3: Firefox Dataset statistics

Firefox Data statistics	
Total Bug Reports	115814
Duplicate Reports	35814
Components	52

3.1.4.4 Kibana dataset

The Kibana dataset represents bug reports from the Kibana open-source data visualization platform. It is characterized by its focus on log and event data analysis. Researchers find this dataset valuable for studying duplicate detection in the context of log analysis tools, where the ability to identify and manage duplicate reports is crucial for maintaining efficient log processing and troubleshooting. Table 3.4 shows eclipse dataset statistics.

TABLE 3.4: Kibana Dataset statistics

Kibana Data statistics	
Total Bug Reports	17015
Duplicate Reports	470
Components	-

3.1.4.5 Selected datasets

We have selected two benchmark bug datasets from large open-source datasets, OpenOffice and Eclipse, which have been widely utilized in previous studies for duplicate bug detection [10, 13, 16, 27, 28, 32, 40, 43, 49]. Published by [52], these datasets provide a substantial volume of bug reports, each accompanied by corresponding ground truth information regarding Duplicate Bug Reports (DBRs), ensuring their relevance and reliability in DBRD research. These datasets serve as valuable resources for training and evaluating DBRD models, offering diverse and

real-world examples to enhance the robustness and generalizability of the research findings.

3.1.5 Preliminary Preprocessing

The preliminary preprocessing phase includes collecting the data from the bugs repository. It also includes removing the extra tags that are associated with each bug report in the title field as shown in Figure 3.3. Some of the bugs have extra information attached to them like stack traces or part of the code where the error is located. This information needs to be removed before using the data. The preprocessed title is shown in Figure 3.4

```

❏ ['Option for CVS Server Name',
    '[CVS Core] Should support customized server strings',
    'DCR: Provide a hide option for workbench Views (1GET83R)',
    '[CVS Core] server .cvsignore file should be considered by client (1GCC6MB)',
    'Preference option Open a projects default perspective???' (1GE8I5X)',
    'Server Error While Releasing a File (1GI93G7)',
    '[CVS EXTSSH] cannot use RSA key for SSH connection (1GKKBJP)',
    'Resource name in window title (1GDSL4M4)',
    'Stream name on Resource History (1GHQHY5)',
    'All server contact should support low level progress/cancelation (1GF84JX)']

```

FIGURE 3.3: Bugs title with tags

title
Sync does not indicate deletion
Usability issue with external editors
how can we support
need better error message if catching up over ...
API - IResource.setLocal has problems

FIGURE 3.4: Bugs title without tags

3.1.6 Proposed Hybrid Model for DBRD

The proposed Duplicate Bug Report Retrieval (DBRR) model uses both IR and DL techniques to identify the DBRs. Our approach consists of two steps. First, we retrieve duplicate bug reports using BM25 a popular IR model that works on lexical matching of the words in a bug report. The top 100 duplicate reports are then reranked using a semantic model SBERT based on semantic similarity. Figure 3.5 shows the overview of our proposed approach.

3.1.7 Preprocessing

The preprocessing of bug datasets is an important step in preparing the data for efficient analysis and modeling. This process consists of two key phases: feature selection and NLP. Each of these phases are discussed below in detail.

3.1.7.1 Feature Selection

Bug reports consist of several fields that provide essential information for understanding and addressing software bugs. An example of a bug report from Eclipse dataset is shown in Table 3.5. The report contains both structured and unstructured files. These fields serve as foundational elements in bug reports, helping developers and testers understand, reproduce, and address software problems efficiently.

Feature selection in this study is guided by insights from the existing literature, specifically drawn from the comprehensive review of prior research in the field. Bug reports contain both structured and unstructured information. The unstructured data includes the title and description fields, which are crucial for identifying duplicates as they provide descriptive insights into the nature of the bug. A study conducted by Zhang et al. [51] has underscored the importance of these two features in duplication identification, while few other studies [14, 39, 49] used only title and description fields for DBRD.

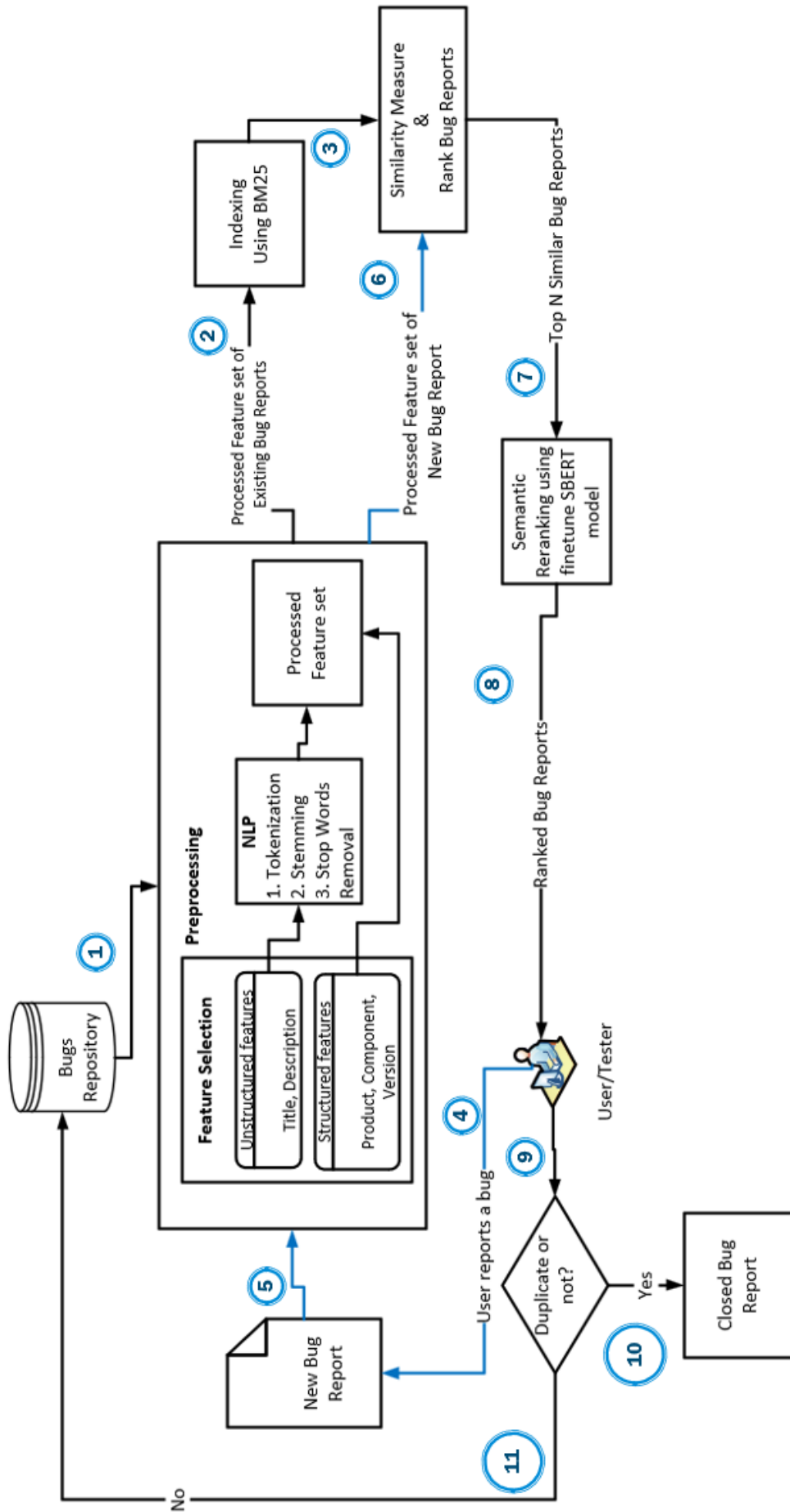


FIGURE 3.5: Proposed approach for DBRD

TABLE 3.5: Eclipse - Bug report format [52]

Eclipse - Bug Report	
Bug_id	232732
Product	PDE
Component	API Tools
Short_description	Preference page does not use dialog font
Status	Verified
Resolution	Fixed
Dup_id	[]
Priority	P3
Severity	Normal
Version	3.4
creation_ts	19/05/2008 5:01:00 am
delta_ts	20/05/2008 5:48:00 pm
Description	3.4 RC1 - go to General > Appearance > Colors and Fonts and change the dialog font. - Close and reopen the preferences, go to the API Errors/Warnings page. Tabs and control descriptions are still shown with the default font

Many studies [10, 13, 15, 16, 32] have adopted a hybrid feature set that combines both structured and unstructured attributes. In our study, we follow suit by amalgamating both unstructured (title and description) and structured fields (product, component, version). This hybrid approach allows us to leverage the strengths of both types of data for more accurate duplicate bug report detection.

3.1.7.2 Natural Language Processing

The bug dataset comprises both structured and unstructured features. The unstructured features encompass the title and description fields, which are expressed in natural language. In the context of NLP, three essential preprocessing steps

are performed: tokenization, removal of stop words, and stemming. Each of these steps is elaborated below.

3.1.7.3 Tokenization

Tokenization is the initial phase, entailing the segmentation of text into meaningful fragments referred to as tokens. These tokens may encompass words or sub-words, contingent upon the specific objective. In our context, we've elected to partition title and description into words, using Natural Language Toolkit (NLTK1) – the most widely recognized and extensively employed library for NLP [53]. Figure 3.6 shows an example of tokenization of title field in a bug report.

```
☞ Title: Set hover/disable icons for actions
   Title Tokens: ['Set', 'hover/disable', 'icons', 'for', 'actions']

   Title: CTRL + C does not work in New CVS
   Title Tokens: ['CTRL', '+', 'C', 'does', 'not', 'work', 'in', 'New', 'CVS']
```

FIGURE 3.6: Tokenization of title using NLTK

3.1.7.4 Stemming

Stemming involves the transformation of words into their foundational or root forms. The benefit of stemming lies in its capacity to diminish vocabulary dimensions. We conducted stemming through the application of the Porter stemmer algorithm (Porter, 1980), which modifies all terms within a text into their root forms. This stemming process was implemented on all title and description fields. Figure 3.7 shows an example of stemming on bug title field.

3.1.7.5 Stop words removal

Stop words represent the most prevalent terms within a language. As these words lack significant meaning, it's essential to eliminate them from the text to attain

```

↳ Title: Set hover/disable icons for actions
  Title Tokens: ['Set', 'hover/disable', 'icons',
  Title Stemmed Tokens: ['set', 'hover/dis', 'icon']

  Title: CTRL + C does not work in New CVS
  Title Tokens: ['CTRL', '+', 'C', 'does', 'not',
  Title Stemmed Tokens: ['ctrl', '+', 'c', 'doe', '

```

FIGURE 3.7: Stemming of tokenized title

precise measurements. To exclude stop words from the title and description fields, we've employed the NLTK library, which encompasses a compilation of stop words. NLTK cross-references its own stop word list with the tokenized inventory and subsequently carries out the process of eliminating stop words from the text collection. Figure 3.8 shows an example of stop word removal on a tokenized title.

```

↳ Title: Set hover/disable icons for actions
  Title Tokens: ['Set', 'hover/disable', 'icons', 'for', 'actions']
  Title Stemmed Tokens: ['set', 'hover/dis', 'icon', 'for', 'action']
  Stop Words Removal: ['set', 'hover/dis', 'icon', 'action']

  Title: CTRL + C does not work in New CVS
  Title Tokens: ['CTRL', '+', 'C', 'does', 'not', 'work', 'in', 'New']
  Title Stemmed Tokens: ['ctrl', '+', 'c', 'doe', 'not', 'work', 'in',
  Stop Words Removal: ['ctrl', '+', 'c', 'doe', 'work', 'new', 'cv']

```

FIGURE 3.8: Stop words removal of tokenized Title

3.1.7.6 Processed Feature Set

The final phase of preprocessing involves combining the selected structured features and the processed unstructured features into a unified dataset. The result is a processed feature set that contains both structured and textual data, ready for the DBRD.

3.1.7.7 BM25 Indexing

BM25 (Best Matching 25) is an IR model that is used for indexing and ranking of reports. It's an improvement over the earlier TF-IDF model. TF-IDF provides insight into the significance of terms within a document, encompassing two key elements: TF and IDF. Term Frequency gauges the frequency of a term's occurrence within a document. Given that document lengths vary, there's a possibility that a term appears more frequently in lengthy documents compared to shorter ones. To address this, the term frequency is often normalized by dividing it by the document length. The term frequency can be defined using Eq 3.12

$$\text{TF} = \frac{\text{No. of times a term appears in a report}}{\text{Total no. of terms in the report}} \quad (3.1)$$

IDF evaluates the significance of a term. In the computation of Term Frequency (TF), all terms are treated with equal importance. Nonetheless, it's recognized that specific terms like "is," "of," and "that" may exhibit high frequency but possess limited relevance. Consequently, it becomes necessary to reduce the influence of commonly occurring terms while elevating the significance of rare ones. This is achieved by calculating the IDF score using the formula in Eq 3.2

$$\text{IDF} = \log_{10} \left(\frac{\text{No. of reports}}{\text{No. of reports in which the term appears}} \right) \quad (3.2)$$

The table 3.6 illustrates an example of TF-IDF calculation on two bug report titles. Bug1: ['Option', 'CVS', 'Server', 'Name'] and Bug2: ['CVS', 'Core', 'support', 'custom', 'server', 'string'].

3.1.7.8 Similarity Measure and Ranking

DBRs are not only similar in title and description fields but also in categorical fields like component, product, version, priority etc. The retrieval function presented is given by the Eq 3.3

TABLE 3.6: Example of TF-IDF calculation

Terms	TF		IDF
	Bug1	Bug2	
options	1/4	0	$\log(2/1)=0.3$
cvs	1/4	1/6	$\log(2/2)=0$
server	1/4	1/6	$\log(2/2)=0$
name	1/4	0	$\log(2/1)=0.3$
core	0	1/6	$\log(2/1)=0.3$
support	0	1/6	$\log(2/1)=0.3$
custom	0	1/6	$\log(2/1)=0.3$
string	0	1/6	$\log(2/1)=0.3$

$$Score(d, q) = \sum_{i=1}^5 feature_i \tag{3.3}$$

The Eq 3.4 and Eq 3.5 define the textual similarity over the title and description fields computed using BM25. The Equations 3.6, 3.7 and 3.8 are based on equality of categorical fields component, product and version. These fields are used in calculating the textual and categorical similarity of DBRs. The bug reports are then ranked in decending order of the score. Figure 3.9 shows a top-N similar bug reports.

$$feature_{1(\text{title})}(br, qr) = BM25(br, qr) \tag{3.4}$$

$$feature_{2(\text{description})}(br, qr) = BM25(br, qr) \tag{3.5}$$

$$feature_3(br, qr) = \begin{cases} 1, & \text{if } br.\text{product} = qr.\text{product} \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

$$\text{feature}_4(br, qr) = \begin{cases} 1, & \text{if } br.\text{component} = qr.\text{component} \\ 0, & \text{otherwise} \end{cases} \quad (3.7)$$

$$\text{feature}_5(br, qr) = \begin{cases} 1, & \text{if } br.\text{version} = qr.\text{version} \\ 0, & \text{otherwise} \end{cases} \quad (3.8)$$

where br and qr are the bug report and query report respectively.



bug_id	title	score
198150	[hovering] Javadoc hover sometimes gets orphan...	44.246722
202423	[hovering] image in Javadoc hover	22.710841
122564	[hovering] Javadoc hover improvements	22.682313

FIGURE 3.9: Top-N similar bugs reports

3.1.8 Semantic Reranking

BM25 model ranks duplicate bugs based on lexical matching of terms. The purpose of semantic reranking is to rank bugs based on semantic similarity between bug reports. A fine-tuned SBERT model is used for semantic reranking of duplicate bug reports.

3.1.8.1 SBERT

Sentence-BERT (SBERT) is a model architecture designed to create semantically meaningful sentence embeddings. It builds upon the principles of the BERT architecture but with modifications tailored specifically for generating sentence embeddings rather than word embeddings. The core idea behind SBERT is to capture the

semantic similarity between sentences and used it for natural language processing tasks.

3.1.8.2 Fine Tuning SBERT Model

Pre-trained models are typically trained on large, generic datasets and designed to understand general language patterns. Fine-tuning allows us to adapt these models to perform specialized tasks. Instead of training a model from scratch, which can be data and resource-intensive, we start with a pre-trained model and adjust it for the specific task. This is especially useful when labeled task-specific data is limited. It also allows us to benefit from both the general knowledge captured during pre-training and the task-specific nuances required for high-performance applications. By leveraging pre-existing knowledge captured during pre-training on large, generic datasets, fine-tuning becomes a resource-efficient approach, particularly beneficial when labeled task-specific data is limited. This strategy ensures a balance between the general language understanding gained through pre-training and the task-specific adaptations necessary for achieving optimal performance in DBRD applications.

3.1.8.3 Data Preparation

The creation of the triple dataset involved leveraging an existing repository of bug reports to enhance the semantic search capabilities of the Sentence-BERT (SBERT) model. Each triple in the dataset was structured with an "anchor" bug report, which acted as a reference point, a "positive" bug report representing a duplicate of the anchor, and a "negative" bug report randomly selected from non-duplicate reports. This framework facilitated the model's understanding of semantic similarity and dissimilarity, crucial for accurate DBR identification. The dataset's composition was thoughtfully designed to encompass a diverse range of anchor-positive-negative combinations, striking a balance between comprehensive training and computational efficiency. Figure 3.10 below shows the triplet dataset

for model training.

	A	B	C
1	anchor	positive	negative
2	Do not scroll current line to top of page	Editor auto-scrolling should not put selected line at the very top	Version is unspecified. Unordered lists don't display properly in main content area
3	Awkward switching between editors when many open	[Editor Mgmt] Usability: open editors are difficult to manage	Provide Customized ODA Design Instance Validation
4	No busy cursor while launching in Scrapbook	No busy cursor if text operation is invoked from editor context menu	Implement a Text Editor to support *.Textile Files

FIGURE 3.10: Bugs triplet dataset

3.1.8.4 Select SBERT model

From the transformers encoder 'distilroberta-base' is used as the word embedding model. The model has 6 layers of Transformer blocks. Each Transformer block contains a multi-head self-attention mechanism and a feedforward neural network. This is in contrast to the original 'roberta-base,' which has 12 layers.

The hidden size or dimensionality of the 'distilroberta-base' model is 768. This means that the output of each Transformer block in the model has a dimension of 768. The hidden size determines the dimension of the contextual embeddings learned by the model.

3.1.8.5 Architecture

Individual layers were defined, with 'distilroberta-base' serving as the word embedding model. The layer was constrained to a maximum sequence length of 256. To achieve a fixed-size representation for an entire bug, a mean pooling layer was employed to consolidate the token embeddings. These layers were then integrated into a new Sentence Transformer model. Figure 3.11 shows the architecture of the SBERT fine-tuned model.

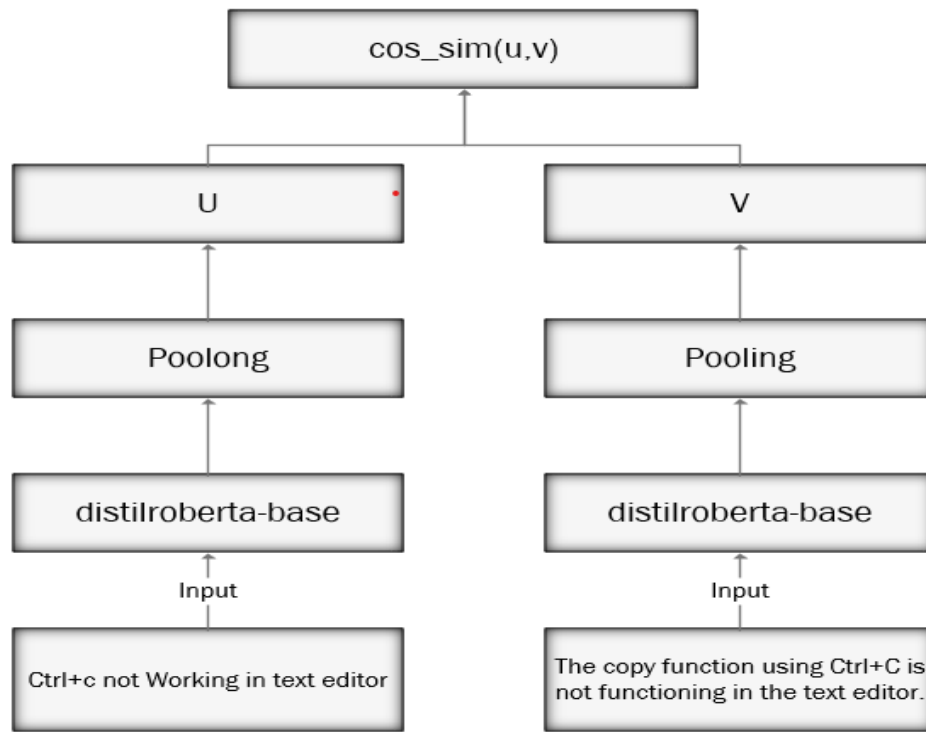


FIGURE 3.11: SBERT architecture

3.1.8.6 Loss Function

Triplet loss is a loss function used in machine learning, particularly in tasks related to metric learning and similarity or dissimilarity learning. It is designed to train models, often neural networks, to learn embeddings (vector representations) of data points in a way that encourages similar items to have embeddings that are close in distance, while pushing dissimilar items apart. Triplet loss is particularly important in tasks where measuring similarity or dissimilarity between data points is crucial. It is given by the Eq 3.9 Components of the triplet loss function are given below

$$L_{\text{triplet}} = \sum_{i=1}^N [d(a_i, p_i) - d(a_i, n_i) + \alpha]_+ \quad (3.9)$$

Where: - L_{triplet} represents the triplet loss. - i is the index for the training samples. - N is the total number of training samples. - $d(a_i, p_i)$ is the distance between the anchor sample and the positive sample. - $d(a_i, n_i)$ is the distance between the

anchor sample and the negative sample. - α is the margin or a constant value that determines the desired separation between the anchor-positive and anchor-negative distances. - The square brackets with the subscript $+$ indicate that the loss is calculated as zero if the value inside the brackets is less than zero, otherwise, it's the value inside the brackets.

- Anchor is the data point for which we want to learn an embedding.
- A positive example is a point that is similar or belongs to the same category as the anchor. The goal is to make the distance between the anchor and the positive example as small as possible.
- Negative example the data point that is dissimilar or belongs to a different category than the anchor. The goal is to make the distance between the anchor and the negative example as large as possible.

Figure 3.12 shows the working of the triplet loss function.

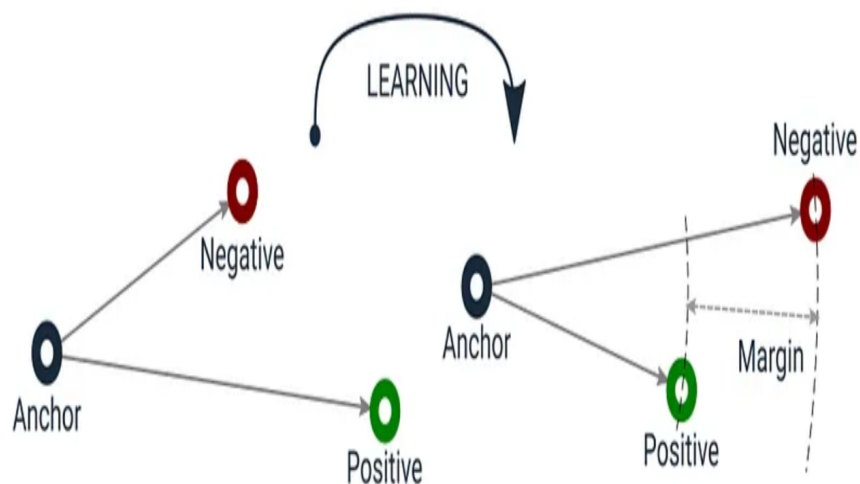


FIGURE 3.12: Triplet loss working

3.1.8.7 Similarity Measure

Cosine similarity is used as a similarity measure. It assesses the degree of content resemblance between two bug reports. In mathematical terms, this metric gauge

the cosine of the angle formed by the projections of two bug report vectors within a multi-dimensional space. This utilization of cosine similarity offers a distinct advantage, as it captures similarities between bug reports that might be distantly positioned in terms of Euclidean distance but still exhibit a closely aligned orientation. A reduced angle corresponds to heightened cosine similarity, while an increased angle corresponds to diminished cosine similarity. Existing literature attests to the widespread adoption of cosine similarity as a prominent measure for quantifying likeness among textual documents. This measure is useful in various domains including IR. The standardized formula for computing cosine similarity is given by Eq 3.10.

$$\text{Cosine Similarity}(q_n, b_m) = \frac{\sum_{i=1}^n q_i b_i}{\sqrt{\sum_{i=1}^n q_i^2} \sqrt{\sum_{i=1}^n b_i^2}} \quad (3.10)$$

In the above formula, the q and b represent query bug report and existing bug report vectors respectively. For understanding purpose, we have calculated the cosine similarity between two bug reports. Table 3.7 shows an example of calculating cosine similarity between two vectors. To do this, we start with two vectors. To determine their similarity, we have executed all the necessary steps.

3.1.9 Evaluation metrics

Evaluation metrics used for duplicate bug retrieval are as follows:

3.1.9.1 Recall@K

Recall@k is the proportion of relevant items found in the top-k recommendations. The measure is expressed by Eq 3.11. High recall@k values show that a significant percentage of the relevant items or documents have been successfully retrieved within the top K results. Previous works [16, 27, 32, 46, 48, 49] used Rcall@k as a metric to evaluate the quality of their proposed techniques in retrieving DBRs given a new report.

TABLE 3.7: Cosine similarity calculation example

Step	Calculation
Bug 1 Vector	(3.4, 0.1, 0.4, 3.1, 2.3, 3.5, 4.3, 0.5, 4.5, 0.6)
Bug 2 Vector	(3.2, 1.5, 0.4, 3.5, 2.6, 5.5, 3.3, 0.5, 4.6, 0.8)
$\sum_{i=1}^{10} B_{1_i}, B_{2_i}$	$(3.4 \cdot 3.2) + (0.1 \cdot 1.5) + (0.4 \cdot 0.4) + (3.1 \cdot 3.5)$ $+ (2.3 \cdot 2.6) + (3.5 \cdot 5.5) + (4.3 \cdot 3.3) + (0.5 \cdot 0.5)$ $+ (4.5 \cdot 4.6) + (0.6 \cdot 0.8)$ $= 10.88 + 0.15 + 0.16 + 10.85 + 5.98 + 19.25$ $+ 14.19 + 0.25 + 20.7 + 0.48 = 82.89$
$\sqrt{\sum_{i=1}^n B_{n_i}^2}$	$\sqrt{(3.4)^2 + (0.1)^2 + (0.4)^2 + (3.1)^2 + (2.3)^2}$ $+ (3.5)^2 + (4.3)^2 + (0.5)^2 + (4.5)^2 + (0.8)^2$ $= \sqrt{78.23} = 8.845$
$\sqrt{\sum_{i=1}^n B_{m_i}^2}$	$\sqrt{(5.5)^2 + (3.3)^2 + (0.5)^2 + (4.6)^2 + (0.8)^2}$ $+ (3.2)^2 + (1.5)^2 + (0.4)^2 + (3.5)^2 + (2.6)^2$ $= \sqrt{90.85} = 9.53$
	$\frac{\sum_{i=1}^{10} B_{n_i} \cdot B_{m_i}}{\sqrt{\sum_{i=1}^n B_{n_i}^2} \cdot \sqrt{\sum_{i=1}^n B_{m_i}^2}} = \frac{82.89}{8.845 \cdot 9.53} = \frac{82.89}{84.31} = 0.97$

$$Recall@k = \frac{1}{N} \sum_{t=1}^N \frac{duplicate}{1} \quad (3.11)$$

$$duplicate = \begin{cases} 1 & \text{if found at least one duplicate in } N \text{ bugs} \\ 0 & \text{otherwise} \end{cases}$$

where N value is the number of bug reports considered in the evaluated queries, and 'duplicate' signifies the condition used to identify instances where at least one duplicate bug is retrieved.

3.1.9.2 Mean Average Precision (MAP)

It is the mean of the average precision for all queries. Unlike precision, which only looks at the top- k results, MAP considers the entire ranking list. It provides a more realistic assessment of the system's performance by accounting for the order in which results are presented to users. Mathematically, this is given by Eq 3.12.

$$MAP(K) = 1 - \frac{1}{|K|} \sum_{i=1}^{|K|} \frac{1}{index_i} \quad (3.12)$$

Given a set K of duplicate reports, for each duplicate, the system continually retrieves masters in descendent order of similarity until the right master is retrieved, and records its index in the ranked list. where $index(i)$ is the index here the right master is retrieved for the i -th query

Chapter 4

Results and Discussion

This chapter states and discusses the results of DBR retrieval. We present a detailed account of our experimental outcomes, evaluate the strengths and weaknesses of different approaches, and draw meaningful conclusions that contribute to the broader field of software quality assurance.

4.1 Dataset Description

For the purpose of evaluating the proposed hybrid model, we used bug repositories from two extensive open-source projects: Eclipse and OpenOffice collected from [52]. Eclipse is an extensible multi-language software development environment written in Java [2]. The Eclipse dataset exhibits a versatile composition encompassing defects originating from 189 projects. These projects are further categorized into nine hundred and twenty-seven discrete components, facilitating a comprehensive representation of the software ecosystem.

Similarly, the second dataset, drawn from the OpenOffice, encompasses bug reports from nineteen distinct projects within the OpenOffice framework. This dataset is characterized by its segmentation into fifteen distinct components, thus enabling a holistic representation of the software's functional areas. Table 4.1 shows Eclipse dataset.

OpenOffice dataset statistics and table 4.2 shows the train and test split of the dataset.

TABLE 4.1: Eclipse and Open Office datasets statistics

Features	Eclipse	Open Office
Product (encompasses values from different eclipse projects)	189	19
Components (Product have multiple components)	927	15
Version (Each project has different versions)	547	11

TABLE 4.2: Train and test split

Dataset	Training reports	Test reports
Eclipse	20000	5000
OpenOffice	17000	5000

4.2 Duplicate Bug Report Retrieval Performance

In this section, we present the results of our proposed hybrid model. The model is a hybrid of IR and DL based techniques to determine whether a bug report is a duplicate or not. We assess our model performance for duplicate bug report retrieval using two datasets and two performance metrics.

Figure 4.1 shows recall@k results on Eclipse dataset using the proposed hybrid model. Starting with a recall@1 of 0.62, the model improved as the recommendation list grew, reaching a peak of 0.84 at recall@25. This suggests the hybrid model’s effectiveness in retrieving relevant DBRs, making it an effective approach for duplicate bug report retrieval systems.

Figure 4.2 shows the recall@k results on OpenOffice dataset and Eclipse dataset. The results show an increase in recall values as the top-K list of duplicates increases.

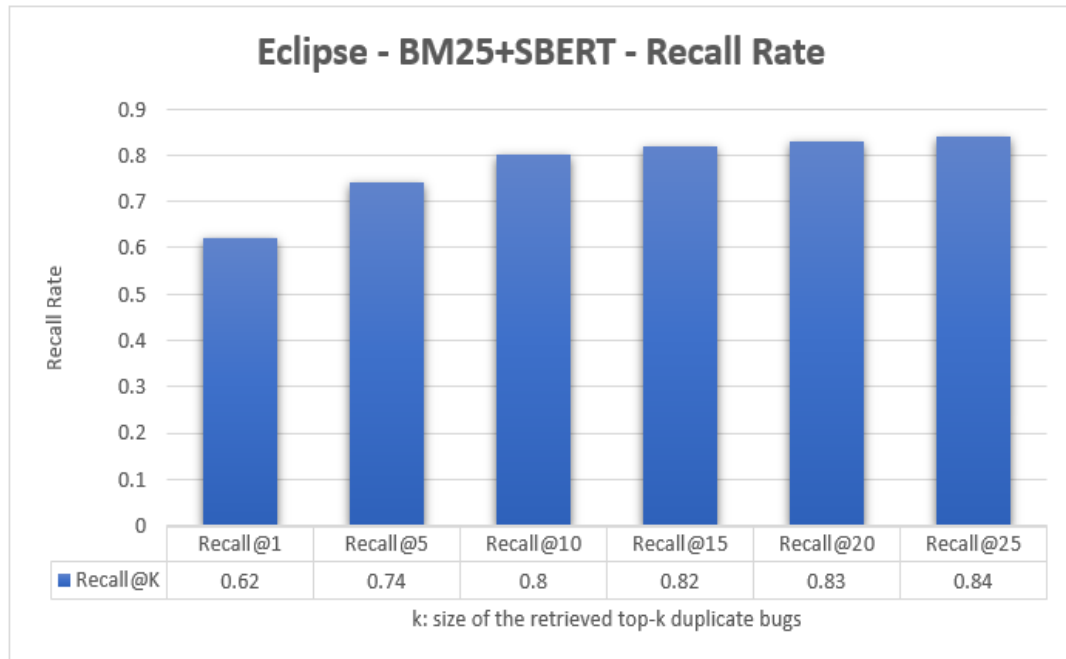


FIGURE 4.1: Recall@K on Eclipse dataset using BM25+SBERT

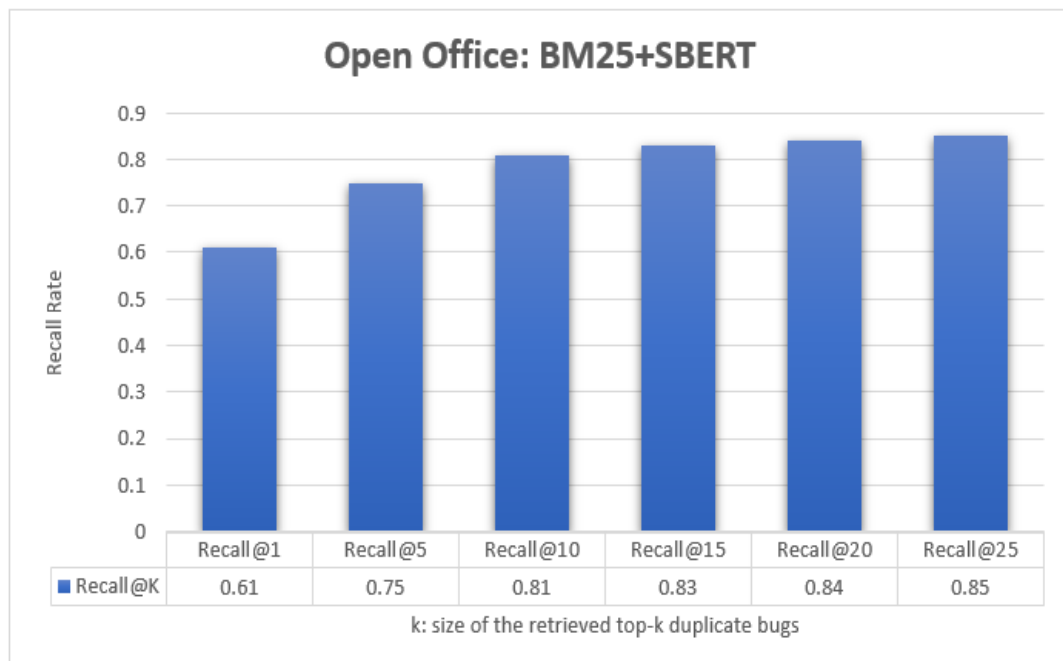


FIGURE 4.2: Recall@K on OpenOffice dataset using BM25+SBERT

Figure 4.3 shows the MAP across Eclipse and OpenOffice datasets. The graph shows that MAP of eclipse datasets is higher as compared to OpenOffice dataset.

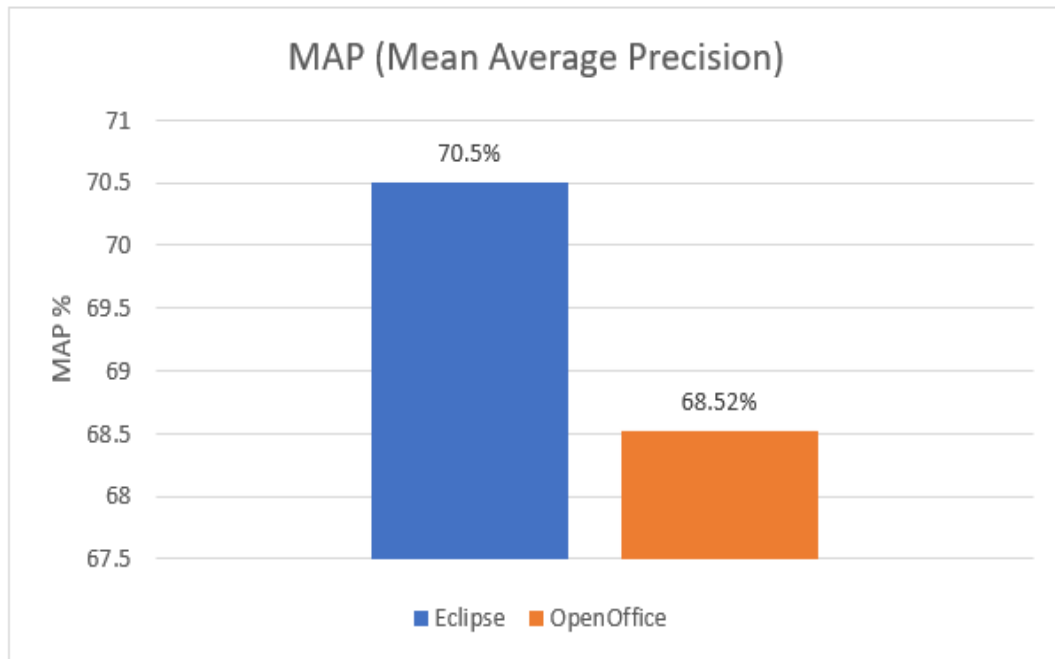


FIGURE 4.3: MAP of Eclipse and OpenOffice datasets

4.3 Comparison with Previous Bug Report Retrieval Techniques

In this section, we compare the experimental results of our proposed hybrid model with traditional models. These baseline techniques are based on IR and DL models proposed by Sun et al. [27] and Rocha et al. [32]. The comparison is done on Eclipse and OpenOffice datasets.

Figure 4.4 and Figure 4.5 provide a graphical representation of the performance of duplicate bug retrieval using two distinct models: BM25 and a proposed hybrid model. The visual comparison illustrates that the hybrid model consistently outperforms the traditional BM25 model in terms of recall rate. In the case of OpenOffice dataset, the relative improvement ranges from 32.81% and 38.89%. For the Eclipse dataset, the improvement is even higher ranging from 25.37% to 42.31%.

The primary rationale behind this marked improvement is attributed to the inherent characteristics of the BM25 model compared to the hybrid model. BM25 relies on lexical search, primarily focusing on text and keyword matching, while largely neglecting the semantic nuances of the content.

In contrast, the proposed hybrid model employs a more sophisticated approach by integrating both IR and semantic-aware SBERT models.

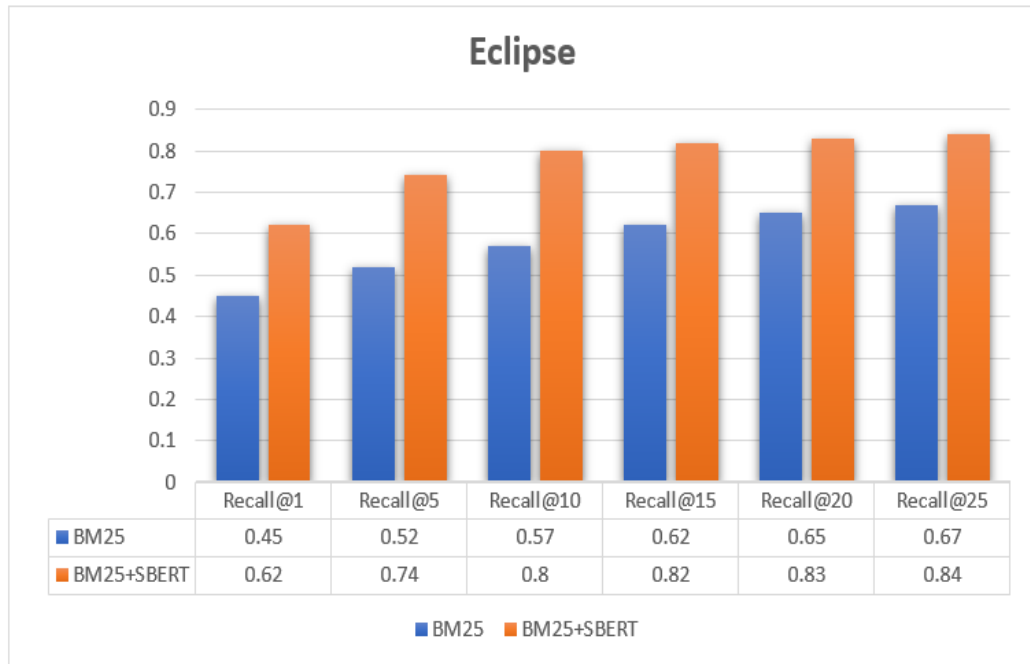


FIGURE 4.4: Comparison of recall@K for BM25 and BM25+SBERT on Eclipse dataset

Table 4.3 and 4.4 shows the results achieved by all approaches on two distinct datasets.

TABLE 4.3: Recall@K for Eclipse dataset on all approaches

	RR@1	RR@5	RR@10	RR@15	RR@20	RR@25
SiameseQAT [32]	0.57	0.73	0.78	0.80	0.82	0.83
BM25	0.45	0.52	0.57	0.62	0.65	0.67
BM25+SBERT	0.62	0.75	0.80	0.82	0.83	0.835

Figures 4.6 and 4.7 provide a comparative analysis of the performance of our hybrid model against BM25 and the baseline model in terms of recall@k. The results clearly demonstrate that our proposed method outperforms the others models.

in the task of retrieving DBRs. Specifically, for the Eclipse dataset, we observe an improvement in the recall rate ranging from 1.2% to 8.77%. Similarly, for

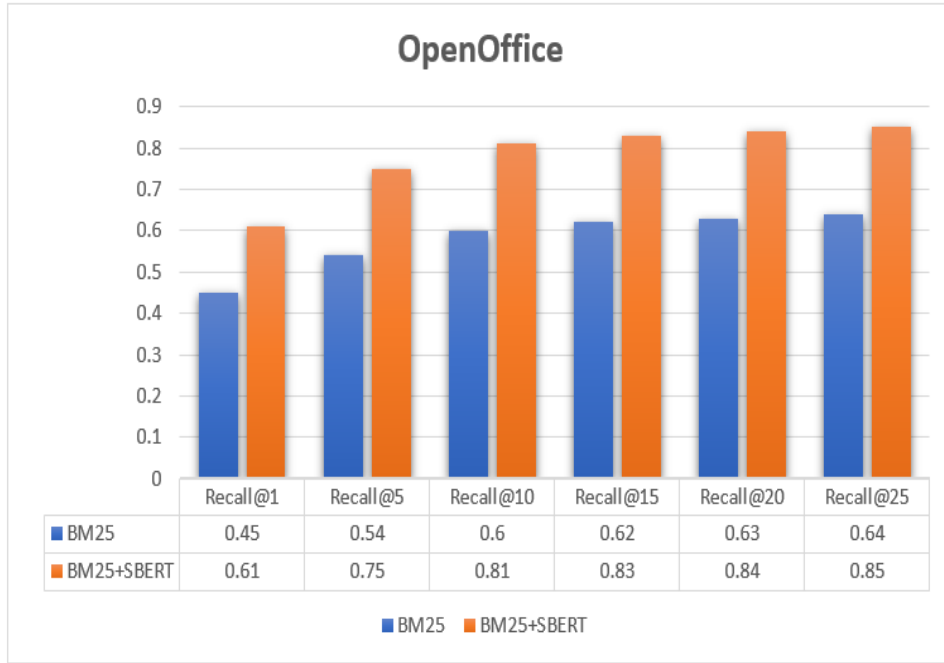


FIGURE 4.5: Comparison of recall@K for BM25 and BM25+SBERT on OpenOffice dataset

TABLE 4.4: Recall@K for Open Office dataset on all approaches

	RR@1	RR@5	RR@10	RR@15	RR@20	RR@25
SiameseQAT [32]	0.59	0.74	0.79	0.81	0.83	0.85
BM25	0.45	0.54	0.60	0.62	0.63	0.64
BM25+SBERT	0.61	0.75	0.81	0.83	0.84	0.85

the OpenOffice dataset, the recall rate shows an increase of approximately 3.39%. These findings underscore the effectiveness of our hybrid model in accurately identifying DBRs among the master bug reports.

4.4 Effectiveness of the Approach in Terms of Response Time

The performance of the proposed approach was assessed using Google Colab with an Intel(R) Xeon(R) CPU @ 2.20GHz, 4GB RAM, and a Tesla T4 GPU.

The program is running on the Windows 10 Operating System. Table 4.5 shows the per-sample processing time of three different approaches.

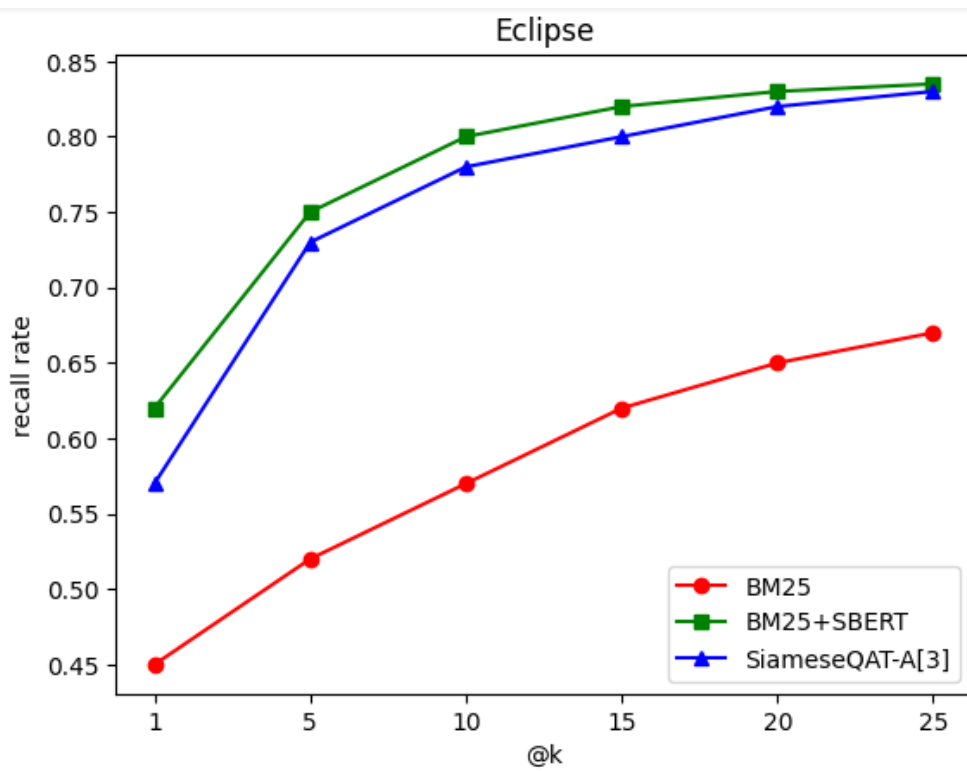


FIGURE 4.6: Recall rate at various values of K, and baseline approach in comparison to BM25 and BM25+SBERT.

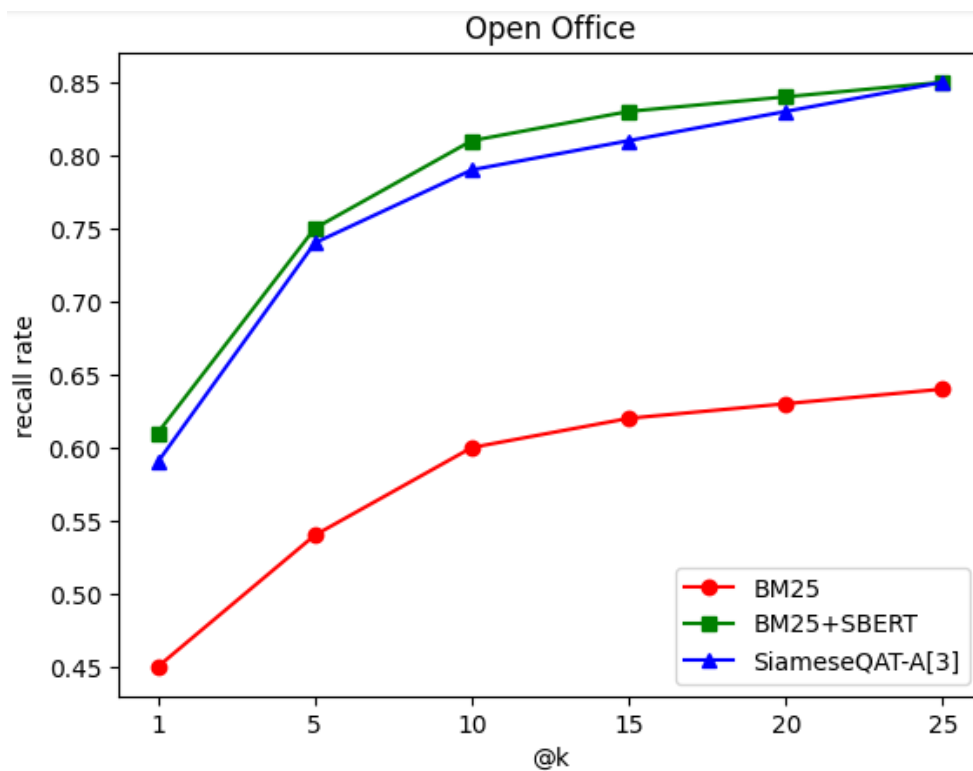


FIGURE 4.7: Recall rate at various values of K, and all baselines in comparison to BM25 and BM25+SBERT.

It shows that the processing time of the proposed approach is 5.88s for 2000K sample size. The proposed approach is approximately 1.608 times faster than the baseline approach.

TABLE 4.5: PSPT for different sample sizes

	10	100	500	1000	2000
M4 [48]	9.257	9.438	9.318	9.3468	9.468
BM25	0.9	0.97	0.98	1.0	1.1
BM25+SBERT	5.54	5.62	5.66	5.76	5.88

Lexical models like BM25 rely on simple heuristics and precomputed indices, making them computationally efficient. They can quickly scan the text to retrieve relevant information. In contrast, semantic models like BERT require more complex computations, such as neural network inference, attention mechanisms, and contextual embeddings, leading to higher per-sample processing times.

BERT, being a deep transformer model, has a large number of parameters and intricate architecture. It performs token-level computations on input sequences, including self-attention mechanisms, which can lead to longer processing times. The complexity of its architecture contributes to higher per-sample processing times compared to simpler models.

To achieve a trade-off between per-sample processing time and recall rate, a hybrid model is proposed. It uses BM25 and SBERT, BM25 offers faster processing time but might have lower recall rates due to its lexical nature. On the other hand, semantic models like BERT offer higher recall rates at the cost of increased processing times. Combining both, as seen in BM25+SBERT, can strike a balance.

BM25 rapidly identifies potential duplicate candidates, allowing deep learning models like BERT to concentrate on fine-grained similarity analysis. This not only expedites the detection of duplicate bug reports but also optimizes resource allocation, ensuring that the computational overhead of deep learning is reserved for cases where its sophistication is indispensable. Figure 4.8 shows the Per Sample Processing Time (PSPT) of proposed approach, BM25 and M4 technique proposed by [48]. It shows that the PSPT of the proposed approach is reduced by 3.5.

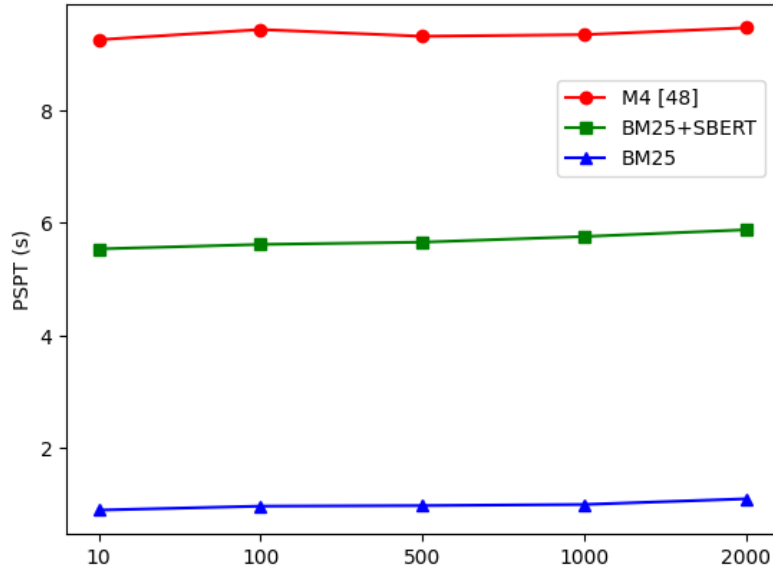


FIGURE 4.8: PSPT of BM25, BERT, and BM25+SBERT

4.5 Addressing Research Questions

To assess the effectiveness of our proposed hybrid system compared to the traditional techniques, we address the following two research questions: **RQ1:** Does the proposed approach using both sparse and dense vectors be more efficient in retrieving the DBRs?

The proposed hybrid model uses both BM25 and SBERT models for DBRs retrieval. The bug reports contain both structured and unstructured fields. NLP steps are applied to unstructured fields of the bug reports to get tokenized words. These tokenized words are then combined with categorical features to get a processed bug report feature set. The BM25 model is then used to index the bug reports. The model is based on TF-IDF which generates a sparse vector for bug representation. The sparse vectors are then used to retrieve duplicate bug reports based on the lexical matching of terms. The BM25 model is fast but fails to get semantically similar bug reports. A semantic-based reranker model is used to rerank the DBRs based on semantic similarity. Table 4.3 and 4.4 show a significant improvement in recall rate. Therefore it is concluded that the proposed hybrid approach using both sparse and dense retrieval vectors is effective in retrieving the possible DBRs.

RQ2: How can the processing time of previous DBRD approaches be reduced compared with the proposed approach?

The processing time of previous DBRD approaches can be significantly reduced compared to the proposed approach by leveraging a hybrid model that combines the strengths of both lexical and semantic models. Lexical models like BM25 offer fast processing times due to their reliance on precomputed indices and simple heuristics. However, they may have limitations in recall rates as they lack semantic understanding.

In contrast, semantic models like BERT or SBERT excel in capturing semantic information and achieving higher recall rates but require more complex computations, leading to longer processing times. The proposed hybrid approach, denoted as BM25+SBERT, takes advantage of this trade-off.

First, BM25 is used to rapidly filter out a subset of potentially relevant bug reports, benefiting from its computational efficiency. Then, semantic models like SBERT or BERT are applied to refine the results further, enhancing recall rates without imposing excessive processing time. This hybrid strategy achieves a balance between processing time and recall rate, offering a solution to reduce the processing time compared to previous DBRD approaches while maintaining or even improving recall rates.

Chapter 5

Conclusion and Future Work

5.1 Limitations

In this section, we discussed a few limitations that may affect the validity of our model.

5.1.1 Dependency on BM25 Model

It is important to acknowledge that the performance of the reranker model is intimately linked to the quality of top-N duplicate bug reports retrieved, which serves as input to the reranker model. Consequently, if the BM25 model fails to adequately prioritize and rank relevant bug reports, it can have a cascading effect on the performance of the subsequent re-ranker model. In instances where the BM25 model encounters challenges in accurately identifying or ranking duplicates, the re-ranker model may also face limitations in its ability to rectify these shortcomings.

5.1.2 Generalization Across Diverse Platforms

Although this study is evaluated on two diverse datasets. Variability in platforms, development practices, and bug report structures could hinder the generalizability

of the proposed system. The system's effectiveness in detecting duplicate bug reports may be influenced by platform-specific idiosyncrasies, potentially limiting its applicability in broader software development contexts. Therefore, it is essential to recognize that the system's performance may not be universally applicable, and future research may be needed to improve its adaptability.

5.2 Conclusion

In this study, a hybrid model for duplicate bug report retrieval using IR and semantic learning is proposed. The model utilizes both structured and unstructured information for the retrieval of duplicate bug reports. It uses BM25 a time-efficient IR model to retrieve top-N similar bug reports. A semantic model is used to re-rank the top-N duplicate bug reports. We have evaluated our approach on two sizeable open-source bug repositories Eclipse, and OpenOffice. The experimental results of the proposed hybrid model compared to BM25 show relative improvement ranging from 32.81% and 38.89% in the case of OpenOffice dataset and even higher improvement ranging from 25.37% to 42.31% in case of Eclipse. When compared with the baseline technique SimaeseQAT the proposed model shows a relative improvement of 4.6% to 12.16% with a per-sample processing time of 5.65s for 2000 sample size.

5.3 Future Work

5.3.1 Cross-domain Data

There are additional publicly accessible datasets such as Firefox, Thunderbird, Cassandra, and JDT mentioned in existing literature. This presents an opportunity to conduct experiments by amalgamating all these datasets into a comprehensive and unified dataset. This experiment would assess the model's ability to create a generalized representation of duplicate reports across different projects.

5.3.2 Optimizing Top-N Selection for Enhanced Duplicate Bug Report Detection

To further enhance the robustness of our approach, future work can focus on the systematic exploration of hyperparameter tuning techniques for determining the most suitable top-N value. This can include employing advanced optimization algorithms, such as Bayesian optimization or Optuna, to efficiently search for the optimal top-N parameter. Additionally, considering the potential variance in optimal top-N values across different software development projects, an adaptive approach that adjusts the top-N value dynamically based on project-specific characteristics could be explored.

5.3.3 Integration with Bug Tracking Tools

Creating seamless integrations with popular bug-tracking tools to facilitate adoption in real-world development environments.

Bibliography

- [1] S. Reddivari and J. Raman, “Software quality prediction: An investigation based on machine learning,” in *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pp. 115–122, 2019.
- [2] G. Tassej, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology*, 05 2002.
- [3] N. H. Adnan and A. D. Ritzhaupt, “Software engineering design principles applied to instructional design: What can we learn from our sister discipline?,” *TechTrends*, vol. 62, pp. 77–94, 2018.
- [4] I. Ushakova, Y. Skorin, and A. Shcherbakov, “Methods of quality assurance of software development based on a systems approach,” *Journal Name*, vol. Volume Number, p. Page Range, 2022.
- [5] Z. Li, X. Jing, and X. Zhu, “Progress on approaches to software defect prediction,” *IET Software*, vol. 12, no. 3, pp. 161–175, 2018.
- [6] N. Anwar and S. Kar, “Review paper on various software testing techniques & strategies,” *Global Journal of Computer Science and Technology*, pp. 43–49, 05 2019.
- [7] N. M. D. Febriyanti, A. A. K. O. Sudana, and I. N. Piarsa, “Implementasi black box testing pada sistem informasi manajemen dosen,” *Jurnal Ilmiah Teknologi dan Komputer*, vol. 2, pp. 535–544, 12 2021.
- [8] S. Supriyono, “Software testing with the approach of blackbox testing on the academic information system,” *IJISTECH (International Journal of Information System and Technology)*, vol. 3, pp. 227–233, May 2020.

- [9] “What is bug tracking system? — kissflow workflow - issue tracking.”
- [10] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, “Duplicate bug report detection using dual-channel convolutional neural networks,” in *Proc. 28th Int. Conf. Program Comprehension*, pp. 117–127, July 2020.
- [11] T. Akilan, D. Shah, N. Patel, and R. Mehta, “Fast detection of duplicate bug reports using lda-based topic modeling and classification,” in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1622–1629, 2020.
- [12] D.-G. Lee and Y.-S. Seo, “Systematic review of bug report processing techniques to improve software management performance,” *Journal of Information Processing Systems*, vol. 15, no. 4, pp. 967–985, 2019.
- [13] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K.-S. Kwak, “Duplicate bug report detection and classification system based on deep learning technique,” *IEEE Access*, vol. 8, pp. 200749–200763, 2020.
- [14] M. B. Messaoud, A. Miladi, I. Jenhani, M. W. Mkaouer, and L. Ghadhab, “Duplicate bug report detection using an attention-based neural language model,” *IEEE Transactions on Reliability*, 2022.
- [15] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng, “Detecting duplicate bug reports with convolutional neural networks,” in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 416–425, December 2018.
- [16] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, “Towards accurate duplicate bug retrieval using deep learning techniques,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 115–124, 2017.
- [17] D. Swapna and K. Thammi Reddy, “A study of information retrieval approaches in duplicate bug detection,” *Indian J. Sci. Technol.*, vol. 9, no. 43, 2016.

-
- [18] J. Lerch and M. Mezini, “Finding duplicates of your yet unwritten bug report,” in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013.
- [19] J. Zou, L. Xu, M. Yang, M. Yan, D. Yang, and X. Zhang, “Duplication detection for software bug reports based on topic model,” in *2016 9th International Conference on Service Science (ICSS)*, October 2016.
- [20] G. Canfora and L. Cerulo, “How software repositories can help in resolving a new change request,” p. 99, September 2005.
- [21] S. Gupta and S. K. Gupta, “A systematic study of duplicate bug report detection,” *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 1, 2021.
- [22] S. Ibrihich, A. Oussous, O. Ibrihich, and M. Esghir, “A review on recent research in information retrieval,” *Procedia Computer Science*, vol. 201, pp. 777–782, 2022. The 13th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 5th International Conference on Emerging Data and Industry 4.0 (EDI40).
- [23] N. Lal, S. Qamar, and S. Shiwani, “Information retrieval system and challenges with dataspace,” *International Journal of Computer Applications*, vol. 147, no. 8, 2016.
- [24] G. Kowalski, *Information Retrieval Architecture and Algorithms*. 01 2011.
- [25] S. Landolt, T. Wambsganß, and M. Söllner, “A taxonomy for deep learning in natural language processing,” 01 2021.
- [26] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, “A discriminative model approach for accurate duplicate bug report retrieval,” in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, vol. 1, pp. 45–54, May 2010.
- [27] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, “Towards more accurate retrieval of duplicate bug reports,” in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 253–262, 2011.

-
- [28] A. Hindle and C. Onuczko, “Preventing duplicate bug reports by continuously querying bug reports,” *Empirical Software Engineering*, vol. 24, pp. 902–936, 2019.
- [29] B. Soleimani Neysiani and S. M. Babamir, “Improving performance of automatic duplicate bug reports detection using longest common sequence,” in *IEEE 5th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, 2019.
- [30] B. Soleimani Neysiani, S. M. Babamir, and M. Aritsugi, “Efficient feature extraction model for validation performance improvement of duplicate bug report detection in software bug triage systems,” *Information and Software Technology*, vol. 126, p. 106344, Oct 2020.
- [31] B. S. Neysiani and S. M. Babamir, “New methodology for contextual features usage in duplicate bug reports detection: Dimension expansion based on manhattan distance similarity of topics,” in *2019 5th International Conference on Web Research (ICWR)*, Apr. 2019.
- [32] T. M. Rocha and A. L. D. C. Carvalho, “Siameseqat: A semantic context-based duplicate bug report detection using replicated cluster information,” *IEEE Access*, vol. 9, pp. 44610–44630, 2021.
- [33] S. Banerjee, Z. Syed, J. Helmick, M. Culp, K. Ryan, and B. Cukic, “Automated triaging of very large bug repositories,” *Information and Software Technology*, vol. 89, pp. 1–13, 2017.
- [34] S. Banerjee, B. Cukic, and D. Adjero, “Automated duplicate bug report classification using subsequence matching,” in *IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE)*, pp. 74–81, IEEE, 2012.
- [35] B. S. Neysiani and S. M. Babamir, “Automatic duplicate bug report detection using information retrieval-based versus machine learning-based approaches,” in *2020 6th International Conference on Web Research (ICWR)*, pp. 288–293, 2020.

- [36] N. Ebrahimi, A. Trabelsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, “An hmm-based approach for automatic detection and classification of duplicate bug reports,” *Information and Software Technology*, vol. 113, pp. 98–109, Sep. 2019.
- [37] Y. Jiang, X. Su, C. Treude, C. Shang, and T. Wang, “Does deep learning improve the performance of duplicate bug report detection? an empirical study,” *Journal of Systems and Software*, vol. 198, p. 111607, 2023.
- [38] A. Sogaard, Ž. Agić, H. Martínez Alonso, B. Plank, B. Bohnet, and A. Johannsen, “Inverted indexing for cross-lingual NLP,” in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1713–1722, July 2015.
- [39] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Proc. 29th Int. Conf. Softw. Eng.*, pp. 499–510, May 2007.
- [40] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in *Proc. 30th Int. Conf. Softw. Eng.*, pp. 461–470, 2008.
- [41] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking systems,” in *Proc. IEEE Int. Conf. Dependable Syst. Netw. With FTCS DCC (DSN)*, pp. 52–61, 2008.
- [42] C. Kamphuis, A. P. de Vries, L. Boytsov, and J. Lin, “Which bm25 do you mean? a large-scale reproducibility study of scoring variants,” in *Advances in Information Retrieval*, pp. 28–34, 2020.
- [43] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner, and E. Stroulia, “Detecting duplicate bug reports with software engineering domain knowledge,” in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, pp. 211–220, March 2015.

-
- [44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.
- [45] X. Wu, W. Shan, W. Zheng, Z. Chen, T. Ren, and X. Sun, “An intelligent duplicate bug report detection method based on technical term extraction,” in *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 1–12, 2023.
- [46] T. Zhang, I. C. Irsan, F. Thung, and D. Lo, “Cupid: Leveraging chatgpt for more accurate duplicate bug report detection,” 2023.
- [47] R. Chauhan, S. Sharma, and A. Goyal, “Denature: Duplicate detection and type identification in open source bug repositories,” *International Journal of Systems Assurance Engineering and Management*, vol. 14, no. Supplement 1, pp. 275–292, 2023.
- [48] T. Akilan, D. Shah, N. Patel, and R. Mehta, “Fast detection of duplicate bug reports using lda-based topic modeling and classification,” in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1622–1629, 2020.
- [49] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava, “Dwen: Deep word embedding network for duplicate bug report detection in software repositories,” in *Proc. 40th Int. Conf. Softw. Eng., Companion*, pp. 193–194, May 2018.
- [50] H. Mahfoodh, “Identifying duplicate bug records using word2vec prediction with software risk analysis,” *International Journal of Computing and Digital Systems*, vol. 11, pp. 763–773, 2022.
- [51] T. Zhang, D. Han, V. Vinayakarao, I. C. Irsan, B. Xu, F. Thung, D. Lo, and L. Jiang, “Duplicate bug report detection: How far are we?,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, may 2023.
- [52] A. Lazar, S. Ritchey, and B. Sharif, “Generating duplicate bug datasets,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 392–395, 2014.

- [53] E. Loper and S. Bird, “NLTK: The natural language toolkit,” *CoRR*, vol. 28, no. 3, pp. 228–236, 2002.

Turnitin Originality Report

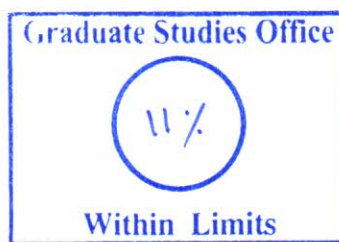
Duplicate Bug Report Detection Using a Hybrid Model

by Nabiya Fatima



From CUST Library (MS Thesis)

- Processed on 15-Sep-2023 09:21 PKT
- ID: 2166633387
- Word Count: 12250



Similarity Index
11%
Similarity by Source

Internet Sources:
5%
Publications:
9%
Student Papers:
2%

sources:

- 1 2% match (Thiago M. Rocha, Andre Carvalho. "SiameseQAT: A Semantic Context-Based Duplicate Bug Report Detection using Replicated Cluster Information", IEEE Access, 2021)
[Thiago M. Rocha, Andre Carvalho. "SiameseQAT: A Semantic Context-Based Duplicate Bug Report Detection using Replicated Cluster Information", IEEE Access, 2021](#)
- 2 1% match (student papers from 10-Oct-2011)
Submitted to University of Warwick on 2011-10-10
- 3 1% match (Jianjun He, Ling Xu, Meng Yan, Xin Xia, Yan Lei. "Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks", Proceedings of the 28th International Conference on Program Comprehension, 2020)
[Jianjun He, Ling Xu, Meng Yan, Xin Xia, Yan Lei. "Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks", Proceedings of the 28th International Conference on Program Comprehension, 2020](#)
- 4 1% match (Yuan Jiang, Xiaohong Su, Christoph Treude, Chao Shang, Tiantian Wang. "Does Deep Learning improve the performance of duplicate bug report detection? An empirical study", Journal of Systems and Software, 2023)
[Yuan Jiang, Xiaohong Su, Christoph Treude, Chao Shang, Tiantian Wang. "Does Deep Learning improve the performance of duplicate bug report detection? An empirical study", Journal of Systems and Software, 2023](#)
- 5 1% match (Internet from 24-Sep-2022)
<http://www.mysmu.edu/faculty/davidlo/papers/ase11-duplicate.pdf>
- 6 < 1% match ()
[SUN, Chengnian, LO, David, WANG, Xiaoyin, KHOO, Siau-Cheng. "A discriminative model approach for accurate duplicate bug report retrieval", Association for Computing Machinery \(ACM\), 2010](#)
- 7 < 1% match (Hussain Mahfooth, Mustafa Hammad. "Identifying Duplicate Bug Records Using Word2Vec Prediction with Software Risk Analysis", International Journal of Computing and Digital Systems, 2022)
[Hussain Mahfooth, Mustafa Hammad. "Identifying Duplicate Bug Records Using Word2Vec Prediction with Software Risk Analysis", International Journal of Computing and Digital Systems, 2022](#)
- 8 < 1% match (Sean Banerjee, Zahid Syed, Jordan Helmick, Mark Culp, Kenneth Ryan, Bojan Cukic. "Automated triaging of very large bug repositories", Information and Software Technology, 2017)
[Sean Banerjee, Zahid Syed, Jordan Helmick, Mark Culp, Kenneth Ryan, Bojan Cukic. "Automated triaging of very large bug repositories", Information and Software Technology, 2017](#)
- 9 < 1% match (Ashima Kukkar, Rajni Mohana, Yugal Kumar, Anand Nayyar, Muhammad Bilal, Kyung S. Kwak. "A Deep Learning Technique Based Duplicate Bug Report Detection and Classification System", IEEE Access, 2020)
[Ashima Kukkar, Rajni Mohana, Yugal Kumar, Anand Nayyar, Muhammad Bilal, Kyung S. Kwak. "A Deep Learning Technique Based Duplicate Bug Report Detection and Classification System". IEEE Access, 2020](#)
- 10 < 1% match (Som Gupta, Sanjai Kumar. "A Systematic Study of Duplicate Bug Report Detection", International Journal of Advanced Computer Science and Applications, 2021)
[Som Gupta, Sanjai Kumar. "A Systematic Study of Duplicate Bug Report Detection". International Journal of Advanced Computer Science and Applications, 2021](#)
- 11 < 1% match (Ruby Chauhan, Shakshi Sharma, Anjali Goyal. "DENATURE: duplicate detection and type identification in open source bug repositories", International Journal of System Assurance Engineering and Management, 2023)
[Ruby Chauhan, Shakshi Sharma, Anjali Goyal. "DENATURE: duplicate detection and type identification in open source bug repositories". International Journal of System Assurance Engineering and Management, 2023](#)
- 12 < 1% match (Thiago Marques Rocha, Andre Luiz Da Costa Carvalho. "SiameseQAT: A Semantic Context-Based Duplicate Bug Report Detection Using Replicated Cluster Information", IEEE Access, 2021)