Evaluating Fine-tuning Approaches for Duplicate Bug Report Detection

Luiz Eduardo Philippi Rosane[*]
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gusphillu@student.gu.se

Robert Einer
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gusphillu@student.gu.se

Mert Yurdakul
Test Scouts
Gothenburg, Sweden
mert.yurdakul@testscouts.com

Francisco Gomes de Oliveira Neto
Department of Computer Science and Engineering
Chalmers University of Technology and the University of Gothenburg
Gothenburg, Sweden
francisco.gomes@cse.gu.se

ABSTRACT

Bug reports are artefacts that document defects encountered by users or developers. Rapid testing and release cycles often lead to the creation of similar or near-duplicate bug reports, introducing redundancy, delaying triage, and increasing maintenance overhead. Although prior research has extensively explored automated methods to detect and manage duplicate bug reports, their natural language nature makes recent advances in large language models (LLMs)—particularly BERT and its successors—a promising avenue for improving robustness and accuracy. In this study, we investigate the use of LLMs to identify duplicate bug reports (DBRs), focusing on the impact of fine-tuning an all-mpnet-base-v2 model, which builds on BERT-based architectures while addressing several of their limitations. We fine-tuned the model using large, open-source bug tracking datasets from the Eclipse, OpenOffice, Firefox, and NetBeans projects. Our evaluation shows that fine-tuning yields only marginal performance improvements across all datasets. We also discuss the trade-offs involved in fine-tuning LLMs for this task, including hyperparameter tuning guidelines and the practical challenges posed by computational and financial cost.

KEYWORDS

Duplicate Bug Reports, Fine-Tuning, Text Similarity

1 Introduction

Identifying faults, typically known as bugs, and their solutions is an important step in the quality assurance of any software development process. When observed through a failure, those bugs are documented as a bug report and often stored in bug tracking tools that help developers document, manage, and resolve bugs efficiently. Bug reports prevent information overload and aid in the resolution of future similar issues [7]. Among those tools, popular ones are Jira and Bugzilla. Other tools such as GitHub and GitLab, which are primarily remote code repositories, also provide bug/issue trackers within their platforms.

However, similar or near-duplicate bug reports often emerge when multiple users report the same issue using slightly different wording. Note that we are *not* necessarily referring to duplicate bug reports (DBR) as identical bug reports that were logged multiple times, rather we refer to them as *very similar reports that address the same or similar faulty behaviour of the software.* The working time of developers can be significantly consumed by the need to analyse (and remove/maintain) duplicate bug reports (DBRs) [8]. Nonetheless, duplicate reports might have details regarding different aspects of failure in their description, including important additional information about the fault (e.g., stack traces, screenshots) and could therefore be helpful during its resolution[6, 18].

Since DBRs can represent significant percentages of all bug reports in some repositories [16], previous studies have evaluated varied promising methods, some of them transferred to tools used in industry [12, 30]. LLMs, particularly BERT models, have then emerged as a promising solution to identify redundant software artefacts due to their ability to identify text similarities [14, 20, 22, 24].

However, initial attempts to use BERT models (all-mpnet-base-v2) to detect DBR have led to limited accuracy and precision [12, 22]. Previous studies argue that the poor performance is attributed to lack of training on bug reports or similar software artefacts. In fact, a few studies have shown improvement [14, 24], but further evidence is needed, particularly in the integration of such approaches to tools used in industry. Note that LLMs require billions of parameters to be trained, with no guarantee of an end product that would generalise well to other repositories[12].

Past studies have shown that fine-tuning can improve the performance of models, but for smaller repositories with bugs written in Japanese (less than 100) [14]. Our study advances the field in two key ways. First, we fine-tune the all-mpnet-base-v2 model using a large-scale dataset of over 700,000 open-source duplicate bug reports and evaluate its performance in a controlled experiment. Second, we integrate the fine-tuned models into an existing industrial tool (Bugle [12]) and provide practical guidelines to address the technical challenges associated with deploying such models in real-world software engineering environments.

Our findings align with prior work showing that fine-tuning yields only modest improvements that may not justify its cost. Our contributions are:

[*]Both authors contributed equally to this research.

- Further evidence of the impact of fine-tuning language models in their ability to detect DBRs.
- Guidelines to support researchers when instrumenting experiments to fine-tune LLMs.
- Extension of a tool used by our industry partner that uses fine-tuned all-mpnet-base-v2 to detect DBR.
- Our dataset and analysis scripts can be used to support future research, particularly as new models and techniques are proposed in the field.

This work advances the state of the art in duplicate bug report detection by providing empirical evidence on the limited benefits of fine-tuning transformer models relative to their cost, reinforcing findings from prior studies. Additionally, by extending an industrial tool and sharing practical guidelines, analysis scripts, and dataset structure, we offer actionable support for researchers and practitioners navigating the technical and experimental challenges of applying language models in this domain.

2 Background and Related Work

A central topic to understand our study is a transformer model. A transformer is a neural network architecture for language models to understand the semantics and relations between words in a text [28]. The use of such models could be useful in identifying duplicate bug reports in scenarios in which they do not necessarily contain many words in common but are *semantically* similar.

Encoder-only transformer models are more suitable for text similarity tasks because they generate *text embeddings*, i.e., numerical representations that capture the contextual meaning of the elements in a given text dataset. Examples of encoder-only transformer models are BERT [11], Sentence-BERT [23] and MPNet [26]. We can then compare how pairs of bug reports are similar by comparing their embeddings using, e.g., cosine similarity. Below, we list studies that have evaluated information retrieval methods, neural network models that did not use transformer models, and research that uses encoder-only transformers to detect DBRs.

2.1 Detecting Duplicate Bug Reports

Simpler approaches to identifying similar bug reports often rely on text embeddings based on word frequency, with cosine similarity used to retrieve matching documents, a common method in information retrieval. Sun et al.[27], for example, proposed REP, a retrieval function built on BM25F—a ranking function used by search engines to estimate the relevance of documents. Their evaluation on bug reports from OpenOffice, Firefox, and Eclipse revealed improved DBR detection (up to 70% accuracy) but high variance and low precision (47%). Due to the stochastic nature of their approach, they used metrics such as Recall Rate@k (RR@K) and Mean Average Precision (MAP). Since LLMs are also stochastic, we use their metrics in our evaluation.

Other researchers applied neural networks to overcome the limitation of syntactic similarity in simpler token-based approaches. For example, Rodrigues et al. [25] introduce the Soft Alignment Model for Bug Deduplication (SABD), a deep learning-based model for DBR detection. A number of other duplicate bug report detection tools are used to compare the results of SABD, including REP and BM25F [27] and Siamese-Pair [10]. Out of all the tools compared in terms of RR@K, SABD performed best on all four examined datasets (Eclipse, OpenOffice, NetBeans and Firefox). However, deep neural networks are costly to run, hence limiting their integration with tools in industry.

BERT models can help overcome some of those costs limitations. Raj and Shetty[22] propose the detection of DBRs by utilising a model based on neural networks, where a version of the RoBERTa model [20] is used. The specific version of RoBERTa that is used is RoBERTa-Large-mnli which was trained on bug reports from the Eclipse and Thunderbird projects. The tool scores high in Precision (0.84), Recall (0.65) and F1-score (0.73). However, their approach targets only the title and description of the bug reports, and disregards other fields that contain relevant debugging information (e.g., steps to reproduce the bug, or comments from the maintainers).

Other studies use BERT for DBR detection, also using large open source datasets for training and testing the model. For example, Patil et al.[21] compares BERT and other models, including a transformer-based one (ADA by OpenAI), and BERT outperformed all models. Kim and Yang[15] combined topic modelling, using the frequency with which a word appears in a report to assign a topic to it, and fine-tuned BERT to identify DBRs in large open source repositories as well, which led to over 87% accuracy in four of the datasets.

Rocha and Carvalho[24] used BERT to generate encodings of the duplicate reports, which are then used to feed a siamese pair of networks, usually used to compare similar inputs, to then give the likelihood of two reports being a pair of duplicates, and it resulted in a Recall Rate@10 around 80% for multiple datasets. As with many other studies, Rocha and Carvalho used open source datasets from projects such as OpenOffice and Eclipse. We re-used the bug reports data they made available in their repository [4].

Isotani et al.[14] have developed a tool for detecting DBRs based on sentence-BERT[23]. Sentence-BERT makes use of a siamese BERT network to efficiently generate embeddings for whole sentences. With such capacity, it is possible to calculate cosine similarity between different texts with less computational resources[23]. Their tool has been trained on a large piece of text documents and later fine-tuned with bug report documents. Their tool scored a value of 0.82 Mean Average Precision compared to the baseline systems, which scored 0.75.

Despite promising results in prior studies, the literature often overlooks the perspective of testers who rely on bug reports. Götharsson et al. developed Bugle, a tool based on the all-mpnet-base-v2 model from SentenceTransformers, and evaluated it in collaboration with an industry partner [12]. While Bugle achieved 94.44% accuracy in retrieving duplicates, testers disagreed with its recommendations 38.9% of the time, often due to missing information in the reports. When applied to open-source datasets from Firefox and Eclipse, the model retrieved only 13% and 35% of correct duplicates, respectively (significantly lower than earlier reported results).

2.2 Research gap

As highlighted in the previous section, most of the research uses some form of machine learning approach to detect DBRs, such as RoBERTa or SBERT [14, 15, 21, 22, 24]. These models are mainly trained on masked language modeling (MLM), where a portion of the words from the training data are replaced with a placeholder symbol. While the model then learns to predict the word that the mask is hiding, it might not capture the full context of a text [29], which might be a hindrance for bug reports as they rely on triggering sequences to reproduce failures. Additionally, because bug reports are often composed of short, task-specific descriptions, the lack of full contextual awareness may lead to reduced model accuracy. This limitation makes it harder for the model to detect semantically similar reports that differ in surface-level wording.

We use the MPNet models, such as the all-mpnet-base-v2, to mitigate the MLM limitations via a permuted pretraining approach with masked tokens [26]. This architecture allows the model to better capture dependencies across all tokens while preserving positional information, which can be particularly helpful for identifying technical problem descriptions. Previous studies have tried to use all-mpnet-base-v2 in Bugle [12], but the model was not fine-tuned with bug report datasets. Therefore, our unique contributions are the extension of Bugle and an evaluation of fine-tuning MPNet models to enhance duplicate bug report detection. By focusing on this specific model and training context, we aim to build on existing work while addressing key shortcomings in current approaches.

3 Research Methodology

In this study, we investigate the impact of fine-tuned MPNet models to identify duplicate bug reports. Our research questions are:

RQ1: How does fine-tuning a transformer affect its effectiveness in detecting duplicate bug reports?

For RQ1, we fine-tune the model with a subset of public bug report datasets with different combinations of hyperparameters. Then, we measure the effectiveness of both the base model and its fine-tuned version.

RQ2: What are the challenges in fine-tuning and using a transformer model to detect duplicate bug reports?

This is a qualitative assessment based on the technical challenges that arise when fine-tuning these models, such as configuring the model's hyperparameters, different costs related to time, computational power and financial aspects. RQ2 aims to support future researchers and practitioners in implementing their own tools with similar language models by understanding the obstacles that they have to overcome when working with such technology.

3.1 Independent and Dependent Variables

To answer our research questions, we conduct a one-factor experiment where the factor is the MPNet model configuration. This factor has two levels: the pre-trained (non-fine-tuned) model and the fine-tuned model. To fine-tune and compare the performance of the models, we use the open source bug report datasets shared by Rocha and Carvalho [24]. The authors shared on Github the datasets with the bugs' descriptions and their duplicates' IDs, together with their analysis code. The reports in all these datasets contain both natural language descriptions of the bugs in English and non-natural language parts as well, such as stack traces and code snippets.

Each bug report contains several columns that hold information about it, such as whether the bug has been resolved, the date it was created, and bug descriptions. Since the all-mpnet-base-v2 model has been trained to find textual similarities between texts, we have chosen to use the short and long description to compare the different bug reports to one another by creating embeddings of the two columns, which can be compared with the cosine similarity.

To evaluate the model's performance, we aim to reflect a realistic scenario in which a software tester is primarily interested in the top suggestions provided by the tool. Retrieving all potential duplicate bug reports (DBRs) would be overwhelming and impractical for manual analysis. Therefore, we focus on the top-k ($k = 5$) bug reports most likely to be duplicates within the repository. Thus, our dependent variables are metrics also used in the literature: Recall Rate@K and the number of detected bug reports (True Positives).

Recall Rate at K (RR@K) measures the proportion of known duplicate bug reports retrieved within the top K results. It is defined as the number of true duplicate bug reports found in the top-K results (D_{found}) divided by the total number of actual duplicates for a given query bug report (D_{total}).

$$\text{RR@K} = \frac{D_{\text{found}}}{D_{\text{total}}}$$

RR@K evaluates how well the tool retrieves all relevant duplicates. Initially, we considered measuring also Precision at K (P@K) which reflects the proportion of DBR among the top-k ones. However, since we choose $k = 5$, we decided to ignore P@K because most bug reports have, on average, between 2–3 duplicates. Therefore, precision would rarely be higher than 40% (2 out of 5 correct predictions).

We also compare the number of true positives found, i.e., how many duplicates the model correctly identified in the top-5 suggestions after querying all the reports that have duplicates. We label the bug reports according to the following:

- **True Positive (TP)** is a duplicate that is found within the top-5 results, and that is linked to the bug report in question. Similarly, **True Negatives (TN)** are bug reports that are not duplicated and have not been included in the top-k suggestion.
- **False Negative (FN)** is a duplicate that is not found within the top-five results, but is linked to the bug report in question. Similarly, a **False Positive (FP)** is a bug report in the top-k recommendations that is *not* a duplicate.

Our hypothesis is that the fine-tuned model can better represent the contextual meaning of the bug reports, and therefore, show improved RR@K. To perform statistical analysis, the null hypothesis of our study is that after performing further training for all-mpnet-base-v2, there will not be any statistically significant difference (SSD) in performance between all models, both fine-tuned and not fine-tuned across the different datasets.

3.2 Fine-tuning Steps

We aim to compare different combinations for the hyperparameters. We split the datasets into: (i) train, (ii) validation, and (iii) test

Table 1: Total size of the datasets, the size of each split and the amount of triplets for each dataset.

Project	Total	Train	Validat.	Test	Triplets
Eclipse	361 006	176 892	75 812	108 302	20 550
NetBeans	216 715	106 190	45 510	65 015	22 374
Firefox	115 814	56 748	24 321	34 745	37 228
OpenOffice	98 070	48 054	20 595	29 421	13 996
Total	791 605				94 148

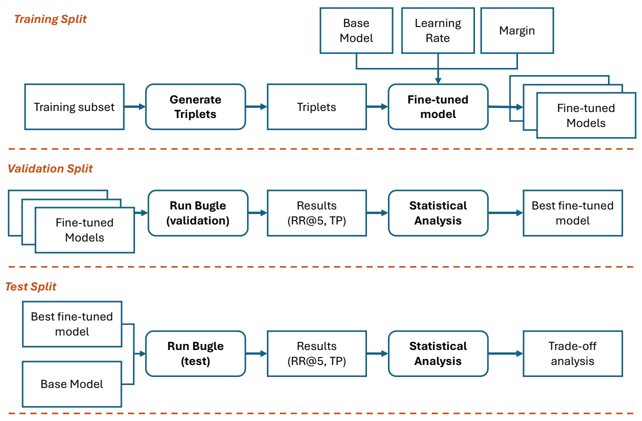


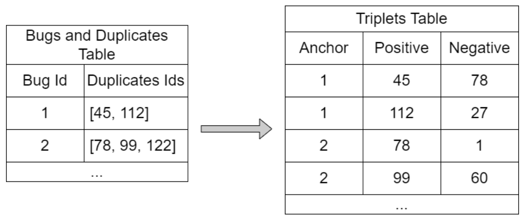
Figure 1: The experiment process, from creating the dataset splits, to finding the best version of the model.

splits (Table 1). The validation split allows one to identify the best combinations of hyperparameters as it is unseen data. However, there is a possibility that one of the models performs better due to how the validation split is composed.

Performing k-fold cross-validation would be ideal in order to derive the best combinations of these values before testing our hypotheses. However, in the context of LLMs, this method would be too expensive and would exceed our resources. Therefore, we choose to have a third split for testing, to mitigate the risks of overfitting to the training or validation data. A visual demonstration with these steps, which we followed during the experiment, can be seen in Figure 1.

Note that the model is pre-trained, we refer to "training" as the first step in our fine-tuning process. When fine-tuning, we used a subset composed of all of the different projects. Then, we evaluated the performance of the models (test split) on each dataset independently. This way, we catered for a scenario where a model has been fine-tuned on varied bug report data, and it is applied to bug reports from a specific project (e.g., two teams within the same company).

Fine-tuning the model requires selecting a loss function, which measures how well the model is performing and guides the adjustment of its internal weights. The SentenceTransformers framework [3] provides several loss functions to choose from. Since our dataset is unlabelled (i.e., the duplicate bug reports do not come with predefined similarity scores), we structured the data to enable



Bugs and Duplicates Table

Bug Id	Duplicates Ids
1	[45, 112]
2	[78, 99, 122]
...	

Triplets Table

Anchor	Positive	Negative
1	45	78
1	112	27
2	78	1
2	99	60
...		

Figure 2: Example of how we formed triplets to be used in the fine-tuning process.

learning. We used a *triplet* format consisting of (i) an anchor bug report, (ii) a positive example (a known duplicate), and (iii) a negative example (an unrelated report).

The TripletLoss function tries to minimise the distance in the vector space between the anchor and the positive examples, and maximise the distance between the anchor and negative examples. In the context of our research, it aims to increase the similarity score between duplicates and decreases it between non-duplicates. As implemented in Isotani et al. [14], we use the DBRs to set the anchors and positive examples, and use a random non-DBR as a negative example. This is done by automatically sampling bug reports from our dataset that acts as ground-truth (see Figure 2). Due to computational constraints, we were unable to apply more sophisticated strategies for selecting negative examples, such as semantic filtering or hard-negative mining.

3.3 Setting Hyperparameters

There are many hyperparameters to set in our experiment, such as the number of epochs and the batch size. During fine-tuning, one epoch has passed when the model has iterated through the whole training dataset once. Since the study that introduced BERT [11] recommended three to five epochs when fine-tuning the model, we decided to keep three epochs. Batch size refers to the number of training examples from the training dataset that the model iterates over before updating its internal weights during an epoch. Larger batch sizes provide more stable weight updates, on the grounds that more triplets are taken into consideration. Unfortunately, our constraints of time and computational resources only allowed us to use a max batch size of 16.

The TripletLoss function in the SentenceTransformers framework includes a key hyperparameter known as the triplet margin, which also requires tuning. This margin defines the minimum distance that a negative example must have from the anchor [3]. We tested two values for this parameter: **1**, which has been used in previous studies [14, 23, 24], and **5**, to explore whether a larger margin could improve the model's performance.

Another important hyperparameter in fine-tuning is the learning rate (LR), which controls how much the model's internal weights are updated after each batch. A high LR may cause overly large weight changes, possibly leading the model away from optimal performance. In contrast, a low LR may update the weights too slowly, requiring many iterations to converge. Since the ideal learning rate depends on factors like model complexity, sample size, and data quality, it must be determined experimentally. We tested two values: $\mathbf{2 \cdot 10^{-7}}$ and $\mathbf{1 \cdot 10^{-8}}$. Although the default learning rate

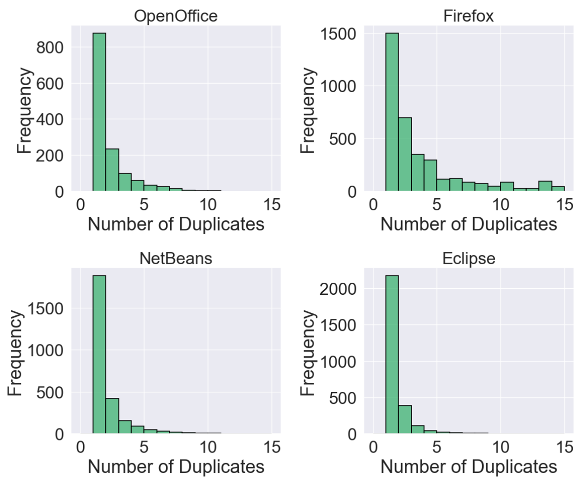


Figure 3: Number of duplicates for reports that have at least one duplicate in the validation split.

in the SentenceTransformers framework is $2 \cdot 10^{-5}$, we opted for smaller values to avoid large updates that might harm the model's performance.[1] All other hyperparameters were kept at their default values as defined in SentenceTransformers version 2.7.0.

3.4 Data Analysis

For our first research question, we assess the difference in performance between the measured values for RR@5, P@5, and the true positives found when using the base model compared to using the fine-tuned version. We use statistical analysis to identify statistically significant differences between our treatments.

Note that several characteristics of our datasets can affect the distribution of the results and, consequently, the conclusions drawn from statistical tests. As shown in Figures 3, 4, and 5, most bug reports with duplicates have only three to five associated duplicates, depending on the dataset, when considering the validation and test splits. For example, if a report has only three duplicates and the model finds all of them, the RR@5 will be 1.0 (i.e., 100% recall). However, if just one is missed, the score drops to 0.66. This variation becomes even more pronounced when a report has only one or two duplicates, resulting in a high standard deviation in the average RR@5. The same reasoning applies to the P@5 metric.

Because of this high variance, which we confirmed when collecting the results (see Section 4), the metric distributions are not normal. As a result, a non-parametric test is more appropriate, as it does not assume a specific data distribution. Furthermore, since we are evaluating the same bug reports across different model configurations, a paired test is required, meaning the compared groups are not independent.

We also used statistical analysis to choose the best combinations of hyperparameters and mitigate the risk of picking a model that had a better performance in relation to the base model due to the inherent randomness in the data split.

[1]https://www.sbert.net/docs/sentence_transformer/training_overview.html

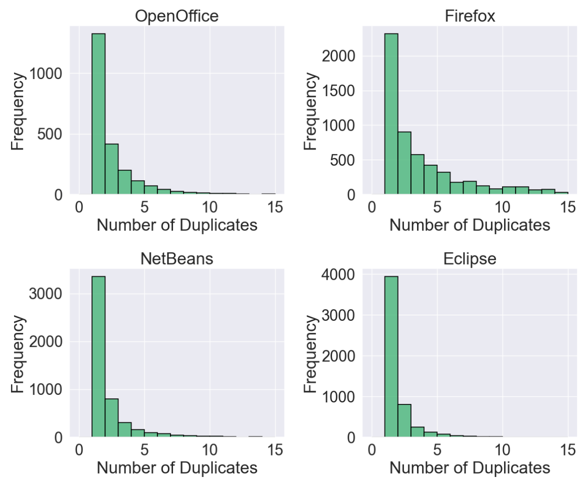


Figure 4: Number of duplicates for reports that have at least one duplicate in the test split.

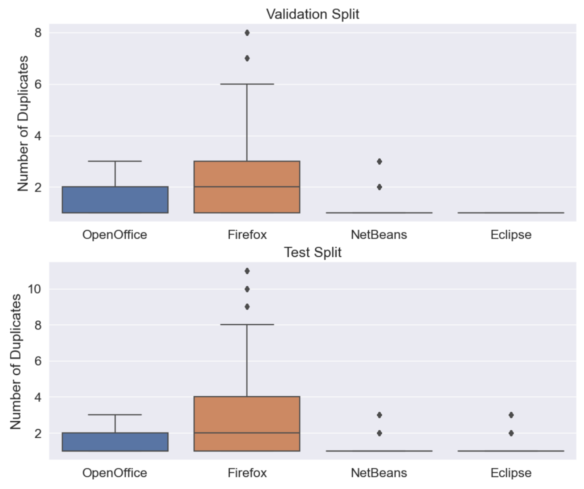


Figure 5: Distribution of duplicates for reports that have at least one duplicate in the test and validation split. We filtered outliers by having the upper-bound set to: third quartile + $1.5\cdot$ **interquartile range.**

Our statistical analysis is conducted in three steps for each dataset. First, we use the Friedman test to determine whether there are statistically significant differences among the five models: the base model and four fine-tuned versions with different learning rates (LR) and triplet margin values.

Next, we perform a post-hoc pairwise comparison using the Wilcoxon signed-rank test, applying a Bonferroni correction to adjust the p-values. This allows us to identify which model pairs differ significantly, using a significance level of $\alpha = 0.05$. Finally, we calculate the Vargha-Delaney $\hat{A}_{12}$ (VDA) statistic to measure the effect size of those differences. VDA provides a probability score indicating how likely one model is to outperform another. A value of 0.5 means both models perform equally (i.e., a small effect), while values above or below 0.5 indicate a stronger likelihood that one model consistently performs better than the other [9].

After selecting the best-performing fine-tuned model based on validation results, we evaluated its performance on the test splits and compared it to the base (non-fine-tuned) model. To assess whether the differences in performance were statistically significant, we used the same statistical procedure described above (skipping the Friedman test, as we only have two levels during the test split).

4 Results

Next, we will present the results from our experiments about the impact of fine-tuning on DBR detection (RQ1). Then, we share some of the main challenges faced when implementing Bugle's extension as well as running our experiments.

4.1 RQ1: Fine-Tuning and DBR Detection

After fine-tuning the model with over 94,000 triplets, we measured the models' performance on the validation split (Table 2). All results for RR@5 were relatively low (below 50%), particularly for all versions of the fine-tuned model. However, remember that most bug reports contain, on average, one or two duplicates, such that this indicates that the models were mainly finding one of the DBR. Note also that the results show high standard deviation which, due to the low number of duplicates per report, means that sometimes the model found both DBR.

Table 2: all-mpnet-base-v2 model average RR@5 on the validation splits.

		RR@5			
LR	Margin	Open Office	Firefox	Eclipse	NetBeans
base model		0.46 ± 0.46	0.44 ± 0.41	0.46 ± 0.48	0.39 ± 0.46
$2 \cdot 10^{-7}$	1	0.38 ± 0.45	0.32 ± 0.39	0.33 ± 0.45	0.30 ± 0.43
$2 \cdot 10^{-7}$	5	0.30 ± 0.42	0.23 ± 0.35	0.24 ± 0.41	0.22 ± 0.39
$1 \cdot 10^{-8}$	1	**0.48** ± 0.46	**0.45** ± 0.42	**0.47** ± 0.48	**0.42** ± 0.46
$1 \cdot 10^{-8}$	5	**0.48** ± 0.46	**0.45** ± 0.42	**0.47** ± 0.48	**0.42** ± 0.46

Therefore, we focus on the relative differences of RR@5 between the base model and all fine-tuned versions. Note that improvements in RR@5 were marginal when fine-tuning the model at a learning rate of 10^{-8} for all projects (between 1–2% higher). However, the higher learning rate ($2 \times 10^{-}7$) led to worse performance (between 8%–20% worse depending on the project). A higher margin (5) seemed to reduce the RR@5 when using a learning rate $LR = 2 \times 10^{-7}$, but did not seem to affect RR@5 for $LR = 10^{-8}$.

We also saw differences when we observed the number of true positives each model found (Table 3). Similarly to RR@5, we focus on the relative differences between the model and the fine-tuned versions. Similar to our results in RR@5, $LR = 10^{-8}$ shows marginal improvement from the base model by increasing the number of TP by circa 100 throughout all different projects. All results for $LR = 2 \times 10^{-7}$ are lower than the base model, with worse performance when using a margin given.

As we can see in Table 4, there is a statistically significant difference between the models across all datasets for RR@5. When doing a post-hoc pairwise analysis with the Wilcoxon test (Table 5), we can see that the drop in performance when fine-tuning with LR set to $2 \cdot 10^{-7}$ was indeed significant, suggesting that the models overfitted to the data used for further training.

Table 3: Distribution of true positives found by the different models on the validation split for each dataset.

LR	Margin	Open Office TP / P	Firefox TP / P	Eclipse TP / P	NetBeans TP / P
base model		978 / 2721	4264 / 13 950	1623 / 4446	1539 / 3913
$2 \cdot 10^{-7}$	1	862 / 2721	3198 / 13 950	1169 / 4446	1197 / 3913
$2 \cdot 10^{-7}$	5	697 / 2721	2375 / 13 950	830 / 4446	872 / 3913
$1 \cdot 10^{-8}$	1	**1020 / 2721**	**4374 / 13 950**	**1661 / 4446**	1607 / 3913
$1 \cdot 10^{-8}$	5	**1020 / 2721**	4373 / 13 950	**1661 / 4446**	**1608 / 3913**

TP (True Positives)
P (Positives: True Positives + False Negatives)

Table 4: Statistical analysis of the RR@5 results using the Friedman test on the validation splits of all datasets.

Friedman test		
Dataset	χ^2	p-value
Open Office	436.05	$<2.2 \times 10^{-16}$
Firefox	2892.8	$<2.2 \times 10^{-16}$
Netbeans	1298	$<2.2 \times 10^{-16}$
Eclipse	1620.2	$<2.2 \times 10^{-16}$

Moreover, the lower learning rate, $1 \cdot 10^{-8}$, led to improved performance, with statistically significant differences observed for all projects except Eclipse ($p = 0.083$). However, the Vargha-Delaney $\hat{A}_{12}$ measure indicated that these differences were small or negligible in all cases. Due to space constraints, we report only the RR@5 results here, but the same pattern was observed in the number of true positives (TP), which is available in our replication package.

Key finding (RQ1): Fine-tuning with a LR of $2 \cdot 10^{-7}$ decreased detection performance significantly. In contrast, fine-tuning with LR of $1 \cdot 10^{-8}$ showed only *marginal improvements* in RR@5 across all datasets. This shows little benefit from fine-tuning the model.

Based on the statistical analysis, we move on to the test split by comparing only the **base model** and the fine-tuned model with $LR = 10^{-8}$ **and margin 1**, henceforth referred as the fine-tuned model. Table 6 shows the results for RR@5 and TP for the four projects.

Note that for both RR@5 and TP, the values are higher in the fine-tuned model. However, our statistical analysis a negligible increase. We see consistent results to what was measured during the validation split such as a high standard deviation (Table 7).

Key finding (RQ1): Despite the negligible improvement in performance, different combinations of parameters affect the model's performance. In our case, the LR was the most impactful.

Table 5: Statistical analysis of the RR@5 results on all projects during the validation step. The table include the results from a Wilcoxon test and Vargha-Delaney $\hat{A}_{12}$ measure.

	Eclipse			Firefox			Netbeans			OpenOffice		
Model	p-value	VDA	ES	p-value	VDA	ES	p-value	VDA	E	p-value	VDA	ES
base model - LR.1e-8.margin.1	8.32×10^{-2}	0.49	N	2.82×10^{-6}	0.49	N	5.48×10^{-7}	0.49	N	1.05×10^{-2}	0.49	N
base model - LR.1e-8.margin.5	8.32×10^{-2}	0.49	N	3.41×10^{-6}	0.49	N	1.66×10^{-7}	0.49	N	1.05×10^{-2}	0.49	N
base model - LR.2e-7.margin.1	5.91×10^{-43}	0.57	S	7.33×10^{-83}	0.54	N	7.91×10^{-28}	0.55	N	4.98×10^{-12}	0.55	N
base model - LR.2e-7.margin.5	4.33×10^{-103}	0.62	S	1.14×10^{-185}	0.58	S	1.21×10^{-76}	0.60	S	5.27×10^{-33}	0.59	S
LR.1e-8.margin.1 - LR.1e-8.margin.5	N/A	0.50	N	1.00	0.50	N	1.00	0.50	N	N/A	0.50	N
LR.2e-7.margin.1 - LR.1e-8.margin.1	4.45×10^{-54}	0.57	S	3.77×10^{-108}	0.55	N	7.65×10^{-48}	0.56	S	2.16×10^{-19}	0.56	S
LR.2e-7.margin.1 - LR.1e-8.margin.5	4.45×10^{-54}	0.57	S	5.16×10^{-108}	0.59	S	4.14×10^{-48}	0.56	S	2.16×10^{-19}	0.56	S
LR.2e-7.margin.5 - LR.1e-8.margin.1	1.98×10^{-115}	0.63	S	3.61×10^{-206}	0.55	N	2.17×10^{-95}	0.61	S	2.87×10^{-42}	0.60	S
LR.2e-7.margin.5 - LR.1e-8.margin.5	1.98×10^{-115}	0.63	S	5.45×10^{-206}	0.59	S	1.06×10^{-95}	0.61	S	2.87×10^{-42}	0.60	S
LR.2e-7.margin.5 - LR.2e-7.margin.1	7.66×10^{-52}	0.55	N	4.56×10^{-96}	0.54	N	2.52×10^{-41}	0.55	N	3.22×10^{-21}	0.55	N

ES (Effect size) is defined as: N = Negligible or S = Small

Table 6: all-mpnet-base-v2 model average on RR@5 and TP during the test splits.

	Open Office	Firefox	Eclipse	NetBeans
RR@5:				
base model	0.40 ± 0.44	0.41 ± 0.41	0.42 ± 0.47	0.37 ± 0.45
fine-tuned	0.42 ± 0.45	0.42 ± 0.41	0.43 ± 0.47	0.38 ± 0.45
TP / P:				
base model	1583 / 5052	7040 / 27 123	2984 / 8006	2789 / 8850
fine-tuned	1664 / 5052	7088 / 27 123	3053 / 8006	2886 / 8850

Table 7: Statistical analysis of the RR@5 and TP results on the test splits of all datasets using the Wilcoxon test.

	Models	p-value	SSD	VDA	ES
RR@5:					
Open Office	base - fine-tuned	1.07×10^{-6}	Yes	0.48	N
Firefox	base - fine-tuned	9.45×10^{-4}	Yes	0.49	N
Netbeans	base - fine-tuned	4.01×10^{-7}	Yes	0.49	N
Eclipse	base - fine-tuned	1.09×10^{-2}	Yes	0.49	N
TP:					
Open Office	base - fine-tuned	3.53×10^{-7}	Yes	0.48	N
Firefox	base - fine-tuned	1.53×10^{-1}	No	0.49	N
Netbeans	base - fine-tuned	7.39×10^{-7}	Yes	0.49	N
Eclipse	base - fine-tuned	7.32×10^{-4}	Yes	0.49	N

SSD (Statistically Significant Difference)
ES (Effect size) is defined as: N = Negligible or S = Small

4.2 RQ2: Guidelines on Fine-tuning for duplicate bug report detection

Transformer-based models and their fine-tuning present several challenges due to their complexity and the advanced technologies they rely on. Here, we describe key issues encountered.

Limited Documentation and External Support: We used the SentenceTransformers framework to support the implementation of our models. While this framework simplifies many aspects of development, it presents challenges — most notably, limited documentation. For example, while training a model from scratch is documented, the process for fine-tuning an existing model is not clearly explained. To address this, we relied on external resources such as HuggingFace [5]. Additionally, fine-tuning is computationally intensive, and distributing the workload across multiple GPUs could improve efficiency. However, the current version of SentenceTransformers does not support multi-GPU training.

Hyperparameter Tuning Challenges: Finding optimal hyperparameter values was another major challenge, as they can significantly impact model performance. Key parameters to experiment with include batch size, number of epochs, learning rate (LR), and the margin used in the TripletLoss function. There are no universally accepted values for these, and the DBR detection literature often lacks explanations for why specific hyperparameter values are chosen. As a result, tuning becomes a trial-and-error process.

Moreover, hyperparameters do not operate independently such that changing one can affect the impact of another. This interdependence increases the complexity of tuning. For instance, as shown in our validation split, the interaction between margin and learning rate influenced model performance on higher learning rates. Each evaluation of a new configuration requires a full fine-tuning cycle, making the process time-consuming.

Data Quality and Availability Another challenge was finding high-quality, publicly available data. No private or preprocessed datasets were available for this study, so we relied on data from literature and open-source issue trackers. In the latter, anyone can submit a bug report, and while templates may be provided, the quality of submissions is not verified. This introduces inconsistencies and noise that can negatively affect model performance.

During fine-tuning, the model learns by associating similar bug descriptions. However, if reports marked as duplicates differ significantly in content, the model may learn incorrect associations. This can lead to poor-quality embeddings that are less effective at identifying true duplicates. Additionally, human error in the duplicate labeling process can further degrade data quality, introducing noise that undermines training. One way to mitigate this challenge is to apply preprocessing and filtering steps that identify and exclude low-quality or inconsistently labeled reports before fine-tuning.

Computational and Financial Constraints: Using transformer models for duplicate bug report (DBR) detection involves significant computational and financial costs. Although fine-tuning is less expensive than training a model from scratch, it still requires powerful hardware (e.g., GPUs with substantial VRAM) which is not commonly available on personal or budget machines. For example, the maximum batch size during fine-tuning is directly limited by available VRAM, and the SentenceTransformers framework requires that this memory be available on a single GPU. As a result, researchers must either invest in high-end hardware or use cloud services, both of which come with financial implications.

We did not have access to suitable local machines and instead used Amazon Web Services (AWS), specifically the SageMaker service, to run our fine-tuning experiments. SageMaker allows users to select various hardware configurations and charges based on usage time. Because fine-tuning large datasets and exploring different hyperparameter combinations can be time-consuming, balancing computational power with cost was crucial. The iterative nature of hyperparameter tuning further increased usage time and, consequently, expenses. To manage these costs, researchers can prioritize early experimentation with smaller subsets of data to narrow down viable hyperparameter ranges before scaling up to full datasets.

Key fingings (RQ2): Below, we summarize the main challenges we encountered.

- Selecting the optimal values for the hyperparameters is difficult and requires a lot of trial and error due to the lack of standardised values.
- Data quality is important to the process of fine-tuning a transformer-model, but not widely available in large public bug report datasets.
- Fine-tuning is cheaper than training a model from scratch, but still requires financial and computational resources that cannot be disregarded.

5 Discussion

5.1 Model Sensitivity to Learning Rate

The SentenceTransformers framework uses a default learning rate (LR) of $2 \cdot 10^{-5}$, but as discussed in Section 3, we experimented with smaller LRs to prevent overfitting. Surprisingly, even a relatively low LR of $2 \cdot 10^{-7}$ caused the model to overfit, leading to decreased performance. This suggests that the model's weights shifted too much during fine-tuning, causing it to "forget" previously learned representations and fail to generalize to unseen data. This behavior is consistent with Lin et al. [19], who highlight that transformer models — despite their flexibility — are especially prone to overfitting when trained on limited data, even when such datasets contain tens of thousands of samples. Their flexibility, which allows them to work without strict input structures, also makes them more likely to learn patterns specific only to the training data.

Implication 1: The performance drop observed with small learning rates highlights that fine-tuning flexible transformer models like all-mpnet-base-v2 may not yield significant benefits unless data scale and learning rate are carefully balanced.

5.2 Marginal Gains and Future Directions

In contrast, the model fine-tuned with an even lower LR of $1 \cdot 10^{-8}$ successfully identified duplicate reports that the base model and other configurations missed. Below are two examples from the OpenOffice dataset:

Example 1 (after fine-tuning):

"space bar does not work in cells in current spreadsheet"

"In build 680 milestones 26 and 28 (Linux) it is impossible to enter some special symbols into cell: whitespace, underscore, quotation mark. Simple quote starts text but is not displayed in input field."

Example 2 (after fine-tuning):

"Excel chart importing in OO spreadsheet (partly) fails When I import a more complicated excel chart (2 y-axis chart, surface plot) into oo, the chart isn't shown correctly. For the 2 y-axis this is solved by e.g defining the correct axis to use (2nd -right- axis i/o 1st -left-axis and vice-versa) for the data columns manually. A (2D) surface plot translates to a (3D) one. The legend doesn't translate quite OK. This may also be caused by the scaling of the 3D plot. Should I also provide an example?"

"Importing this file: http://cycle.lightship.tv/files/Bicycle_Geometry_101_v311.xls
into OOo-2.0 does not work properly. The frame plot produced by OOo-2.0 (and StarOffice-8.0, for that matter) is not rendered properly:
http://www.os2.dhs.org/~john/Bicycle_Geometry_plot-OOo2.pdf
It does, however, render properly in gnumeric-1.4.3:
http://www.os2.dhs.org/~john/Bicycle_Geometry_plot-gnumeric-1.4.3.pdf
This may be related to issue #15555 but I'm not certain."

Despite these improvements, the gains were marginal, and the sensitivity to learning rate adjustments suggests that the model's original weights are already near a local or global minimum in the loss function. We suspect that the marginal performance gains observed during fine-tuning are due to a combination of factors. These include (i) the strength of the pre-trained embeddings (all-mpnet-base-v2), (ii) the presence of label noise, and (iii) the limited domain-specific signal in the fine-tuning data where each dataset corresponds to a different SUT. While we applied early stopping and used low learning rates to reduce overfitting, the absence of hard negatives and potential semantic drift may have further diluted the fine-tuning signal.

Another contributing factor may be catastrophic forgetting, a phenomenon where a model, during fine-tuning, loses previously acquired general knowledge as it adapts too narrowly to the new data. This could explain why performance improvements remained limited despite model adaptation efforts.

Given the multiple uncertain factors influencing performance mentioned above, the marginal gains of the results raises concerns about whether the effort and complexity of fine-tuning all-mpnet-base-v2 are justified for the duplicate bug report identification task. Other studies, such as those by Shetty et al. [22] and Rocha and Carvalho [24], addressed this by embedding transformer models into more complex architectures. Meanwhile, Isotani et al. [14] also fine-tuned a transformer model (SBERT) but unlike our results, they observed clear improvements, possibly due to their use of smaller, more focused datasets with a higher density of duplicates.

Their approach also calculated similarity by comparing bug report titles and descriptions separately, then using the maximum similarity score between the two. This could help reduce noise from unstructured or irrelevant content in the descriptions, such as stack traces, and is worth exploring in future work.

> **Implication 2:** While fine-tuning with lower learning rates showed some benefit, the limited performance gains suggest that alternative strategies (e.g., refining input structure or adapting similarity calculations) may offer more impactful improvements.

5.3 Transparency Regarding Hyperparameter Choices

Given the importance of selecting appropriate hyperparameter values, we were surprised by the lack of transparency in the DBR detection literature regarding how these values are chosen. The level of detail varies across studies: some mention specific hyperparameter values [10, 15, 25], while others only list the parameters they tuned without sharing which values led to the best performance [22]. The studies we reviewed often omit the rationale or trade-offs behind their hyperparameter choices.

In our experiments, we observed that hyperparameter values significantly affected performance. For example, reducing the learning rate improved the average RR@5 by up to 3%, while increasing it caused a drop of up to 22% compared to the base model. However, these findings emerged through "trial and error", as we initially relied on values from previous studies without knowing if they were appropriate for our context. Greater transparency in hyperparameter decisions could help future researchers make more informed choices, reduce experimentation time, and potentially lower the financial cost of fine-tuning, especially when using cloud resources.Note that Deshmukh et al. [10] and Kim and Yang [15] both suggest that further tuning could improve their models, reinforcing the importance of optimal hyperparameter selection.

> **Implication 3:** Clear reporting and justification of hyperparameter choices would support more efficient and cost-effective experimentation in future DBR detection research.

5.4 Limitations of Using Open Source Datasets

One key challenge with using open-source datasets for duplicate bug report (DBR) detection is the inconsistency in how different users describe the same issue. While semantic differences are expected, in some cases it was difficult to understand why certain reports were marked as duplicates at all. During our experiments, we encountered multiple cases where reports labeled as duplicates had very low cosine similarity scores, indicating minimal textual overlap or shared context. For example, the following pairs were marked as duplicates in the OpenOffice dataset, yet seem unrelated based on their content:

Example 3:

> *"Data should be copied rather than moved during drag and drop to data source. When I drag and drop data from calc to a registered data source the data is moved rather than copied, which means that if anything goes wrong with the copying the data is lost. Actual loss of data can be avoided if the user is careful, but the more fault tolerant approach would be to copy the data in case of user error."*

> *"Bullets*
> *By default OpenOffice give me a strange bullet character. It is the #10 of the Windings font that is a solid black circle with a white '10' inside of it. It is the same character as 0xF095 using Insert - Special Character of the Wingdings font. Also on the Format - Numbering/Bullets dialog, there are no other bullet styles to choose from. Everything else is the empty box character. Thanks, Steve"*

Example 4:

> *"dragging text in outline mode does not work*
> *i created a document using the powerpoint template attached to bug 39098 and entered some hierarchical text on a slide. i then switched to 'Outline' view instead of 'Normal' view, chose a sentence and dragged it to a different point on the slide. The text in the outline view was changed, but the preview in the 'Slides' window to the left of the outline did not update. I then switched to 'Normal' view and back to 'Outline' view, and the effects of the drag were completely lost. The sentence i dragged was right back where it started."*

> *"can't 'find all' in simpress, the 'find all 'button in the 'Find & Replace' dialog is always disabled."*

Even though we used a model trained to understand semantic context, the large dissimilarities in these examples suggest they may have been incorrectly labeled as duplicates in the original dataset. When these low-similarity reports are included in the evaluation, they reduce the model's performance on metrics like RR@K. Since the cosine similarity scores for such pairs are unlikely to rank within the top-5 results, they act as outliers that penalize the model unfairly. This challenge is not unique to our chosen metrics, as other commonly used metrics like recall and precision would also require a similarity threshold, which introduces a similar problem.

One potential solution, as suggested by Götharsson and Stahre [12], is to curate and clean the bug report descriptions before using them for training or evaluation. However, given the scale of open-source repositories, this would be a highly resource-intensive task (e.g., some contain hundreds of thousands of reports). Due to the scope and time constraints of our study, this was not feasible, but it remains an important consideration for future work.

Implication 4: The use of open-source datasets without manual curation introduces noisy and possibly mislabeled duplicates, which can distort evaluation results and limit model performance, thus highlighting the need for better dataset quality control in future research.

6 Threats to validity

We mitigate risks with *internal validity* by ensuring that observed differences in performance were due to the treatment (fine-tuning) and not other variables. Therefore, all models were evaluated on the same set of bug reports. We also used a clear data split to avoid information leakage: the train set was used exclusively for fine-tuning, while validation and test sets remained untouched during training. Additionally, we applied statistical tests to confirm whether performance differences were significant.

There is a risk that fine-tuning may lead to overfitting, where the model performs well on training data but fails to generalize. To address this *conclusion validity threat*, we split the dataset into training (50%), validation (20%), and test (30%) subsets. This allowed us to evaluate model performance on unseen data. Furthermore, we used bug reports from multiple open-source projects to increase the diversity and generalizability of the training input.

To ensure we are accurately measuring the model's ability to detect duplicate bug reports, we used Recall Rate at K (RR@K), a well-established metric in DBR detection research. This choice mitigates risks with **construct validity** since it aligns with previous studies and reflects a realistic use case of presenting the top-ranked results to a tester.

7 Conclusions and Future Work

In this study, we investigated whether fine-tuning the all-mpnet-base-v2 model could substantially improve duplicate bug report (DBR) detection on large, open-source datasets. Our results show that although a lower learning rate ($1 \cdot 10^{-8}$) led to statistically significant improvements over the base model, the effect size was negligible. Conversely, a higher learning rate ($2 \cdot 10^{-7}$) resulted in significant performance decreases, suggesting overfitting. Thus, for the datasets used, we conclude that the resource investment in fine-tuning all-mpnet-base-v2 is not justified by the marginal gains achieved.

The challenges we encountered were multifaceted. Hyperparameter selection required extensive trial and error, which increased computational and financial costs, particularly when using cloud resources. Additionally, the quality and labeling consistency of open-source bug report datasets presented significant obstacles; many reports featured unclear descriptions or were inaccurately marked as duplicates, introducing noise into both model training and evaluation.

Our findings suggest that fine-tuning offers limited added value for duplicate bug reports, which often feature concise and formulaic language. This observation may extend to other software artefacts that similarly aim to reduce linguistic ambiguity, such as test cases or requirements. In contrast, fine-tuning could be more beneficial for artefacts involving complex or nuanced language, where semantic interpretation plays a larger role (e.g., regulatory documents or user feedback). In many software engineering contexts, however, the marginal gains observed may not warrant the additional cost and effort associated with fine-tuning.

Future work can explore whether using smaller, more focused datasets could lead to better performance, such as those with a higher density of duplicate reports like in Isotani et al. [14]. Moreover, future ablation studies can investigate whether different fields of a bug report (e.g., stack traces, steps to reproduce) can impact the fine-tuning. Another promising direction involves fine-tuning only select layers or adding lightweight trainable layers to preserve more pre-trained knowledge, as suggested by Houlsby et al. [13]. This approach could mitigate overfitting while reducing computational costs.

Further investigation is also needed to compare the performance of emerging models. For instance, mxbai-embed-2d-large-v1 [2] provides higher-dimensional embeddings and is open-source and fine-tunable, while OpenAI's text-embedding-3-large [1] offers a powerful proprietary alternative — though it raises privacy concerns due to its API-only usage. Comparing these models in the DBR detection domain would help clarify whether open-source options can match or outperform commercial solutions while keeping data on-premise.

In summary, while fine-tuning state-of-the-art transformer models can lead to small performance improvements, they come with significant costs and limitations. Future work should focus on dataset quality, selective fine-tuning strategies, and the evaluation of emerging embedding models to develop more practical and effective DBR detection tools.

ARTIFACT AVAILABILITY

This research was conducted in collaboration with our industry partner Test Scouts, a consulting firm based in Gothenburg, Sweden, that specializes in software and system testing. As a result, the Bugle tool, the fine-tuned models used, and the outcomes of the experiments cannot be publicly shared due to non-disclosure agreements. However, to promote transparency and support reproducibility, we share data used in our analysis.

Our study's Github repository[2] includes Python code for loading the result CSV files and generating the tables presented in the paper, offering insight into the reliability of our analysis. Due to size restrictions in the repository, the datasets were not included in the repository. However, details of the dataset can be found in their original work [17], whereas the files can be downloaded in BugRepo's Github[3].

REFERENCES

[1] [n. d.]. Embeddings. https://platform.openai.com/docs/guides/embedding platform.openai.com.

[2]https://github.com/fgoneto/study-fine-tuning-analysis
[3]https://github.com/logpai/bughub

[2] [n. d.]. Fresh 2D-Matryoshka Embedding Model. https://www.mixedbread.ai/blog/mxbai-embed-2d-large-v1 mixedbread.ai, accessed 11/05/2024.

[3] [n. d.]. Sentence-Transformers. https://www.sbert.net/index.html# SBERT.net, accessed 20/02/2024.

[4] 2019. thiagomarquesrocha/siameseQAT. GitHub, =https://github.com/thiagomarquesrocha/siameseQAT?tab=readme-ov-file, accessed 09/02/2024.

[5] 2023. transformers. Hugging Face, =https://huggingface.co/docs/transformers/index, accessed 03/02/2024.

[6] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate bug reports considered harmful… really?. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 337–345.

[7] Yguarata Cerqueira Cavalcanti, Eduardo Santana de Almeida, Carlos Eduardo Albuquerque da Cunha, Daniel Lucrédio, and Silvio Romero de Lemos Meira. 2010. An Initial Study on the Bug Report Duplication Problem. In *2010 14th European Conference on Software Maintenance and Reengineering*. 264–267. doi:10.1109/CSMR.2010.52

[8] Yguarata Cerqueira Cavalcanti, Paulo Anselmo da Mota Silveira Neto, Daniel Lucrédio, Tassio Vale, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. 2013. The bug report duplication problem: an exploratory study. *Software Quality Journal* 21 (2013), 39–66.

[9] Francisco Gomes de Oliveira Neto, Richard Torkar, Robert Feldt, Lucas Gren, Carlo A. Furia, and Ziwei Huang. 2019. Evolution of statistical analysis in empirical software engineering research: Current state and steps forward. *Journal of Systems and Software* 156 (2019), 246–267. doi:10.1016/j.jss.2019.07.002

[10] Jayati Deshmukh, KM Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. 2017. Towards accurate duplicate bug retrieval using deep learning techniques. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 115–124.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Malte Götharsson, Karl Stahre, Gregory Gay, and Francisco Gomes de Oliveira Neto. 2024. Exploring the Role of Automation in Duplicate Bug Report Detection: An Industrial Case Study. (2024), 193–203. doi:10.1145/3644032.3644450

[13] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International conference on machine learning*. PMLR, 2790–2799.

[14] Haruna Isotani, Hironori Washizaki, Yoshiaki Fukazawa, Tsutomu Nomoto, Saori Ouji, and Shinobu Saito. 2023. Sentence embedding and fine-tuning to automatically identify duplicate bugs. *Frontiers in Computer Science* 4, 1032452.

[15] Taemin Kim and Geunseok Yang. 2022. Predicting Duplicate in Bug Report Using Topic-Based Duplicate Learning With Fine Tuning-Based BERT Algorithm. *IEEE Access* 10 (2022), 129666–129675. doi:10.1109/ACCESS.2022.3226238

[16] Berfin Kucuk and Eray Tuzun. 2021. Characterizing Duplicate Bugs: An Empirical Analysis. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 661–668. doi:10.1109/SANER50967.2021.00084

[17] Alina Lazar, Sarah Ritchey, and Bonita Sharif. 2014. Generating duplicate bug datasets. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 392–395. doi:10.1145/2597073.2597128

[18] Mingyang Li, Lin Shi, and Qing Wang. 2019. Are all duplicates value-neutral? an empirical analysis of duplicate issue reports. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 272–279.

[19] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2022. A survey of transformers. *AI open* 3 (2022), 111–132.

[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[21] Avinash Patil, Kihwan Han, and Sabyasachi Mukhopadhyay. 2023. A comparative study of text embedding models for semantic text similarity in bug reports. *arXiv preprint arXiv:2308.09193* (2023).

[22] Vijay Raj and Jyoti Shetty. 2023. TicketTrace: Intelligent High Parity Ticket Detection Through Deep Learning Techniques. In *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. 1–6. doi:10.1109/CSITSS60515.2023.10334156

[23] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[24] Thiago Marques Rocha and André Luiz Da Costa Carvalho. 2021. SiameseQAT: A Semantic Context-Based Duplicate Bug Report Detection Using Replicated Cluster Information. *IEEE Access* 9 (2021), 44610–44630. doi:10.1109/ACCESS.2021.3066283

[25] Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. 2020. A soft alignment model for bug deduplication. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 43–53.

[26] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. Mpnet: Masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems* 33 (2020), 16857–16867.

[27] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 253–262.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[29] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[30] Ting Zhang, Donggyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. 2023. Duplicate Bug Report Detection: How Far Are We? *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 97 (may 2023), 32 pages. doi:10.1145/3576042