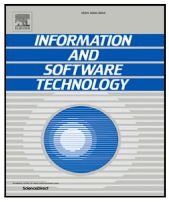
Information and Software Technology 160 (2023) 107217

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



BERT- and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: A comparative study

Luiz Gomes [a,c,*], Ricardo da Silva Torres [b,1], Mario Lúcio Côrtes [c,1]

[a] Department of Software Engineering, Institute of Exact Sciences and Informatics (ICEI), PUC MG, Belo Horizonte, 30535-901, MG, Brazil
[b] Department of ICT and Natural Sciences, NTNU – Norwegian University of Science and Technology, Ålesund, Norway
[c] Department of Information Systems, Institute of Computing (IC), University of Campinas, UNICAMP, Campinas, 3521-7000, SP, Brazil

ARTICLE INFO

Dataset link: A Dataset for Long-lived Bug Prediction in FLOSS (Original data)

Keywords:
Software maintenance
Bug Tracking System
Long-lived bugs
Machine learning
Text mining
Natural Language Processing
BERT

ABSTRACT

Context: The correct prediction of long-lived bugs could help maintenance teams to build their plan and to fix more bugs that often adversely affect software quality and disturb the user experience across versions in Free/Libre Open-Source Software (FLOSS). Machine Learning and Text Mining methods have been applied to solve many real-world prediction problems, including bug report handling.
Objective: Our research aims to compare the accuracy of ML classifiers on long-lived bug prediction in FLOSS using Bidirectional Encoder Representations from Transformers (BERT)- and Term Frequency - Inverse Document Frequency (TF-IDF)-based feature extraction. Besides that, we aim to investigate BERT variants on the same task.
Method: We collected bug reports from six popular FLOSS and used the Machine Learning classifiers to predict long-lived bugs. Furthermore, we compare different feature extractors, based on BERT and TF-IDF methods, in long-lived bug prediction.
Results: We found that long-lived bug prediction using BERT-based feature extraction systematically outperformed the TF-IDF. The SVM and Random Forest outperformed other classifiers in almost all datasets using BERT. Furthermore, smaller BERT architectures show themselves as competitive.
Conclusion: Our results demonstrated a promising avenue to predict long-lived bugs based on BERT contextual embedding features and fine-tuning procedures.

1. Introduction

Today's software maintenance activities in FLOSS and Closed Source Software (CSS) rely mainly on information extracted from bug reports opened in Bug Tracking Systems (BTS). This kind of system plays a key role as a communication and collaboration tool in both environments. However, with many users and developers with different expertise spread out worldwide, FLOSS increased the requirement of using a BTS in the maintenance pipeline.

Users and developers often interact with the maintenance team filling out a brief description, a long description, and a provisional severity level associated with a bug in a bug report form provided by a BTS. Next, a maintenance team member reviews the bug report and either approves or rejects it. If a team member approves the bug report, he or she provides further information, such as indicating its priority and assigning a person in charge of fixing it. Due to the high number of bug reports in medium- and large-size FLOSS projects, the manual handling of bug reports may be entirely subjective, tiresome, and error-prone [1–3]. Therefore, a wrong decision within the bug report lifecycle may strongly affect the planning of maintenance activities.

Allocating resources to fix bugs is another critical activity to plan software maintenance [4]. Estimating the "bug fix time" is essential for many stakeholders [5,6]. For software managers, it is one of the main factors that help them to perform such allocation more effectively [7–9]. Large projects with tight schedules and limited resources may be unable to close known bugs before the release. There are many bugs that can be reported over a long period of time [10,11]. Software managers must decide which bugs to fix in the current release and which to defer to the next version. This could accumulate many unfixed defects [11]. When faced with a lot of bug reports, timely identification of bugs with long fixing time may help managers allocate resources more effectively [8,12].

* Corresponding author at: Department of Software Engineering, Institute of Exact Sciences and Informatics (ICEI), PUC MG, Belo Horizonte, 30535-901, MG, Brazil.
E-mail addresses: luizgomes@pucpcaldas.br (L. Gomes), ricardo.torres@ntnu.no (R. da Silva Torres), cortes@ic.unicamp.br (M.L. Côrtes).
[1] These authors contributed equally to this work.

https://doi.org/10.1016/j.infsof.2023.107217
Received 9 July 2022; Received in revised form 9 February 2023; Accepted 31 March 2023
Available online 11 April 2023

Estimating or predicting long-lived bugs are not only critical for software managers, but also they are essential for the quality assurance team. Software bugs with long fixing times can adversely affect the software quality [12]. Structural problems could emerge in complex software systems if bugs are not fixed quickly [13]. End-users may be disturbed for a long time, even when there are only a small number of bugs [14]. Users may switch software to favor the competition [15]. In short, the proper classification of long-lived bugs may increase customer satisfaction.

In the literature, despite its importance, it seems there is not a common understanding about what is a long-lived bug [14,16–18]. While some authors [14,16,19,20] consider absolute values as thresholds (based on the release cycle), others [5,7–9,11,21] consider threshold values based on the statistical distribution of bug-fixing times for each FLOSS project. These different visions may suggest that the definition of the long-live threshold is related to particular characteristics of each project (e.g., the size of the team). Despite this fact, we adopted a conservative one-year threshold that covers at least one cycle of most projects [14,16,17]. Hence, we can safely classify a bug as long-lived if it survived for more than one year (threshold value). Second, such a threshold (one year) enables us to compare the population of long-lived bugs independently and uniformly considering different projects.

Machine Learning (ML) and Text Mining (TM) techniques have solved many real-world prediction problems, including those related to automating bug report handling, such as bug severity prediction [14,18,22–25]. Advances in Natural Language Processing (NLP) have shifted the research focus from traditional to deep-learning-based techniques. Bidirectional Encoder Representations from Transformers (BERT), a deep neural learning network for NLP, have recently surpassed classical text mining approaches in various text classification tasks [26–28]. There are two approaches to adapting BERT for particular tasks: feature extraction and fine-tuning. The first method freezes model weights, and the pre-trained representations are used in a downstream model like standard feature-based approaches. In the second method, in turn, the pre-trained model can be unfrozen and fine-tuned on a new task [29].

There are broad research efforts toward the use of BERT and the effect of its contextual embedding in Software Engineering tasks [30–40]. Even so, to the best of our knowledge, few studies applied it in long-lived bug prediction task [41]. Ardimento et al. [41], for instance, yielded relevant results in their experiment using fine-tuning in this prediction task. However, they carried out their investigation on only one dataset extracted from LiveCode BTS with a relatively small number of bug reports.

In this paper, we perform a comparative study related to the use of BERT-based feature extractors and a fine-tuning approach for long-lived bug prediction. Furthermore, we compare their accuracy with the traditional TF-IDF on six popular FLOSS projects. In this context, we evaluate the long-lived prediction accuracy of five well-known machine learning classifiers when using BERT and TF-IDF as feature extractors or BERT fine-tuning.

This specific goal leads to the definition of the following Research Questions (RQs) addressed in our study:

1. How accurate are ML classifiers when they use BERT as a feature extractor in predicting long-lived bugs?
2. What is the comparative accuracy of ML classifiers when predicting long-lived bugs using BERT-based feature extraction or TF-IDF-based feature extraction?
3. Are smaller BERT variants with fine-tuning better than the approach that employs BERT as a feature extraction method for long-lived bug prediction?

The contributions of this paper can be summarized as follows:

- Evaluation of traditional ML classifiers accuracy when they used features extracted with BERT for long-lived bug prediction;
- A quantitative accuracy comparison between BERT and TF-IDF for extracting features on the long-lived prediction task;
- Evaluation of the BERT fine-tuning and smaller BERT variants impact on accuracy in the long-lived prediction task.

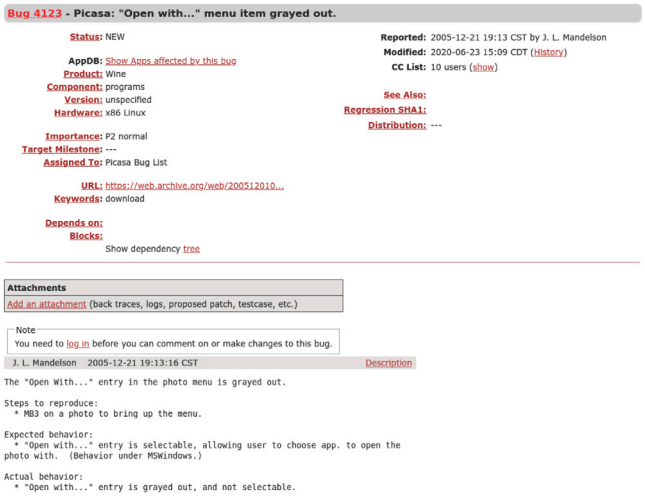


Fig. 1. A bug report example from WineHQ (https://bugs.winehq.org/show_bug.cgi?id=4123), as of June 2022.

We organized the paper as follows. Section 2 provides background concepts related to bug-tracking systems, text mining, and machine-learning techniques. Section 3 presents related work. Section 4 describes the methodology used. Section 5 reports our results. Section 6 describes the significance of our findings and how they can be interpreted. Section 7 describes the main threats to the validity of our research. Finally, Section 8 states our conclusions and highlights possible future research associated with our findings.

2. Background concepts

We provide in this section a set of fundamental concepts related to the main contributions of the paper.

2.1. Bug Tracking Systems & bug report

Bug Tracking System (BTS) [1] is a system that allows users to open bug reports and track associated information. A bug report may refer to change requests, bug fixes, or technical support that could occur during the life cycle of any given software. It is usually a form that a user should fill out to communicate a bug. This information is required to reproduce, diagnose, and fix a bug. Fig. 1 illustrates a bug report from the WineHQ project containing data that describe bug 4123. If we observe the **Status**, **Reported**, and **Modified** fields in this bug report, we can note that this bug still exists as of January 2023 even though been reported on 2005-12-21.

After the user has reported a bug, the development team is responsible for its assessment, which consists of its approval or rejection. In case of approval, the team may provide complementary information, for example, assigning a person responsible for the request or defining the severity level for the bug. A bug report initially is said to be *Unconfirmed*. The development team can change this status to *Resolved*, if a bug is not confirmed, or to *New*. The bug report state is changed to *Assigned* when someone is in charge of fixing the bug. Therefore, in the standard flow, the bug report status is assigned to *Resolved* (bug fixed), then *Verified* (bug checked), and finally *Closed*. During the bug report lifecycle, there may be other state transitions. The states for a bug report can vary from project to project, as projects can add or

remove states to match their process. All changes that are registered in a bug report are often stored in a repository, keeping valuable historical information about a particular software [42].

2.2. Machine Learning

Machine Learning (ML) is a field of study of Artificial Intelligence (AI). It gives computers the ability to learn and improve from data without being explicitly programmed [43,44]. There are many types of ML algorithms: supervised, unsupervised, semi-supervised, and reinforcement learning. The long-lived bug prediction is considered a supervised learning task. A supervised algorithm builds a model based on historical training data features. It then uses the built model to predict the output or class label for a new sample.

2.2.1. Classifiers

An ML algorithm works over a dataset, which contains many samples x_i, where $i = 1, 2, \ldots, n$. Each instance is composed of $\{x_{i1}, x_{i2}, \ldots, x_{id}\}$ input attributes or independent variables, where $d = 1, 2, \ldots, m$, and one output attribute or dependent variable, $x_{i(m+1)}$. Input attributes are often called features, and output attributes are target labels in Machine Learning. Many ML algorithms can be used in more than one learning task. However, this paper regards the selected algorithms only in the classification scenario. A brief description of each classifier used in our experiments is presented below [45]:

- **k-Nearest Neighbors (KNN)** classifies a new sample based on the geometric distance to the k-nearest labeled neighbors. The KNN commonly quantifies the proximity among neighbors using the Euclidean distance. Each instance in a dataset represents a point in an n-dimensional space in order to calculate this distance.
- **Naïve Bayes (NB)** decides to which class an instance belongs based on the Bayesian theorem of conditional probability. The probabilities of an instance belonging to each of the C_k classes given the instance x is $P(C_k|x)$. Naïve Bayes classifiers assume that, given the class variable, the value of a particular feature is independent of the value of any other feature.
- **Neural Network (NN)** is a classifier that is inspired by the structure and functional aspects of biological neural networks [46]. It is structured as a network of units called neurons, with weighted, directed connections. Neural network models have been demonstrated to be capable of achieving remarkable performance in document classification [47].
- **Random Forest (RF)** relies on two core principles: (i) the creation of hundreds of decision trees and their combination into a single model; and (ii) the final decision based on the majority of the considered trees [48].
- **Support Vector Machine (SVM)** is a classifier in which each feature vector of each instance is a point in an n-dimensional space. In this space, SVM learns an optimal way to separate the training instances according to their class labels. The output of this classifier is a hyperplane, which maximizes the separation among feature vectors of different classes. Given a new instance, SVM assigns a label based on which subspace its feature vector belongs to [49].

A common criterion to assess the prediction performance of classifiers relies on the use of the Balanced Accuracy [45,50,51]. The Balanced Accuracy is calculated as:

$$Balanced\,Accuracy = \frac{\frac{TP}{TP+FN} + \frac{TN}{FP+TN}}{2}, \quad (1)$$

where:

- **True Positive (TP)**: number of instances that were correctly predicted by the classifier as positive.
- **True Negative (TN)**: number of instances that were correctly predicted by the classifier as negative.
- **False Positive (FP)**: number of instances that were incorrectly predicted by the classifier as positive.
- **False Negative (FN)**: number of instances that were incorrectly predicted by the classifier as negative.

2.2.2. Hyperparameter tuning

Adjusting the hyperparameters is critical in Machine Learning. The goal is to identify parameter values that lead to optimal model accuracy. Each classifier has its set of hyperparameters, and these values can significantly impact the classifier performance [52].

Hyperparameter tuning is an iterative activity that occurs during the training phase of an ML model-building process. In typical protocols, researchers often employ three standard procedures [53]: (i) following default values specified in software packages, (ii) manual configuration based on the literature, experience, or trial-and-error procedures, or (iii) configuring them for optimal predictive performance by using tuning approaches (e.g., grid search or random search). After adjusting the hyperparameters for a selected classifier, researchers should train the model again until the predicting model achieves satisfactory prediction accuracy. When this goal is reached, the predictive modeling process can be considered complete.

2.3. Text mining

Common ML classifiers cannot directly process unstructured text (e.g., bug reports' *summary* and *description* attributes). Therefore, unstructured text documents are often encoded into more suitable representations in a preprocessing task. Next, the converted content is represented by feature vectors (points of an n-dimensional space). Text mining is the process of transforming unstructured text to fit into the Machine Learning pipeline [54]. It is composed of three primary activities [55]:

- **Tokenization** is the action of parsing a character stream into a sequence of tokens by splitting the stream at delimiters. A token is a block of text or a string of characters (without delimiters such as spaces and punctuation), a useful portion of the unstructured data.
- **Stop word removal** eliminates commonly used words that do not provide relevant information to a particular context, including prepositions, conjunctions, articles, verbs, nouns, pronouns, adverbs, and adjectives.
- **Stemming** is the process of reducing or normalizing inflected (or sometimes derived) words into their word stem or base form (e.g., "working" and "worked" into "work").

Two of the most traditional ways of representing a document rely on the use of a bag of words (unigrams) or a bag of bigrams (when two terms appear consecutively, one after the other) [54]. In this approach, all terms represent features, and thus the dimension of the feature space is equal to the number of different terms in all documents (in our context, bug reports).

Methods for assigning weights to features may vary. Two common approaches are Term Frequency (TF) and Term Frequency-Inverse Document Frequency (TF-IDF). The former method considers the number of times in which the term appears in each document. The latter method is a more complex weighting scheme that considers the frequency of the term in each document and in the whole collection. The importance of a term in the TF-IDF scheme is proportional to its frequency in a document and inversely proportional to its frequency in the collection [56].

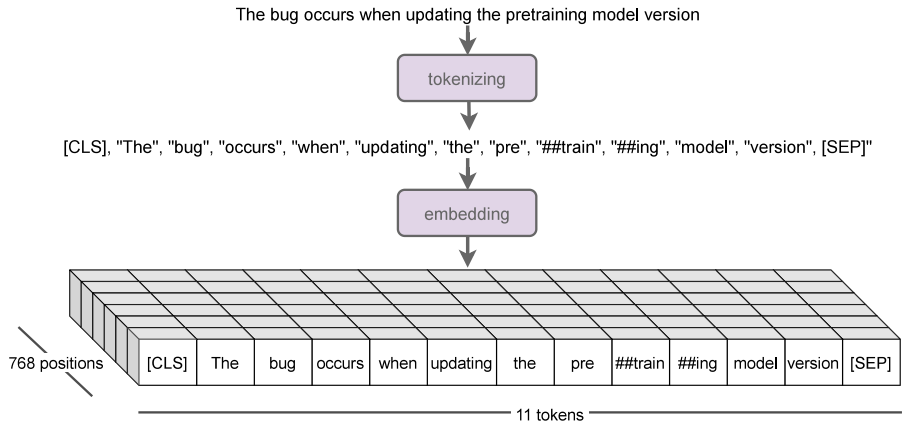


Fig. 2. $BERT_{BASE}$ input representation. Since the $BERT_{BASE}$ has 768 hidden units, each token size representation will be 768 positions. *Source:* Figure adapted from Ravichandiran et al. [59].

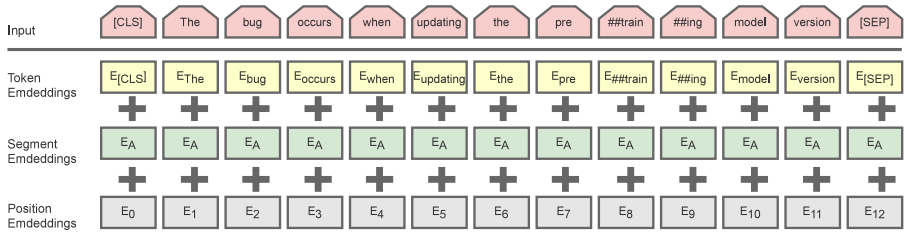


Fig. 3. Input representation in BERT, adapted from Devlin et al. [26]. Input tokens are represented in red color. Token embeddings are represented in yellow color. Segment embeddings are represented in green color. Position embeddings are represented in gray color.

2.4. Bidirectional Encoder Representations from Transformers

Bidirectional Encoder Representations from Transformers (BERT) [26] is a deep learning language model which has been successfully utilized in many NLP tasks, such as question answering and text classification [26,57,58]. Unlike other context-free models, which ignore the context and always give the same static embedding for the word, BERT is an embedding model that can "understand" the context and then generate the dynamic embedding for the word on a given context [59].

BERT considers a sentence as any sequence of tokens, and its input can be a single sentence or a pair of sentences. The token embeddings are generated from a vocabulary built over Word Piece embeddings with 30,000 tokens. Fig. 2 shows an example of token embeddings for the sentence "The bug occurs when updating the pretraining mode version" [59]. As the Word Piece tokenizer did not recognize the "pretraining" word, it will be split into subwords (e.g., "pre", "##train", and "##ing") until the tokenizer finds the subword or until it reaches individual characters, which is handled as the Out-of-Vocabulary (OOV) word.

BERT adds the [CLS] token at the beginning of the first sentence and is used for classification tasks. This token holds the aggregate representation of the input sentence. The [SEP] token indicates the end of each sentence [59]. Fig. 3 shows the embedding generation process executed by the Word Piece tokenizer. First, the tokenizer converts input sentences into tokens before figuring out token embeddings. Thereafter, these tokens feed segment and position embedding computations. Lastly, the tokenizer integrates all embeddings and inputs the results into BERT. Fig. 3 illustrates three kinds of embeddings [26].

- **Token embedding**: the word embedding represented by a vector of numbers.
- **Segment embedding**: the fixed embedding that defines if a token belongs to the first or the second sentence if this second sentence was inputted.
- **Position embedding:** the position embedding for each token in a sentence. If there are two sentences, the position in the second sentence continues from the last position of the first sentence plus one.

The BERT model is pre-trained from two approaches: masked language modeling and next-sentence prediction. In the first approach, 15% of the word piece input tokens are randomly masked, and the network is trained to predict masked words. The model then reads the sentence in both directions to predict the masked words. In the second approach, BERT receives two sentences as input and has to predict whether the second one is the subsequent sentence to the first one.

2.4.1. BERT variants

BERT has many variant models. Their proposal often aimed to make the original formulation more efficient or be more adapted to several contexts. Table 1 shows the descriptions of variants used in our research.

3. Related work

Giger et al. [21] conducted experiments on bug reports from six FLOSS projects hosted by Eclipse, Mozilla, and Gnome, and proposed a classifier based on a decision tree classifier to classify bugs into "fast" or "slow". Furthermore, they empirically demonstrated that the addition of post-submission bug report data of up to one month in the feature vector might improve the model performance.

Lamkanfi et al. [64] observed that a fraction of the conspicuous fix-times bug reports is often fixed within a few minutes in Eclipse and Mozilla. The authors proposed to filter out these conspicuous bug reports when using data mining techniques to predict the fixing times of reported bugs.

Zhang et al. [4] performed an empirical study on bug fixing times in three projects of CA Technologies company. They proposed a model

Table 1
BERT variants. (L) the number of layers, (H) the number of hidden units, (A) the number of self-attention operations, and (P) millions of parameters.

Model	Description	L	H	A	P
$ALBERT_{BASE}$ [60]	A "lite" version of BERT with greatly reduced number of parameters.	12	768	12	12
$BERT_{L2H128A2}$ [61]	A smaller BERT model is intended for environments with restricted computational resources.	2	128	2	0.5
$BERT_{L4H256A4}$ [61]	A smaller BERT model is intended for environments with restricted computational resources.	4	256	4	4
$BERT_{L4H512A8}$ [61]	A smaller BERT model is intended for environments with restricted computational resources.	4	512	8	16
$BERT_{L8H512A8}$ [61]	A smaller BERT model is intended for environments with restricted computational resources.	8	512	8	32
$BERT_{BASE}$ [26]	A smaller BERT model is intended for environments with restricted computational resources.	12	768	12	105
DISTILBERT [62]	A small, fast, cheap, and light Transformer model trained by distilling BERT base cased model.	6	768	12	55
$ELECTRA_{SMALL}$ [63]	ELECTRA is a BERT-like model that is pre-trained as a discriminator in a set-up resembling a generative adversarial network (GAN)	4	512	8	24
$ELECTRA_{BASE}$ [63]	ELECTRA is a BERT-like model that is pre-trained as a discriminator in a set-up resembling a generative adversarial network (GAN)	12	768	12	105

based on a Markov chain to predict the number of bugs that could be fixed in the future. Furthermore, they employed a Monte Carlo simulation to predict the total fixing time for a given number of bugs. Moreover, they classified bugs as "fast" and "slow" regarding different threshold times. Akbarinasaji et al. [12] replicated the study performed by Zhang et al. [4]. Rather than a CSS project, Akbarinasaji et al. [12] investigated an open-source software project and confirmed the results achieved by Zhang et al. [4].

Saha et al. [14] extracted the bug repositories from seven well-known FLOSS projects and analyzed long-lived bugs from five different perspectives: proportion, severity, assignment, reasons, and the nature of fixes. Although less frequent than short-lived bugs, they showed a fair number of long-lived bugs in FLOSS projects, and more than 90% of them negatively affected user experience. The reasons for these long-lived bugs are many, including, for example, longer assignment time and the lack of understanding of their priority. However, many bugs resulted in long-lived bugs without a specific reason.

Rocha et al. [25] characterized the workflow followed by Mozilla Firefox developers when resolving bugs. They proposed the concept of bug flow graphs (BFG) to help understand the workflow. They concluded that (a) when a bug is not formally assigned to a developer, it takes ten more days to be resolved; (b) approximately 94% of duplicate bugs are resolved within two days or less after they appear in the tracking system; (c) incomplete bugs, which are never assigned to developers, usually take 70 days to be closed; (d) more skilled developers resolve bugs faster in comparison to less skilled ones; (e) for less skilled developers, assigning a person responsible for the bug usually takes more time in comparison to the time taken to fix the bug.

Habayeb et al. [65] proposed a novel approach using Hidden Markov models and temporal sequences to predict when a bug report will be closed. The approach is empirically demonstrated using eight years of bug reports collected from the Firefox project. The results indicate around 10% higher accuracy than the frequency-based classification approaches.

Ardimento et al. [41] proposed a method based on $BERT_{BASE}$ to predict bug-fixing time. Their approach combined description and comments attributes of a bug report to provide for the $BERT_{BASE}$ neural network. They evaluated their proposed model on Live Codee, a large-scale open-source project, and they claimed that their proposed approach has an effective ability to predict a bug in "slow" or "fast". They used the median to label each bug report in the training dataset in "slow" and "fast".

Sepahvand et al. [9] proposed an approach based on Long Short-Term Memory (LSTM) to classify a bug in short fixing time or long fixing time. The class of each bug report was determined by applying a threshold. Short-fixing time class is assigned for bugs with a fixing time lesser than the threshold, and a long-fixing time class is for others. The threshold was the median of all bug-fixing time extracted from Mozilla from 2008 to 2014. The results indicate that the proposed method had better performance than the hidden Markov-based [65] model in the same task.

In our previous work [24], we investigated the population of long-lived bugs in six popular FLOSS. Also, we confirmed a significant percentage of long-lived bugs in these projects; we characterized them using many bug report attributes, which confirmed some differences between short- and long-lived bugs. Furthermore, we compared the accuracy of five well-known ML classifiers and traditional text mining techniques in long-lived bug prediction. Our experiments used unstructured text attributes and demonstrated that it is possible to predict long-lived bugs with good accuracy using basic methods.

There are broad research efforts toward the use of BERT and the effect of its contextual embedding in Software Engineering tasks: identifying correct patches [30], searching and documenting code [31], representing data flow [32], summarizing code [34], predicting defects [35], detecting and repairing bug [36], extracting software requirements [37], traceability of software artifacts [38], and localizing bugs [40].

The studies above present relevant results for researchers in this area. However, we can observe shortcomings in some works that investigated "long-lived prediction" itself. First, studies often used a few ML classifiers: Decision Tree [21], Markov Chain [4], LSTM [9], and BERT [41]. Different from them, in this paper, we perform a more comprehensive investigation including different feature extractors and classifiers. We compare the accuracy of five well-known ML classifiers' long-lived bug prediction using BERT and TF-IDF as feature extractors. Also, we detail and discuss cases of success associated with the best predictors.

Furthermore, another limitation of those studies refers to the fact that they considered few FLOSS projects. In this paper, we consider six popular FLOSS projects (Eclipse, Freedesktop, Gnome, GCC, Mozilla, and WineHQ) in our evaluation protocol.

4. Methodology

Fig. 4 presents the methodology used in our experiments to address the proposed research questions. The methodology follows the traditional machine learning pipeline that comprises four main steps: data collecting, feature engineering, model training, and model testing.

It is worth noticing that we used only $BERT_{BASE}$ for feature extraction among existing BERT models. For simplicity, for the rest of this document, we will use just BERT to stand for the $BERT_{BASE}$ model.

4.1. Data collecting

First, we downloaded bug reports from Bugzilla repositories of six FLOSS projects which are fairly used in research papers [1,2,24,64,66,67]: Eclipse, Freedesktop, GCC, Gnome, Mozilla, and WineHQ. Then, we extracted, investigated, and interpreted their data structure and labeled each bug report as short-lived or long-lived.

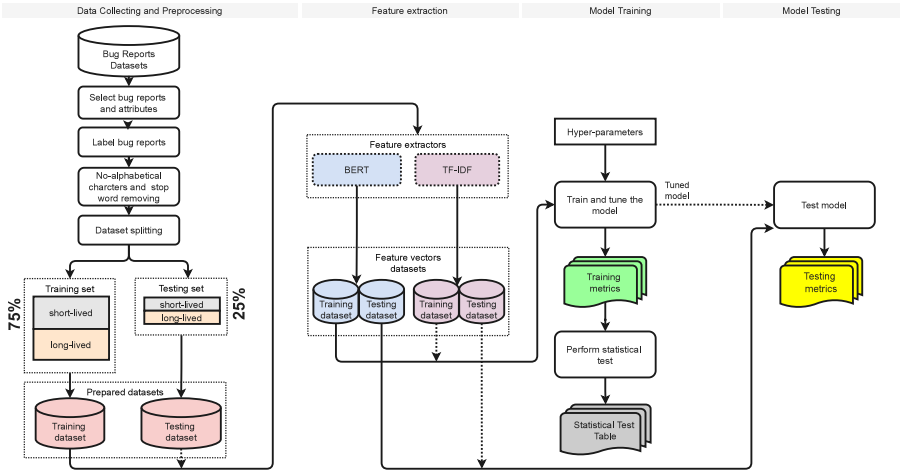


Fig. 4. The four steps of the experiment methodology are based on a typical machine learning pipeline: data collecting and preprocessing, feature extraction, model training, and model testing.

Table 2
FLOSS projects used in our research.

Project	URL (As of June 2022)	Number of bug reports		
		Total	Training set	Testing set
Eclipse	https://www.eclipse.org	9540	7155	2385
Freedesktop	https://www.freedesktop.org	7626	5719	1907
GCC	https://gcc.gnu.org/	9961	7470	2491
Gnome	https://www.gnome.org	7755	5816	1939
Mozilla	https://www.mozilla.org	9776	7332	2444
WineHQ	https://www.winehq.org	6058	4543	1515

Lastly, we stored them in one file per project in a CSV format. These datasets are publicly available at https://data.mendeley.com/datasets/v446tfssgj/2 (as of June 2022), and Table 2 shows additional information about them.

4.2. Data preprocessing

Raw data collected from FLOSS' Bugzilla BTS were not suitable for training and testing steps in ML pipeline [68]. The recommended approach to convert the raw data to an appropriate format is to run procedures to extract, organize, and structure relevant features to address the proposed research questions. To accomplish this, we wrote specific codes to perform the following data preprocessing tasks:

- Selection of bug reports with a *Closed* or *Solved* status and a *Fixed* resolution status. The development team effectively fixed this bug report. It can no longer have altered its resolution date.
- Choosing of relevant attributes: *bug id*, *opened date*, *description*, *resolution date*;
- Computation of the bug fix time in days (the resolution date minus the open date). We considered the resolution date as the ground truth.
- Labeling each bug report in 'short' or 'long-lived' based on its bug fix time. We labeled bugs with bug fix time less or equal to its median as 'short-lived'; otherwise, as 'long-lived' [24].
- Cleaning bug report description text to remove out non-alphabetical characters and English stop words.

Finally, we split each dataset into training (75% of bug reports total) and testing (25% of bug reports total) sets.There is no fixed rule for selecting the size of the training set or testing set. A good rule of thumb is 75%–80% for training sets. Higher proportions are a good idea if the number of repetitions in cross-validation is large [51]. A 75/25 split was used to compare the results of BERT-based methods with those described in [18].

4.3. Feature extraction

After the data preprocessing step, we used two distinct strategies to extract features from bug report descriptions for comparing them in long-lived prediction task, as shown in (Fig. 4): BERT or TF-IDF. Although BERT allows 512 tokens per input sentence [26], for saving memory resources and faster training of the pre-training model, we selected the 128 first tokens of the bug report description [28,41]. Then, we generated an embedding feature vector with 768 positions from the aggregate representation of the input sentence denoted by [CLS] token of BERT. In turn, we used the 128 words with the highest score in TF-IDF from the bug report description. In this way, TF-IDF generated a feature vector of inverse-frequency words with 128 positions.

4.4. Model training

To train our models for the long-lived bugs predicting task, we selected the five well-known ML classifiers described in Section 2. They were implemented them using Scikit-learn[2] – a Python Machine Learning Library to build the predictive models, including a grid search procedure to select the best hyperparameters for each classifier in training among the hyperparameters indicated in Table 3.

The scripts, using the before-mentioned library, evaluated each model using the Balanced Accuracy metric and reported the resulting values. To select the best model for each ML classifier, we trained and tested each model using the Repeated 10 × 5 Fold Cross-Validation technique [69],[3] as shown in Fig. 5.

Table 3 presents the hyperparameters for each ML classifier used in our study.

Finally, we performed the Wilcoxon signed-rank statistical test [70] (with a significance level of 95%) to evaluate the statistical significance among the ML classifier accuracies based on values reported in the Repeated Cross-Validation procedure (Fig. 6). We ran these steps either for BERT or TF-IDF feature vectors independently.

[2] https://scikit-learn.org/stable/ (As of June 2022).

[3] Repeated Cross-Validation $n \times k$: divides a dataset into k folds in n iterations. In each iteration, it saves a different fold for testing and uses all the others for training [51].

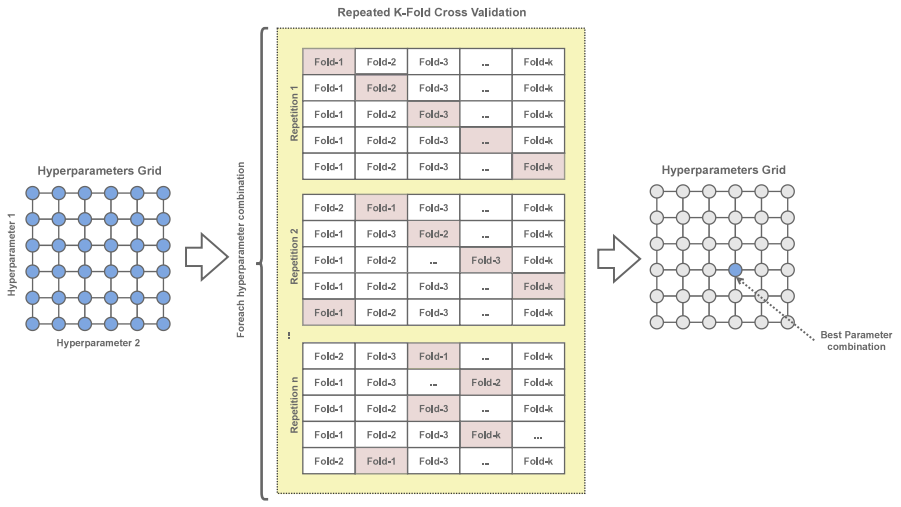


Fig. 5. The procedure of repeated cross-validation. The salmon colored folder partition is the test partition, and the others are the training partitions. The procedure involves repeating the cross-validation procedure multiple times and reporting the mean result for each hyperparameter combination across all folds. The best hyperparameter for the final model is determined by the mean standard error from each training iteration. This paper assumes that $k = 5$ and $n = 10$.

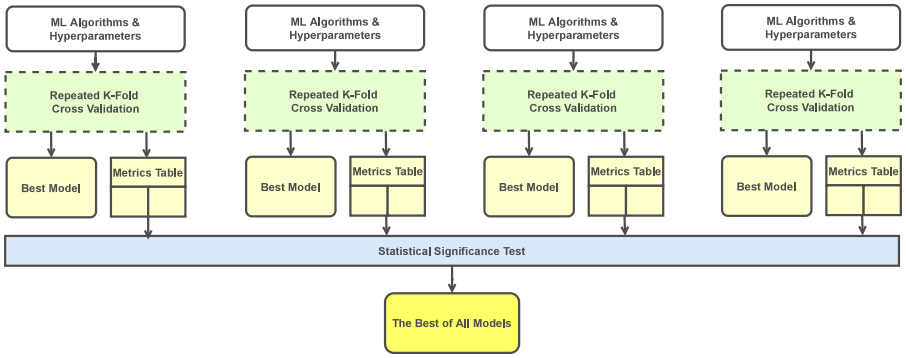


Fig. 6. The statistical test schema is based on metrics from repeated k-fold cross-validation of the best model yielded. For each ML algorithm pair, the test is conducted to determine if there is a statistical difference between them. Finally, the best of all models is chosen.

Table 3
Description of the hyperparameters for each ML classifier investigated.

ML classifier	Hyper-parameters
KNN	**k**: Number of neighbors
Naïve Bayes	**var_smoothing**: Portion of the largest variance of all features that is added to variances for calculation stability.
Neural network	**size**: Hidden units **activation**: Activation function for the hidden layer. **solver**: The solver for weight optimization. **alpha**: L2 penalty (regularization term) parameter.
Random Forest	**max_features**: The number of features to consider when looking for the best split. **n_estimators**: The number of trees in the forest.
SVM	**C**: Regularization cost parameter **gamma**: Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. **kernel**: Specifies the kernel type to be used in the algorithm.

4.5. Model testing

In the testing phase, each of the best long-lived prediction models was validated with 25% of each bug report testing dataset to measure its balanced accuracy in an unknown dataset. Furthermore, we ran this step for either BERT or TF-IDF feature vectors, separately.

5. Results

This section reports our experiments' results and is organized according to the raised research questions (Section 1).

5.1. RQ_1. *How accurate are ML classifiers when they use BERT as a feature extractor in predicting long-lived bugs?*

Table 4 shows the long-lived bug prediction performance of ML classifiers for the six projects. In this experiment, we considered the features extracted from the bug report's description using BERT. In the figure, we can observe that SVM was the best in three datasets: 59.5% in Freedesktop, 56.8% in GCC, and 61.5% in Mozilla. Also, we can see that SVM was slightly worse than Random Forest (RF) in Eclipse (57.4% versus 57.6%) and Gnome (56.4% versus 56.6%), and k-NN (57.3% versus 57.9%) in WineHQ.

Table 5 shows a pairwise comparison of the statistical significance for each ML classifier's balanced accuracy pair yielded during the Repeated Cross-Validation process in the training step, as mentioned in the Methodology section. We can observe that the SVM classifier (highlighted in blue) outperformed the others with statistical significance in many cases.

Table 4
The performance for all classifiers over all datasets based on the balanced accuracy metric. The underlined results refer to the best classifier when using the BERT- or the TF-IDF-based feature extraction for each dataset in the testing step.

	BERT					TD-IDF				
	KNN	NB	NN	RF	SVM	KNN	NB	NN	RF	SVM
Eclipse	0.555	0.563	0.533	**0.576**	0.574	0.504	0.505	0.505	0.508	**0.525**
Freedesktop	0.549	0.569	0.531	0.563	**0.595**	0.497	0.497	0.497	0.504	**0.510**
GCC	0.542	0.535	0.554	0.557	**0.568**	0.497	**0.517**	0.497	0.514	0.502
Gnome	0.554	**0.566**	0.528	**0.566**	0.564	0.520	**0.556**	0.537	0.555	0.540
Mozilla	0.595	0.580	0.570	0.606	**0.615**	0.500	0.519	**0.521**	0.513	0.515
WineHQ	**0.579**	0.550	0.535	0.562	0.573	0.505	0.536	0.515	**0.541**	0.527

Table 5
The Wilcoxon Signed-Rank statistical significance for predicting long-lived bugs. Assuming a significance level $\alpha = 0.05$, each cell shows the results of one classifier paired with the other. The '–' indicates there is no statistical difference between the pairs of classifiers ($p > \alpha$). The direction of arrows Left ('←') and up ('↑') indicates the most accurate classifier in terms of balance accuracy, when ($p \leq \alpha$).

		KNN	NB	NN	RF	SVM		KNN	NB	NN	RF	SVM
Eclipse	KNN	–	↑	←	↑	↑	Freedesktop	–	↑	←	↑	↑
	NB	←	–	←	–	–		←	–	←	↑	↑
	NN	↑	↑	–	↑	↑		↑	↑	–	↑	↑
	RF	←	–	←	–	–		←	←	←	–	↑
	SVM	←	–	←	–	–		←	←	←	←	–
Gcc	KNN	–	–	–	↑	↑	Gnome	–	–	←	↑	↑
	NB	–	–	–	↑	↑		–	–	←	↑	↑
	NN	–	–	–	↑	↑		↑	↑	–	↑	↑
	RF	←	←	←	–	↑		←	←	←	–	–
	SVM	←	←	←	←	–		←	←	←	–	–
Mozilla	KNN	–	–	←	↑	↑	WineHQ	–	↑	–	↑	↑
	NB	–	–	←	↑	↑		←	–	←	↑	↑
	NN	↑	↑	–	↑	↑		–	↑	–	↑	↑
	RF	←	←	←	–	↑		←	←	←	–	↑
	SVM	←	←	←	←	–		←	←	←	←	–

5.2. RQ_2. *What is the comparative accuracy of ML classifiers when predicting long-lived bugs using BERT-based feature extraction or TF-IDF-based feature extraction?*

Table 4 also shows a comparison between the classifiers' accuracy performance when the long-lived bug prediction relies on features extracted from the bug report's description using BERT or TF-IDF. We can observe that the long-lived bug prediction's general performance using BERT-based feature extraction was systematically better than TF-IDF in all datasets. The most remarkable difference (9.4%) between classifiers' performance occurred in the Mozilla dataset when SVM with BERT reached 61.5% and Neural Network, 52.1%. Only in Gnome, the Neural Network with TF-IDF was slightly better than this classifier with BERT.

Fig. 7 summarizes the accuracy performance difference between ML classifiers using feature extraction based on BERT and TF-IDF for all project datasets. The highest difference in favor of BERT was observed for Mozilla and the lowest, for Gnome.

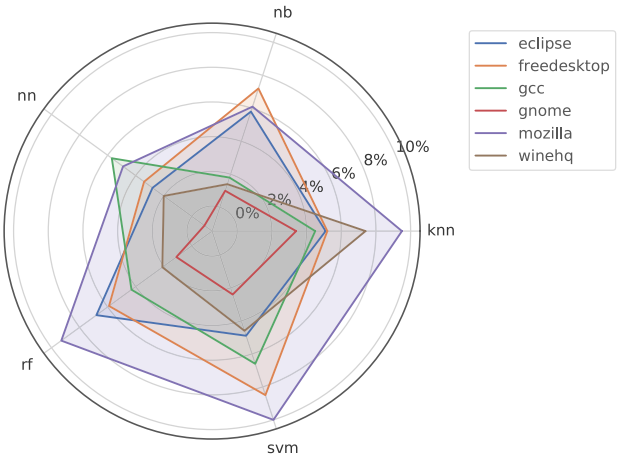


Fig. 7. The chart shows the percentage of improved performance obtained by the studied ML algorithms by using BERT as a feature extractor over the TF-IDF alternative. A vertex indicates the percentage of balanced accuracy gains of the best BERT model over the best TD-IDF model in each dataset.

5.3. RQ_3: *Are smaller BERT variants with fine-tuning better than the approach that employs BERT as a feature extraction method for long-lived bug prediction?*

Fig. 8 compares the balanced accuracy performance of the BERT variant's on the long-lived prediction task. We can note that the long-lived bug prediction's general performance using BERT variants in fine-tuning mode was systematically better than TF-IDF. The best performance by dataset was yielded by $\text{BERT}_{L2H128A2}$ in Eclipse (56.8%), ELECTRA_{BASE} in Freedesktop (57.6%), $\text{BERT}_{L2H128A2}$ in GCC (57.0%), DISTILBERT in Gnome (57.1%), $\text{BERT}_{L4H256A4}$ in Mozilla (59.8%), and $\text{BERT}_{L4H512A8}$ in WineHQ (59.8%). BERT variants had a performance worse than TF-IDF in four datasets: ALBERT_{BASE} in Eclipse (51.6%) and GCC (50.0%), BERT_{BASE} in Gnome (54.9%), and $\text{BERT}_{L8H512A8}$ in WineHQ (53.7%). Furthermore, the BERT variants with fine-tuning overcame the BERT_{BASE} as feature extractor in 3 of 6 datasets: $\text{BERT}_{L2H128A2}$ in GCC (+0.7%), DISTILBERT in GNOME (+0.5%), and $\text{BERT}_{L4H512A8}$ in WineHQ (+3.2%). Finally, we can observe that smaller BERT variants overcame the larger BERT variants in 5 of 6 datasets.

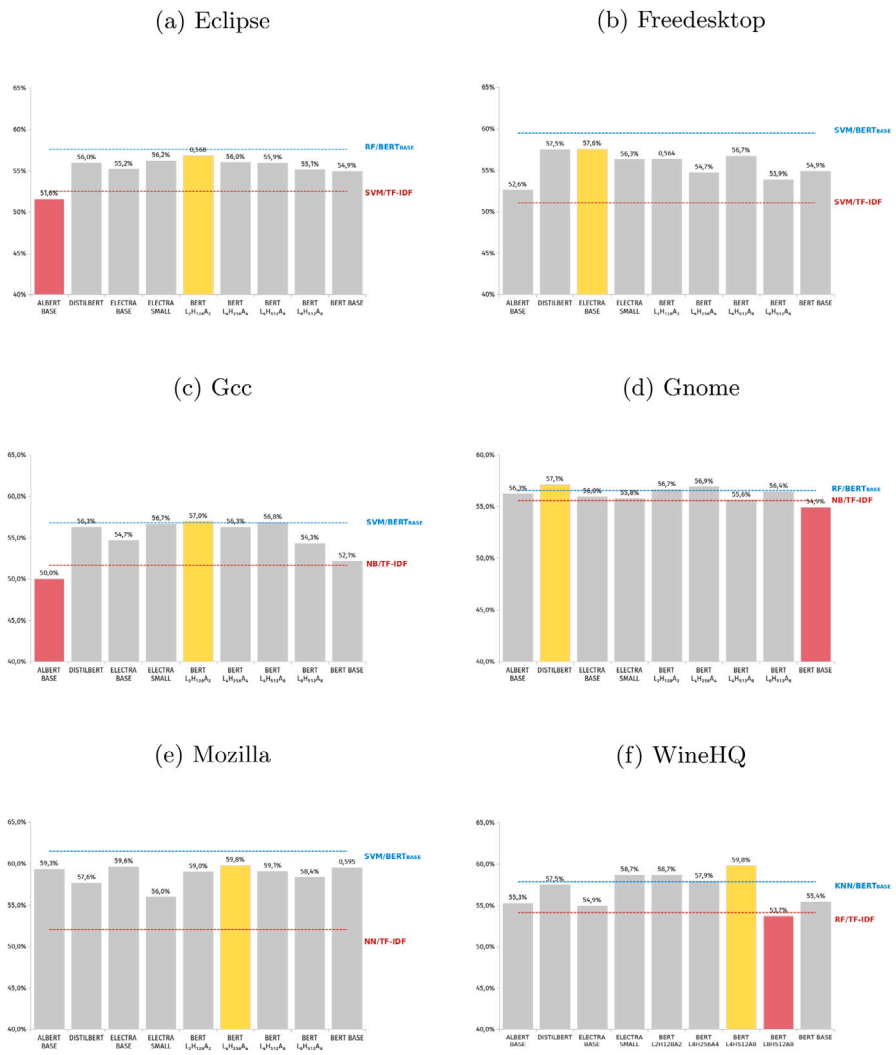


Fig. 8. The balanced accuracy performance for classifiers based on BERT variants to all datasets. The red line indicates the best ML algorithm in performance using TD-IDF as feature extraction, and the blue line indicates the best ML algorithm in performance using BERT_{BASE}.

6. Discussions

This section discusses the significance of our findings and it is divided by research question, as the Results section.

6.1. RQ_1. *How accurate are ML classifiers when it uses BERT as features extractor in predicting long-lived bugs?*

The results from our experiment to answer the first proposed research question indicate that BERT-based feature extraction leads to a good performance in the long-lived bug prediction problem. Most of the experimental predicting models outperformed reasonably the random prediction, which is 50% of the probability of predicting a long-lived bug (Table 4). The SVM and Random Forest models seem to have drawn a more precise decision boundary based on BERT contextual sentence embedding in the testing phase. Thus, they could separate short-lived bugs from long-lived bugs more accurately than other classifiers. Furthermore, using a pre-trained network minimized the side effects of dataset size in the prediction performance. Our datasets are relatively small and of different sizes. Even so, the best performance score in each dataset was similar. This performance accuracy can be considered very good for an initial experiment where we used the basic BERT architecture and fed the feature extractor with only 128 headed tokens from only one bug report attribute.

6.2. RQ_2. *What is the comparative accuracy of ML classifiers when predicting long-lived bugs using BERT-based feature extraction or TF-IDF-based feature extraction?*

Our results demonstrated that BERT-based feature extraction is better than TF-IDF-based for long-live bug prediction tasks in investigated FLOSS projects. The first extraction method was systematically better than the second in most datasets for most ML classifiers. It seems that contextual embedding and dense representation may have a superior generalization capability that can capture similarities among bugs of the same class (short- or long-live bug). These results confirm the strength of the contextual embedding and denser feature vector over the traditional bag-of-word strategy [27].

6.3. RQ_3. *Are smaller BERT variants with fine-tuning better than feature extraction method for long-lived bug prediction?*

Finally, our results have shown that BERT variants with smaller architectures ($\text{BERT}_{L2H128A2}$, $\text{BERT}_{L4H512A8}$, $\text{BERT}_{L8H512A8}$, and DISTILBERT) had a performance in most cases superior to BERT variants with larger architectures (BERT_{BERT}, ALBERT_{BERT}, and ELECTRA_{BERT}). These results may be explained by reduced overfitting in smaller than larger deep neural networks. Furthermore, both feature-extraction and fine-tuning BERT-based classifiers in most cases overcame classifiers-based on TF-IDF features, which confirms the strength of the contextual

embedding and denser feature vector over the traditional bag-of-word strategy [27].

7. Threats to validity

The main threats to the validity of this study are summarized below:

- We have assumed that bug fix times extracted from repositories are correct. However, the actual time efforts spent by developers involved in bug-fixing are not publicly available.
- We have considered six repositories, which lead to a collection composed of more than 50,000 bug reports. Although they are not representative of the population of all open-source and commercial projects, the characteristics presented by them are similar to those shown in other studied repositories [14,21,64].
- Another limitation is the lack of investigation regarding the impact of different quantities and truncation methods over description bug report attributes.

8. Conclusions

Our paper investigated the impact of BERT-based feature extraction on the accuracy of ML classifiers in contrast with TF-IDF in long-lived prediction tasks. We used six well-known ML classifiers: KNN, Näive Bayes, Neural Network, Random Forest, and SVM. The datasets used in our experiments were built from bug reports extracted from six popular datasets: Eclipse, Freedesktop, Gnome, Gcc, Mozilla, and WineHQ.

The results indicated that the accuracy of ML classifiers using BERT-based feature extraction, considering only the description attribute, was very promising. The SVM and Random Forest outperform others in almost all datasets (RQ_1). In comparison, the performance of ML classifiers when they used feature extraction based on BERT was systematically better than feature extraction based on TF-IDF. The highest accuracy difference occurred in Mozilla and the lowest in the Gnome project (RQ_2). Finally, findings related to RQ_3 showed that fine-tuning on smaller BERT architectures may be a computationally cheaper choice to predict long-lived bug reports than larger architecture.

A possible venue for future research is to investigate other quantities and truncating methods (head-only, tail-only, head+tail) for extracting words from the description attribute. Another research direction is to investigate an end-to-end deep learning neural network predictor by performing the combination of fine-tuning on the BERT pre-trained model and bug report structured fields. Finally, we think that using a Graph Neural Network (GNN) [71–73] might improve the results for long-live bug prediction problems. The GNN can be used to encode relationships of bug reports and the temporal evolution of those relationships and of the reports themselves.

CRediT authorship contribution statement

Luiz Gomes: Conceptualization, Methodology, Data curation, Writing – original draft, Visualization, Investigation, Writing – reviewing and editing. **Ricardo da Silva Torres:** Supervision, Writing – reviewing and editing, Validation. **Mario Lúcio Côrtes:** Supervision, Writing – reviewing and editing, Validation.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2023.107217.

Data availability

I have shared the link to the section “Data Collecting” of our paper.

A Dataset for Long-lived Bug Prediction in FLOSS (Original data) (Mendeley Data)

Acknowledgments

The authors thank the National Council for Scientific and Technological Development - CNPq (grant #307560/2016-3), the São Paulo Research Foundation – FAPESP (grants #2014/12236-1, #2015/24494-8, #2016/50250-1, and #2017/20945-0), the FAPESP-Microsoft Virtual Institute (grants #2013/50155-0 and #2014/50715-9), and The Pontifical Catholic University of Minas Gerais (PUC-MG) that supports the first author through the Permanent Program For Professor Qualification (PPCD). This study was partially funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010, pp. 1–10.
[2] A. Lamkanfi, S. Demeyer, Q.D. Soetens, T. Verdonck, Comparing mining algorithms for predicting the severity of a reported bug, in: 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 249–258.
[3] G. Yang, S. Baek, J.-W. Lee, B. Lee, Analyzing emotion words to predict severity of software bugs: A case study of open source projects, in: Proceedings of the Symposium on Applied Computing, SAC ’17, ACM, New York, NY, USA, 2017, pp. 1280–1287.
[4] H. Zhang, L. Gong, S. Versteeg, Predicting bug-fixing time: An empirical study of commercial software projects, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 1042–1051.
[5] W. Abdelmoez, M. Kholief, F.M. Elsalmy, Bug fix-time prediction model using naïve Bayes classifier, in: 2012 22nd International Conference on Computer Theory and Applications (ICCTA), 2012, pp. 167–172.
[6] W.H.A. Al-Zubaidi, H.K. Dam, A. Ghose, X. Li, Multi-objective search-based approach to estimate issue resolution time, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, in: PROMISE, Association for Computing Machinery, New York, NY, USA, 2017, pp. 53–62, Available from: https://doi.org/10.1145/3127005.3127011.
[7] P. Ardimento, M. Bilancia, S. Monopoli, Predicting bug-fix time: Using standard versus topic-based text categorization techniques, 2016, pp. 167–182.
[8] P. Ardimento, A. Dinapoli, Knowledge extraction from on-line open source bug tracking systems to predict bug-fixing time, in: Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics, WIMS ’17, Association for Computing Machinery, New York, NY, USA, 2017, Available from: https://doi.org/10.1145/3102254.3102275.
[9] R. Sepahvand, R. Akbari, S. Hashemi, Predicting the bug fixing time using word embedding and deep long short term memories, IET Softw. 14 (3) (2020) 203–212.
[10] C. Liu, J. Yang, L. Tan, M. Hafiz, R2Fix: Automatically generating bug fixes from bug reports, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 282–291.
[11] P. Francis, L. Williams, Determining "grim reaper" policies to prevent languishing bugs, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 436–439.
[12] S. Akbarinasaji, B. Caglayan, A. Bener, Predicting bug-fixing time: A replication study using an open source software project, J. Syst. Softw. 136 (2018) 173–186.
[13] B.S. Rawal, A.K. Tsetse, Analysis of bugs in google security research project database, in: 2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS), 2015, pp. 116–121.
[14] R.K. Saha, S. Khurshid, D.E. Perry, Understanding the triaging and fixing processes of long lived bugs, Inf. Softw. Technol. 65 (2015) 114–128.
[15] M.E. Mezouar, F. Zhang, Y. Zou, Are tweets useful in the bug fixing process? An empirical study on firefox and chrome, Empir. Softw. Eng. 23 (3) (2018) 1704–1742, Available from: https://doi.org/10.1007/s10664-017-9559-4.
[16] R.K. Saha, S. Khurshid, D.E. Perry, An empirical study of long lived bugs, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 144–153.

[17] R.K. Saha, J. Lawall, S. Khurshid, D.E. Perry, Are these bugs really "normal"? in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 258–268.

[18] L.A.F. Gomes, R. da Silva Torres, M.L. Côrtes, On the prediction of long-lived bugs: An analysis and comparative study using FLOSS projects, Inf. Softw. Technol. 132 (2021) 106508, Available from: https://www.sciencedirect.com/science/article/pii/S0950584920302482.

[19] G. Canfora, M. Ceccarelli, L. Cerulo, M. Di Penta, How long does a bug survive? An empirical study, in: 2011 18th Working Conference on Reverse Engineering, 2011, pp. 191–200.

[20] L. Marks, Y. Zou, A.E. Hassan, Studying the fix-time for bugs in large open source projects, in: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise '11, Association for Computing Machinery, New York, NY, USA, 2011, Available from: https://doi.org/10.1145/2020390.2020401.

[21] E. Giger, M. Pinzger, H. Gall, Predicting the fix time of bugs, in: Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering, RSSE '10, ACM, New York, NY, USA, 2010, pp. 52–56.

[22] V.B. Singh, S. Misra, M. Sharma, Bug severity assessment in cross project context and identifying training candidates, J. Inf. Knowl. Manage. 16 (01) (2017) 1750005.

[23] N.K.S. Roy, B. Rossi, Cost-sensitive strategies for data imbalance in bug severity classification: Experimental results, in: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2017, pp. 426–429.

[24] L.A.F. Gomes, R. da Silva Torres, M.L. Côrtes, Bug report severity level prediction in open source software: A survey and research opportunities, Inf. Softw. Technol. 115 (2019) 58–78.

[25] H. Rocha, G. de Oliveira, M.T. Valente, H. Marques-Neto, Characterizing bug workflows in mozilla firefox, in: Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES 2016, Maringá, Brazil, September 19 - 23, 2016, 2016, pp. 43–52.

[26] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186, Available from: https://www.aclweb.org/anthology/N19-1423.

[27] S. González-Carvajal, E.C. Garrido-Merchán, Comparing BERT against traditional machine learning text classification, 2020, arXiv preprint arXiv:2005.13012.

[28] C. Sun, X. Qiu, Y. Xu, X. Huang, How to fine-tune BERT for text classification?, 2020.

[29] M.E. Peters, S. Ruder, N.A. Smith, To tune or not to tune? Adapting pretrained representations to diverse tasks, in: Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019), Association for Computational Linguistics, Florence, Italy, 2019, pp. 7–14, Available from: https://www.aclweb.org/anthology/W19-4302.

[30] V. Csuvik, D. Horváth, F. Horváth, L. Vidács, Utilizing source code embeddings to identify correct patches, in: 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF), IEEE, 2020, pp. 18–25.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, 2020, pp. 1536–1547, Available from: https://aclanthology.org/2020.findings-emnlp.139.

[32] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.

[33] A. Kanade, P. Maniatis, G. Balakrishnan, K. Shi, Learning and evaluating contextual embedding of source code, in: Proceedings of the 37th International Conference on Machine Learning, ICML '20, JMLR.org, 2020.

[34] R. Wang, H. Zhang, G. Lu, L. Lyu, C. Lyu, Fret: Functional reinforced transformer with BERT for code summarization, IEEE Access 8 (2020) 135591–135604.

[35] E.N. Akimova, A.Y. Bersenev, A.A. Deikov, K.S. Kobylkin, A.V. Konygin, I.P. Mezentsev, V.E. Misilov, A survey on software defect prediction using deep learning, Mathematics 9 (11) (2021) 1180.

[36] M. Allamanis, H. Jackson-Flux, M. Brockschmidt, Self-supervised bug detection and repair, Adv. Neural Inf. Process. Syst. 34 (2021).

[37] A.F. de Araújo, R.M. Marcacini, RE-BERT: Automatic extraction of software requirements from app reviews using BERT language model, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1321–1327, Available from: https://doi.org/10.1145/3412841.3442006.

[38] J. Lin, Y. Liu, Q. Zeng, M. Jiang, J. Cleland-Huang, Traceability transformed: Generating more accurate links with pre-trained BERT models, in: Proceedings of the 43rd International Conference on Sofware Engineering, Vol. 43, Available from: https://par.nsf.gov/biblio/10262158.

[39] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, X. Jiang, SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation, 2021, http://dx.doi.org/10.48550/ARXIV.2108.04556, arXiv. Available from: https://arxiv.org/abs/2108.04556.

[40] W. Zou, E. Li, C. Fang, BLESER: Bug localization based on enhanced semantic retrieval, 2021, arXiv preprint arXiv:2109.03555.

[41] P. Ardimento, C. Mele, Using BERT to predict bug-fixing time, in: 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS), 2020, pp. 1–7.

[42] T. Zhang, G. Yang, B. Lee, A.T.S. Chan, Predicting severity of bug report by mining bug repository with concept profile, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, ACM, New York, NY, USA, 2015, pp. 1553–1558.

[43] A. Géron, Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques To Build Intelligent Systems, O'Reilly Media, 2019.

[44] P. Flach, Machine Learning: The Art and Science of Algorithms that Make Sense of Data, Cambridge University Press, New York, NY, USA, 2012.

[45] S. Marsland, Machine Learning: An Algorithmic Perspective, Second Edition, second ed., Chapman & Hall/CRC, 2014.

[46] S. Haykin, Neural Networks: A Comprehensive Foundation, second ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[47] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? - More accurate information retrieval-based bug localization based on bug reports, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 14–24.

[48] L. Breiman, Random Forests, Mach. Learn. 45 (1) (2001) 5–32.

[49] Y. Tian, N. Ali, D. Lo, A.E. Hassan, On the unreliability of bug severity data, Empir. Softw. Engg. 21 (6) (2016) 2298–2323.

[50] Y. Zhao, Y. Cen, Data Mining Applications with R, first ed., Academic Press, 2013.

[51] M. Kuhn, K. Johnson, Applied Predictive Modeling, in: SpringerLink : Bücher, Springer New York, 2013.

[52] G. Luo, A review of automatic selection methods for machine learning algorithms and hyperparameter values, Netw. Model. Anal. Health Inform. Bioinform. 5 (1) (2016) 18.

[53] P. Probst, B. Bischl, A.-L. Boulesteix, Tunability: Importance of Hyperparameters of Machine Learning Algorithms, 2018, arXiv e-prints, arXiv:1802.09596. arXiv:1802.09596 [stat.ML].

[54] R. Feldman, J. Sanger, Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data, Cambridge University Press, New York, NY, USA, 2006.

[55] G. Williams, Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery, Springer, 2011, p. 374.

[56] A. Srivastava, M. Sahami, Text Mining: Classification, Clustering, and Applications, first ed., Chapman and Hall/CRC, 2009.

[57] A. Torfi, R.A. Shirvani, Y. Keneshloo, N. Tavaf, E.A. Fox, Natural language processing advancements by deep learning: A survey, 2021.

[58] S. Landolt, T. Wambsganss, M. Söllner, A Taxonomy for Deep Learning in Natural Language Processing, Hawaii International Conference on System Sciences, 2021.

[59] S. Ravichandiran, Getting Started with Google BERT: Build and Train State-of-the-Art Natural Language Processing Models using BERT, Packt Publishing, 2021.

[60] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, R. Soricut, ALBERT: A lite BERT for self-supervised learning of language representations, 2019, http://dx.doi.org/10.48550/ARXIV.1909.11942, Available from: https://arxiv.org/abs/1909.11942.

[61] I. Turc, M.-W. Chang, K. Lee, K. Toutanova, Well-read students learn better: On the importance of pre-training compact models, 2019, arXiv Preprint arXiv:1908.08962v2.

[62] V. Sanh, L. Debut, J. Chaumond, T. Wolf, DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, 2019, http://dx.doi.org/10.48550/ARXIV.1910.01108, arXiv. Available from: https://arxiv.org/abs/1910.01108.

[63] K. Clark, M.-T. Luong, Q.V. Le, C.D. Manning, ELECTRA: Pre-training text encoders as discriminators rather than generators, in: International Conference on Learning Representations, 2020, Available from: https://openreview.net/forum?id=r1xMH1BtvB.

[64] A. Lamkanfi, S. Demeyer, Filtering bug reports for fix-time analysis, in: 2012 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 379–384.

[65] M. Habayeb, S.S. Murtaza, A. Miranskyy, A.B. Bener, On the use of hidden Markov model to predict the time to fix bugs, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, p. 700.

[66] Y. Tian, D. Lo, C. Sun, Information retrieval based nearest neighbor classification for fine-grained bug severity prediction, in: 2012 19th Working Conference on Reverse Engineering, 2012, pp. 215–224.

[67] H. Valdivia Garcia, E. Shihab, Characterizing and predicting blocking bugs in open source projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, in: MSR 2014, ACM, New York, NY, USA, 2014, pp. 72–81.

[68] E. de Jonge, M. van der Loo, An introduction to data cleaning with R, Statist. Netherl. (2013) 53.

[69] N. Japkowicz, M. Shah, Evaluating Learning Algorithms: A Classification Perspective, Cambridge University Press, New York, NY, USA, 2011.

[70] F. Wilcoxon, Individual Comparisons by Ranking Methods, Springer New York, New York, NY, 1992, pp. 196–202.

[71] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 2016, arXiv preprint arXiv:1609.02907.

[72] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, M. Sun, Graph neural networks: A review of methods and applications, 2018, CoRR, abs/1812.08434. Available from: http://arxiv.org/abs/1812.08434.

[73] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P.S. Yu, A comprehensive survey on graph neural networks, 2019, CoRR, abs/1901.00596. Available from: http://arxiv.org/abs/1901.00596.