An Empirical Study on the Potential of Word Embedding Techniques in Bug Report Management Tasks

Bingting Chen [1] · Weiqin Zou* [1] · Biyu Cai[1] ·

Qianshuang Meng[1] · Wenjie Liu[1] ·

Piji Li[1] · Lin Chen[2]

Abstract Context Representing the textual semantics of bug reports is a key component of bug report management (BRM) techniques. Existing studies mainly use classical information retrieval-based (IR-based) approaches, such as the vector space model (VSM) to do semantic extraction. Little attention is paid to exploring whether word embedding (WE) models from the natural language process could help BRM tasks.
Objective To have a general view of the potential of word embedding models in representing the semantics of bug reports and attempt to provide some actionable guidelines in using semantic retrieval models for BRM tasks.
Method We studied the efficacy of five widely recognized WE models for six BRM tasks on 20 widely-used products from the Eclipse and Mozilla foundations. Specifically, we first explored the suitable machine learning techniques under the use of WE models and the suitable WE model for BRM tasks. Then we studied whether WE models performed better than classical VSM. Last, we investigated whether WE models fine-tuned with bug reports outperformed general pre-trained WE models.
Key Results The Random Forest (RF) classifier outperformed other typical classifiers under the use of different WE models in semantic extraction. We rarely observed statistically significant performance differences among five WE models in five BRM classification tasks, but we found that small-dimensional WE models performed better than larger ones in the duplicate bug report detection task. Among three BRM tasks (i.e., bug severity prediction, reopened bug prediction, and duplicate bug report detection) that showed statistically

---

[1]Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[2]Department of Computer Science and Technology, Nanjing University, Nanjing, China
E-mail: btchen@nuaa.edu.cn, weiqin@nuaa.edu.cn, caibiyu@nuaa.edu.cn,
qs_meng@nuaa.edu.cn, wenwenmu@nuaa.edu.cn, pjli@nuaa.edu.cn, lchen@nju.edu.cn
Weiqin Zou is the corresponding author.

significant performance differences, VSM outperformed the studied WE models. We did not find performance improvement after we fine-tuned general pre-trained BERT with bug report data.

Conclusion Performance improvements of using pre-trained WE models were not observed in studied BRM tasks. The combination of RF and traditional VSM was found to achieve the best performance in various BRM tasks.

Keywords Bug report · Word embedding · Pre-trained models · Vector space model

1 Introduction

Bug reports are an important kind of software artifact. They generally provide information about the behavior of software and bug-occurring environments (Bettenburg et al., 2008). With bug reports, software practitioners could better understand and fix a bug and identify potential software-quality improvement directions (Bertram et al., 2010). Currently, many software projects use a bug tracking system such as Bugzilla[1] or Jira[2] to collect and track their bug reports, where any end users are free to submit a bug report to describe what problems they are facing (Anvik, 2006). The openness of receiving bug reports and the increasing software scale make software practitioners have to handle a large number of bug reports (Zou et al., 2018). How to effectively and efficiently manage as many bug reports as possible under limited resources is increasingly becoming a big challenge for modern software projects.

To help address the problem mentioned above, researchers have been developing various kinds of bug report management (BRM) techniques, such as bug localization (Kim et al., 2013; Ye et al., 2015), duplicate bug detection (Wang et al., 2008; Sun et al., 2011), bug priority prediction (Tian et al., 2013, 2015), bug fixing time prediction (Weiss et al., 2007; Choetkiertikul et al., 2017), etc. A key step of BRM techniques is to correctly retrieve the semantics of bug reports as much as possible since it is the basis for further prediction. A bug report's two most important items are the brief one-line summary of a bug and the detailed problem description that provides important information such as reproduction steps and observed/expected behavior (Lamkanfi et al., 2010). These two items are generally written in plain text and are widely considered in existing BRM research. To a large extent, the semantic retrieval of a bug report is taken as the semantic retrieval of textual bug summary and description items.

Currently, existing BRM studies mainly use some information retrieval models to represent the semantics of bug reports, where the vector space model (VSM) is mainly used. These models, like VSM, generally take bug reports as a bag of words, considering little about the contextual semantics of words,

---

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

[2] https://www.atlassian.com/software/jira

which may negatively affect the BRM performance. With the fast development of NLP techniques, more advanced semantic retrieval approaches from the NLP area have appeared. Among them, word embedding models, such as Word2Vec (Mikolov et al., 2013a) and BERT (Devlin et al., 2018), attract much attention from both the academic and industrial community (Yang et al., 2016; Xiao et al., 2019). Word embedding models represent the semantics of a word with an N-dimensional numeric vector, with each element capturing a specific semantic aspect of the word in the world. They are expected to capture the contextual semantics of words better and have been proven effective in many NLP downstream tasks (such as text classification and question answering) relying on textual semantic retrieval. Inspired by the applications of WE models in the NLP area, some researchers also try to integrate WE models such as Word2Vec and GloVe into BRM techniques (Van Nguyen et al., 2017; Deshmukh et al., 2017; Jia et al., 2021).

While word embedding has become more prevalent in NLP, its adoption in BRM tasks remains relatively limited. Most existing studies have focused on earlier WE models like Word2Vec, even though the NLP community has since introduced a range of more advanced models, such as BERT. The success of WE in NLP tasks encourages us to study the potential of WE in BRM tasks systematically. Our exploration in this study is motivated by two key observations. On one hand, bug reports can be taken as general text in the same form as NLP tasks. It provides the basis for applying WE models to bug reports. On the other hand, unlike general text, bug reports are software artifacts that have domain-specific knowledge. How well WE models can capture the semantics of bug reports is still a question mark.

We attempt to answer four research questions to understand the value of WE in BRM tasks. Five widely recognized WE models are thoroughly tested on six typical BRM tasks using a dataset of 20 products from different domains to get as general conclusions as possible. Word2Vec, GloVe, FastText, ELMo, and BERT are the five WE models. Six BRM tasks include duplicate bug report detection, bug severity prediction, bug priority prediction, bug fixing time prediction, reopened bug prediction, and bug field reassignment prediction. The concrete RQs, as well as their corresponding key findings, are as follows.

RQ1. Which machine learning approach performs best for BRM tasks on the whole? This research question is about finding the most suitable machine learning (ML) algorithm for BRM tasks under the constraint of applying WE models to retrieve the semantics of bug reports. Four classical machine learning algorithms are compared in the study, namely Naive Bayes, Random Forest, Logistic Regression, and Support Vector Machine.

Findings: The Random Forest classifier always outperformed other machine learning algorithms based on retrieved semantic features of five pretrained WE models. Naive Bayes performed worst among all classifiers.

RQ2. Is there a word embedding model that generally outperforms others for BRM tasks? This research question is to study whether there is a dominant WE model that outperforms other WE models in various

BRM tasks. This RQ could show some insights into selecting the WE model in performing BRM tasks.

Findings: We rarely observed statistically significant performance differences among five WE models in five BRM classification tasks (i.e., bug severity prediction, bug priority prediction, bug fixing time prediction, reopened bug prediction, and bug field reassignment prediction); we only detected a statistically significant performance difference between smaller-dimensional WE models and relatively larger ones for the duplicate bug report detection task (a recommendation task).

RQ3. Do word embedding models outperform the most used VSM model for BRM tasks? This RQ could help us understand whether WE models are preferred to be applied in domain-specific BRM tasks rather than general NLP tasks. This gives practitioners better guidance in adopting traditional VSM or WE models in their BRM tasks.

Findings: Among three BRM tasks, i.e., bug severity prediction, reopened bug prediction, and duplicate bug report detection, that showed statistically significant performance differences, VSM outperformed studied WE models; the performance differences are found to be negligible to small on the bug severity prediction and reopened bug prediction tasks, but are large on the duplicate bug report detection task, according to Cliff's delta effect sizes.

RQ4. How will the performance change if word embedding models are fine-tuned with bug data? This RQ investigates the performance of fine-tuned BERT with bug reports compared to the generic pre-trained BERT model. Answering this RQ could help us understand the value of domain-specific corpus for pre-trained WE models in semantic extraction.

Findings: We did not observe performance improvements of the fine-tuned BERT over the generic pre-trained one. Their performance differences that show statistical significance are all negligible according to Cliff's delta effect sizes on experimental products.

The main contributions of our work are as follows:

– We conduct a systematic study to explore the potential of word embedding models on bug report management tasks by evaluating five typical embedding techniques across six representative downstream BRM tasks.
– We attempt to investigate the most suitable machine learning algorithm for applying WE models, the best-performance WE model in BRM tasks, the advantage of WE over the most used VSM model, and the value of the model fine-tuned with bug report corpus.
– We obtain a list of actionable findings that could guide developers or researchers in adopting suitable semantic retrieval models while performing BRM tasks.

Paper Organization The remaining parts are structured as follows: we present the background of WE and BRM in Section 2 and describe our experimental design in Section 3. Section 4 presents our experimental results and answers to research questions. Section 5 lists the implications and the threats

to the validity of our experiment. After surveying the related work in Section 6, we conclude our work in Section 7.

2 Background

In this section, we discuss the background related to this study. We first introduce word embedding(WE), including five typical word embedding models and pre-trained models. Then, we talk about bug report management(BRM).

2.1 Word Embedding

Text representation is an important work in the field of natural language processing (NLP). Effectively converting text data into a meaningful representation that computers can recognize is a very important step in the NLP area. The word embedding technique is such a popular and effective method for presenting the semantics of text contents. This technique maps a single word to a δ-dimensional vector space (Hinton et al., 1986). It aims to capture the latent semantics of words by using neural network-based word embedding vectors that are dense, distributed, and with a fixed length. Textual content within bug reports can also be transformed by word embedding technology into numeric embedding vectors. In this section, we describe the existing popular and representative embedding techniques evaluated in this work, i.e., Word2Vec, GloVe, FastText, ELMo, and BERT.

2.1.1 Five Typical Word Embedding Models

Word2Vec is a language model published in 2013 (Mikolov et al., 2013a). It takes a large text corpus as input and generates a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word2Vec has two model architectures: continuous bag-of-words (CBOW) and continuous skip-gram. In the CBOW architecture, the model predicts the masked word by a given window of its surrounding context words, and the order of context words does not influence the prediction results (due to the bag-of-words assumption). In the continuous skip-gram architecture, the model uses a given word to predict the context words surrounding the word. Unlike CBOW, the skip-gram architecture weighs nearby context words more heavily than more distant context words (Mikolov et al., 2013b). Regardless of CBOW or skip-gram, Word2Vec would assign a unique numeric vector to a word, limiting it in handling the polysemy problem.

    GloVe (Global Vectors for Word Representation) is proposed by Pennington et al. (Pennington et al., 2014) in 2014. It is a global log-bilinear regression model for unsupervised learning of word representations. Compared to Word2Vec, GloVe aims to use a global word-word co-occurrence matrix

to learn word representation, which is expected to better represent a word's meaning. It is reported to outperform other models on word analogy, word similarity, and named entity recognition tasks.

FastText is proposed by Bojanowski et al. (Bojanowski et al., 2017) in 2017. Unlike models like Word2Vec or GloVe which treat a word as an atom during model training, FastText treats a word as a set of n-gram units. Based on these n-gram sets, a skip-gram model is built, and an embedding vector for each n-gram is obtained. The vectors of n-gram sets of a word are then integrated to obtain the final embedding vector for the word. In this way, FastText can capture the semantics of short/rarely appeared words and those pre/suffixes.

ELMo (Embeddings from Language Models) is proposed by Peters et al.(Peters et al., 2018) in 2018. The ELMo embedding vectors are derived from a bi-directional LSTM trained with a coupled language model on a large text corpus[3]. In ELMo, character-level tokens are taken as the input to a bi-directional LSTM that would then produce word-level embeddings. It is a new type of deep contextualized word representation that models both (1) complex characteristics of word use (e.g., syntax and semantics), and (2) how these uses vary across linguistic contexts. Unlike "Bag of Words" approaches such as Word2Vec, ELMo is context-sensitive and could generate different representations for words with the same spelling but different meanings.

BERT (Bidirectional Encoder Representations from Transformers) is proposed by Devlin et al. (Devlin et al., 2018) in 2018. BERT is based on the transformer architecture and is pre-trained on a large corpus of unlabelled text, including the entire Wikipedia (with 2,500 million words) and Book Corpus (with 800 million words). BERT is a deeply bidirectional model. Bidirectional means that BERT learns information from both the left and the right side of a token's context during training. The word embedding representation obtained through BERT incorporates more grammatical, lexical, and semantic information. With BERT, words would have different word embedding representations in different contexts.

2.1.2 Pre-trained Word Embedding Models

As mentioned above, in the NLP area, word embedding is generally an n-dimensional numeric vector that captures the semantics of a word, with similar words that are closer in the vector space being expected to have similar meanings. Generally speaking, one can obtain the word embedding for a word with two approaches. One approach is to train a word embedding model (e.g., Word2Vec) on their dataset from scratch to obtain the word embeddings, while the other way is to obtain the word embeddings from an existing word embedding model trained on other large datasets, i.e., a pre-trained word embedding model.

---

[3] For example, the 1 billion word benchmark: https://www.statmt.org/lm-benchmark/

Due to the sparsity of training data and a large number of trainable parameters, it becomes a challenging problem for users to learn their own embeddings from scratch, especially when they have limited small-scale datasets. In this case, pre-trained models would be a preferred choice for them due to the following advantages: 1) Pre-training on large-scale unlabeled data can learn more general language representation and apply it to downstream tasks; 2) Pre-training provides better initialization parameters, which helps to improve the generalization performance of the model and accelerate the convergence of the model; 3) Pre-training can be regarded as an effective regularization method to avoid over-fitting on small data sets. Currently, there is a list of pre-trained WE models kindly shared by the academics and industry, including the five typical word embedding models we compared in the study, such as the pre-trained BERT through training BERT on a large amount of Wikipedia content. Sometimes, practitioners would use domain-specific data to fine-tune the pre-trained models to understand their tasks better. The advantages of pre-trained models have made them find numerous important applications in the real world (Yang et al., 2019; Zhou et al., 2020).

2.2 Bug Report Management

In this part, we will first briefly introduce typical bug reports collected by bug tracking systems of Eclipse and Mozilla. Then we introduce six typical bug report management techniques we focused on in this paper.

2.2.1 Bug Report

Bug reports are an important information carrier for software practitioners to find/fix bugs and ensure software quality. Figure 1 shows a typical bug report collected by bug tracking systems. As shown in the figure, a bug report generally has information items like *BugID, Status, Product, Component, Importance, Reported, Summary, Description, Comment, etc. BugID* (e.g., 550200 in the figure) is a number that uniquely identifies a bug of a project. *Summary* is generally one sentence briefly summarizing a bug. The *Description* item provides details about the bug, where the details generally include steps to reproduce the bug, the expected behavior or observed behavior, etc. *Status* describes the current state of a bug report. During the lifecycle of a bug report, its status could be from Unconfirmed to Closed/Resolved as shown in Figure 2, each closed bug report would have a corresponding resolution such as FIXED, DUPLICATE, etc. Detailed introduction to the status and resolutions can be found in the website [4]. The *Reported* field includes the reporting time of a bug and the person who reports it. The *Importance* field has two parts, namely priority (e.g., "P3" in the figure) and severity (e.g., "normal" in the figure). The modifications towards a bug report, such as status changes

---

[4] https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Status/RESOLVED

would all be recorded in the "History" item, and the *Modified* item memos the last modification time.

Fig. 1: A Bug report example with ID=550200 in Eclipse

2.2.2 Six BRM Techniques

To help resolve as many bug reports as possible with limited resources, researchers proposed a list of techniques to manage these bug reports, namely bug report management techniques (Zou et al., 2018), including bug localization, bug severity/priority prediction, bug fixing time prediction, etc.

This section mainly introduces six bug report management techniques studied in our work, also considered important by software practitioners(Huo et al., 2014; Zou et al., 2018). They are duplicate bug report detection, bug severity prediction, bug priority prediction, bug fixing time prediction, bug field reassignment prediction, and re-opened bug prediction. These techniques can help us more purely understand the usefulness of WE models in representing bug report semantics. This is because they mainly rely on the analysis of textual summary and description of a bug report. In other words, we can avoid the situation that our findings get biased by the potential interactions between
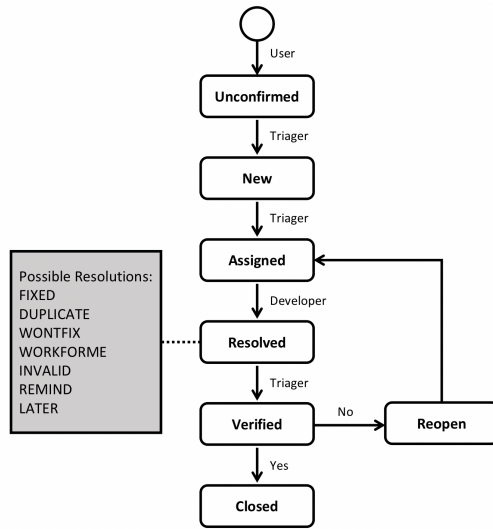
Fig. 2: The life-cycle of a bug report

bug reports and other software artifacts, as well as imprecise processing steps such as bug-code linking in other BRM techniques.

Duplicate Bug Report Detection aims to identify whether or not a newly reported bug is duplicated with existing bugs in the bug tracking system. With duplicate bug detection, users could avoid reporting the same problems, while developers could save time in handling more bug reports without wasting time doing redundant work. In a bug tracking system, a bug report with a "DUPLICATE" resolution represents that the bug is duplicated with an existing bug(s). Existing studies mainly take duplicate bug report detection as a recommendation task (Runeson et al., 2007; Wang et al., 2008; Sun et al., 2011; Zhang et al., 2023). The process is to use typical VSM to represent the semantics of the summary and description items first, then rank bug reports based on the cosine similarity of VSM vectors where a word is mainly weighted by TF-IDF (Sun et al., 2010), BM25F (Sun et al., 2011), etc. Some studies attempt to take this task as a classification one by applying machine learning algorithms like SVM, LSTM, and CNN to predict whether a bug report is duplicated with another one or not (Sun et al., 2010; Budhiraja et al., 2018a; Rodrigues et al., 2020).

Bug Severity Prediction aims to predict the severity of a bug report. The severity of a bug report reveals how severe the bug is in the software. Some bugs are particularly critical such as data corruption and need to be fixed immediately; while some may be minor problems and can be delayed until resources are available. Thus, bug severity prediction can help stakeholders better arrange resources in fixing bugs and ensure that software works properly as much as possible. Existing research usually regards bug severity prediction

as a classification problem. Among those studies, the summary and description items are mostly used, items like stack trace, components, priorities, product, and code change history are sometimes used to improve prediction performance(Menzies and Marcus, 2008; Yang et al., 2014; Roy and Rossi, 2014; Sahin and Tosun, 2019; Ramay et al., 2019; Arokiam and Bradbury, 2020; Kim and Yang, 2022). The VSM with TF/TF-IDF weighting is commonly adopted to represent the textual semantics of bug reports; other models like LDA or Word2Vec are also used in a few studies. Regarding the selection of ML models, most studies tend to use traditional classification algorithms like SVM, NB, and RF; few studies use deep learning methods such as MLP and LSTM to make severity predictions.

Bug Priority Prediction aims to predict the priority of a bug report, which could save practitioners from the time-consuming and error-prone manual priority assignment activity. Bug reports with high priority, such as those impacting a large number of consumers should be fixed first. With automated priority prediction, software maintainers could have a better mind about collected bug reports before manually diving into the bugs and make a better resource allocation in resolving them to avoid letting high-priority bugs go without notice among large amounts of bugs. For bug priority prediction, mainstream research also regards it as a classification task. Besides the summary and description features, other features like historical similar bug reports, reporter information, and the component/product a bug report belonged to, would also be integrated to facilitate prediction performance(Kanwal and Maqbool, 2012; Tian et al., 2013, 2015; Umer et al., 2019; Zhang and Challis, 2020). The VSM is mostly used in the semantic retrieval of textual content; the LDA and word embedding models are also used in some cases. As for the adoption of ML methods, it shows a route from using traditional ML models like SVM, NB, and KNN to (deep) neural networks like CNN.

Bug Fixing Time Prediction aims to predict how long a bug report would be fixed. For a large and evolving software system, a project team may receive many bug reports over a long period. It would be very valuable to achieve a quantitative understanding of bug-fixing time for those bugs, in that not only users could know possible feedback or status of their reported bugs but also project maintainers could make a better resource schedule for fixing bugs. Towards the bug-fixing time prediction task, some researchers choose to estimate the total time required to fix a bug(Panjer, 2007; Weiss et al., 2007; Hewett and Kijsanayothin, 2009; Vieira et al., 2022) while some researchers try to predict whether it would be a slow or fast fix(Giger et al., 2010; Marks et al., 2011; Zhang et al., 2013; Habayeb et al., 2017; Yuan et al., 2021). The items used in existing studies mainly include the summary and description, bug severity/priority, reporter popularity, open bugs, developer activities, field updates, etc. Prediction models also mainly focus on traditional ML methods such as Logistic Regression, Random Forest, and Hidden Markov Model, with some recent work also applying neural networks such as MLP(Yuan et al., 2021; Vieira et al., 2022).

Re-opened Bug Prediction Re-opened bugs are bugs closed by developers but re-opened later (i.e., status being "Reopen" in the lifecycle of a bug in Fig. 2). Bugs can be re-opened for a variety of reasons such as partial fixing. The existence of re-opened bugs makes developers generally have to take a longer time to resolve them and hence further increase the maintenance costs of a project. Thus, a series of studies have emerged to automatically predict whether a bug report would be re-opened during its lifecycle(Shihab et al., 2010; Zimmermann et al., 2012; Shihab et al., 2013; Xia et al., 2013, 2015b; Mi et al., 2018; Tagra et al., 2022). These studies generally take re-opened bug prediction as a classification task, by applying traditional supervised machine learning models like NB, DT, LR, and Bagging towards bug instances represented by a set of features such as textual bug description, reporter/fixer reputation, work habit, etc.

Bug Report Field Reassignment Prediction A bug report contains many fields, such as product, component, severity, priority, and so on. Those important information items can help developers a lot in bug fixer assignments and bug fixing. However, as reported by (Xia et al., 2014), reporters often provide wrong values for bug report fields, which prevents developers from effectively fixing bugs. Bug report field reassignment prediction aims to predict which fields would have their values reassigned so that developers or reporters could better cope with their bug reports, e.g., by trying to correct them timely. Bug report field reassignment is also generally taken as a classification task. In this field, ML algorithms like NB, KNN, and HMM are generally tested; the textual summary and description items are commonly used to build prediction models, function call sequences from stack traces and some categorical features like operating systems are also considered in specific studies (Lamkanfi and Demeyer, 2013; Xia et al., 2015a; Islam et al., 2021).

3 Experiment Setup

In this section, we will first briefly present the overall workflow of our experiments. Then we introduce the datasets we use to conduct our experiments. After that, we describe how we use WE models to retrieve bug report semantics and build prediction/recommendation models to perform BRM tasks. Last we present the four research questions whose answers may help us understand the potential of WE models in BRM tasks.

3.1 Overflow

Figure 3 shows the whole experiment process of our study. First of all, we separately crawled all bug reports from the bug tracking systems (BTS) of Eclipse and Mozilla by the time we conducted our study. For each bug report, we extract the summary and description items from bug reports for experiments. Then, we use pre-trained word embedding models to extract semantics from the textual content of summary and description items.
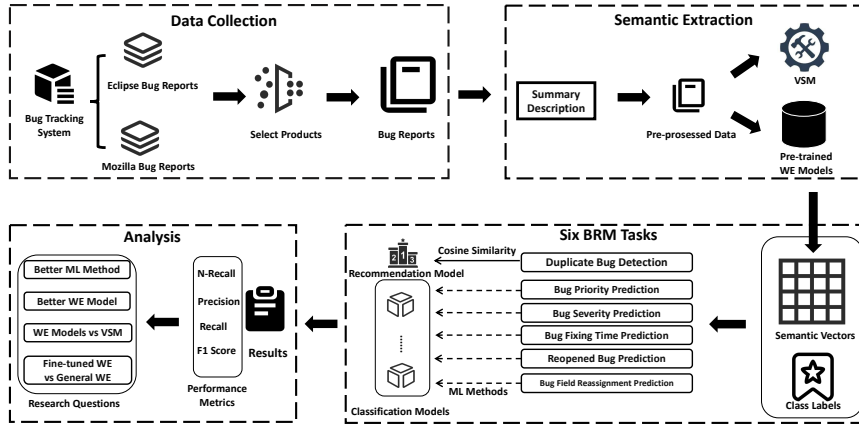
Fig. 3: Overall framework of our study

After applying WE models, each word of a bug report would be represented as a numeric vector with N dimensions (e.g., general pre-trained Word2Vec would generate a 300-dimensional numeric vector for a word), and a final N-dimensional vector would be generated for a bug report after operations such as max/average-pooling. The final N-dimensional vector represents the semantic features of a bug report.

Based on these embedding vectors, six models for six downstream BRM tasks are built. These models are then tested on the new bug data. The prediction results are further analyzed to answer four research questions that explore the most suitable ML for WE models, the best WE models, the comparison between WE models and the VSM model, and the value of fine-tuning general pre-trained WE models for BRM tasks.

As shown in the figure, when we build corresponding BRM models, we only consider textual semantic features of summary and description items without leveraging other helpful factors such as developer experience. We adopt this strategy mainly because: 1) The two textual items are most important for a bug report and play a major role in existing BRM techniques; 2) The goal of our study is to understand the potential of WE in representing bug report semantics for various BRM tasks not to propose most-advanced BRM techniques; 3) Considering other factors may introduce confounding factors that prevent us from obtaining unbiased conclusions related to the value of using WE in BRM tasks; 4) Since we use performance difference to answer our research questions, the possibly even-not-so-high performance values would not harm our analysis.

3.2 Data Collection

We conduct our experiments on data from two famous open-source foundations, Eclipse and Mozilla. For each foundation, we first crawl all bug reports from its bug tracking system (BTS). The BTS supports several bug data formats, including XML, CSV, JSON, etc. In this study, we downloaded the corresponding XML files of those bug reports. Each XML file represents a whole bug report, and a tag can identify each information item of the bug report. For example, the tags indicated the one-line summary and description items are "short_desc" and "long_desc" for Eclipse and Mozilla. We also use these tags to retrieve the needed information items.

After obtaining all bug reports for each foundation, we select the top 10 products with the largest number of bug reports. The selected 20 products are popular among developers and users. Some are often used in existing studies(Lamkanfi et al., 2010; Nguyen et al., 2012; Tian et al., 2013; Zhang et al., 2016). They come from different domains and are of different sizes, which to some extent guarantee the generalizability of our findings in practice. We would first retrieve each product's bug reports and prepare corresponding datasets for different BRM tasks. Specifically, we would filter out not-yet-closed bug reports for bug severity/priority/reassignment prediction tasks since we can not determine their final status or resolutions. For the bug fixing time prediction task, we only consider the bug reports whose resolution is "fixed". As for the reopened bug prediction task, besides closed ones, bug reports whose status is "reopened" are also included during experiments. Table 1 shows the basic statistics of our experimental products.

After obtaining the text of the one-line summary and description items for each bug report, we take the extracted content as plain text and follow a common flow to preprocess it. That is, we first tokenize the text into words. At the same time, special symbols, digits, and single characters are also removed. We further split camel case words into individual words (e.g., windowWidth → window + width), with original camel case words being also kept. After that, we convert each word to its lowercase and then use Porter Stemmer to do stemming by transferring individual words into their root forms (e.g., reported → report). At last, we use the NLTK stop-word list[5] to remove stopwords (such as "the", "and", and "a") that frequently appeared in a corpus and generally contribute little to one's understanding of text content. Like most BRM studies (Roy and Rossi, 2014; Xia et al., 2015b,a), we did not further handle the hyperlinks and stack traces after applying the above preprocessing steps towards them.

3.3 Semantic Extraction

We mainly use five typical pre-trained word embedding models (i.e., Word2Vec, FastText, GloVe, ELMo, and BERT) to extract the textual semantics of bug

---

[5] https://www.nltk.org/search.html?q=stopwords

Table 1: Basic Statistics of Our Experimental Products

Project	Product	BRs	Closed BRs	Fixed BRs	Duplicate Pairs	Domain	Time Period
Eclipse	CDT	22,209	17,352	12,367	3,396	C/C++ Integrated Development Environment	Oct 2001 - Jan 2022
	Community	24,597	24,416	18,738	1,858	Community Service	
	Equinox	14,490	14,046	8,039	2,355	Service Delivery/Management Architecture	
	JDT	62,927	57,894	28,466	23,810	Java Development Tools	
	Mylyn	8,864	7,929	5,223	1,956	User Interface	
	Orion	10,470	10,310	7,136	625	Web IDE	
	Papyrus	12,742	9,990	8,472	543	Modeling Framework	
	PDE	17,518	16,848	9,722	3,466	Plug-in Development Environment	
	Platform	121,939	114,782	53,527	12,945	Software Development Infrastructure	
	z_Archived	85,072	77,258	51,803	24,768	Bug Archive	
Mozilla	Core	465,252	404,058	206,914	14,222	Web Content Handling Utilities	Apr 1998 - Feb 2022
	DevTools	38,387	30,802	17,053	9,512	Web Developer Tools	
	Firefox	196,618	178,129	39,978	17,617	Web Browser	
	Firefox for Android Graveyard	39,151	39,146	15,499	13,080	Mobile Browser	
	Firefox OS Graveyard	70,532	70,532	29,617	16,281	Operating System	
	MailNews Core	34,090	29,909	10,904	21,457	MailNews Infrastructure	
	SeaMonkey	96,208	92,844	21,168	215,171	Internet Suite	
	Testing	34,484	30,654	18,858	8,861	Testing Tools/Frameworks	
	Thunderbird	60,535	52,517	12,091	59,377	Email Client	
	Toolkit	55,540	47,527	21,397	34,880	Programming Interfaces	

reports. For each word, a WE model would generate a N-dimensional numeric vector for it. For a bug report of a list of words, we take each dimension's average value from vectors for all words it contains (i.e., average pooling) and generate a final N-dimensional vector. This final vector is taken as the semantic representation of the bug. We directly use the shared five pre-trained WE models from their official websites[6].

All WE models except Word2Vec (N=300) provide several vector-dimension options. For example, users can download pre-trained GloVe models that generate 50/100/200/300-dimensional vectors for words. The dimensional options of BERT and ELMo are 768/1024 and 1024/2048/4096 respectively. FastText provides a 300-dimensional model but also provides a tool to transform it to a customized size. Based on our preliminary exploration, we find three options of FastText, i.e., 100, 200, and 300, are mostly used in previous studies. As we have no idea in advance whether the dimension size would affect the findings or not, we decide to test all those options.

Furthermore, considering that larger feature numbers generally require a much larger dataset to well approximate their weights during model building, for BERT and ELMo, we only test the smallest pre-trained model, i.e., 768 for BERT and 1024 for ELMo. Based on our investigation, larger BERT/ELMo pre-trained models are not so commonly used in existing software engineering studies. Finally, the dimensions we use for numeric vectors generated by BERT, ELMo, FastText, Word2Vec, and GloVe are 768, 1024, 100/200/300, 300, 50/100/200/300 respectively.

Meanwhile, since one of our goals is to compare the performance of WE models to mostly used VSM in existing BRM studies, we also use the VSM model to represent the semantics of bug reports. VSM (Vector Space Model) is a typical information retrieval model for text. It treats a bug report as a bag of words and generates a bug vector. The vector size is the number of

---

[6] https://code.google.com/archive/p/word2vec/
https://nlp.stanford.edu/projects/glove/
https://fasttext.cc/docs/en/english-vectors.html
https://huggingface.co/bert-base-uncased
https://allenai.org/allennlp/software/elmo

unique words in a corpus (all bug reports), with each element corresponding to a unique word, and the value of an element is the weight of a term in a document (bug report). Following existing studies (Zhou et al., 2012; Saha et al., 2013; Ye et al., 2014), we use TF-IDF to measure the term weight.

3.4 Model Building and Evaluation for Six BRM Tasks

3.4.1 Model Building

After obtaining semantic feature vectors of individual bug reports, our next step is to train models for six BRM tasks, namely duplicate bug report detection, bug severity prediction, bug priority prediction, bug fixing time prediction, re-opened bug prediction, and bug field reassignment prediction. Following the mainstream BRM studies (Jalbert and Weimer, 2008; Lamkanfi et al., 2010; Giger et al., 2010; Nguyen et al., 2012; Deshmukh et al., 2017), we take duplicate bug report detection as a recommendation task and the other five as classification tasks. In other words, duplicate bug report detection aims to recommend a list of potential duplicate bug reports for a given bug report, where the recommendation is mainly based on the cosine similarity between two semantic feature vectors. The remaining five tasks aim to classify a bug report into pre-defined categories, where a prediction model generally needs to be built by applying a machine learning (ML) algorithm to a dataset of labeled bug reports represented in semantic feature vectors. In this study, we test four typical machine learning algorithms, namely NB (Naive Bayes), SVM (Support Vector Machine), RF (Random Forest), and LR (Logistic Regression) in model building. These algorithms are classical machine learning approaches for classification tasks and are often used in BRM studies (Lamkanfi et al., 2010; Tian et al., 2013, 2015; Zhang et al., 2016). The combination of different WE/VSM models and ML algorithms could help us explore the suitable ML or semantic feature extraction models in the study.

To build or evaluate a model, we need to have a benchmark dataset for the six BRM tasks. The previous subsection has helped us obtain semantic feature vectors for individual bug reports, and we need to label each bug report further to have a ground truth (with both instance features and labels determined) for model training and prediction. The detailed labeling strategies of six BRM tasks are as follows.

Duplicate Bug Report Detection. We mainly rely on the resolution field to find those duplicate bug pairs. Specifically, for a bug report, if it is a duplicate one with another bug report, the resolution field of this bug report would be set as "duplicate", with the corresponding dupId also being attached. dupId indicates which bug report the current bug report is duplicated with. For example, for a bug with id 1234, suppose its resolution is "duplicate" and its dupId is 5678, then we can understand that bug 1234 is a duplicate of bug 5678. The bug 5678 is also called the master bug report. We can identify whether two bug reports are duplicates by referring to the "duplicate" resolution and

dupId. We can find all possible duplicate pairs by collecting all such pairs and constituting a directed diagram. Those pairs would work as the ground truth when we evaluate the built duplicate bug detection model.

Bug Severity Prediction. According to the configuration of bug tracking systems adopted by Eclipse and Mozilla, a bug report generally has a severity field, whose values are among Blocker, Critical, Major, Minor, Trivial, and Enhancement; sometimes, Mozilla would also use S1, S2, S3 and S4 to indicate severity [7]. Following (Lamkanfi et al., 2010, 2011; Tian et al., 2015), we classify bug reports into two categories, i.e., Severe or Non-severe. Bug reports with severity levels of Blocker, Critical, and Major, or S1/S2 are put into the Severe category. Bug reports with severity levels of Minor and Trivial, or S4 are considered Non-severe. Bug reports with the Enhancement level or S3 are ignored because these reports are requests for new functionalities.

Bug Priority Prediction. Similar to bug severity, a bug report has a field to indicate its priority. The priority field has 5 levels, namely P1, P2, P3, P4, and P5, with P1 indicating the highest priority and P5 indicating the lowest priority. Reports whose priority is "–" are ignored in this task. Following (Alenezi and Banitaan, 2013; Izadi et al., 2022), we divide these five levels into three class labels namely *High, Medium, and Low*. High includes P1 and P2, Medium includes P3, and Low includes P4 and P5.

Bug Fixing Time Prediction. We take a two-step way to determine the fixing time label for a bug report with a "fixed" resolution inspired by the study of (Vieira et al., 2022). First, we calculate the time span from the creation time of a bug report to the time the bug was fixed. Then we classify all bug reports into three categories, i.e., < 5 days, 5-10 days, and > 10 days. As claimed in (Vieira et al., 2022), 5 days roughly represent a week of 5-weekdays and are considered as a more reasonable or practical time unit than other studies by, e.g., classifying bug reports into slow-fixing or fast-fixing categories.

Re-opened Bug Prediction. For each resolved bug report, we retrieve its history item and check its content to find whether there is a record reporting that the status of a bug report has ever been changed to "reopened". If there is, then the re-opened label of the bug report is 1, otherwise 0. The bug reports whose current status is "reopened" are directly labeled as 1.

Bug Report Field Reassignment Prediction. For each resolved bug report, we retrieve its history item and check whether the values of certain items have ever been changed to other values. Following (Xia et al., 2015a), we mainly check the field reassignment problem of meta-items including status, component, product, priority, severity, operating system(OS), and version. These items are important for bug locating/fixing and tend to have wrong values submitted by reporters. Note that unlike (Xia et al., 2015a), we ignored the reassignment to the assignee item (who or which team is responsible for fixing a bug) as it has little to do with reporters and is most likely to be changed during the lifecycle of a bug report. For each considered item, if its value has ever been modified, as shown in the history item, then the reassignment label is 1,

---

[7] http://wiki.mozilla.org/BMO/UserGuide/BugFields$bug_severity$

otherwise 0. Notice that for the status item, like (Xia et al., 2015a), we only consider the case where the status field is changed from resolved, removed, and reopened values to other values.

During model building for the five classification tasks, our project datasets present the class imbalance problem to different degrees, i.e., a class may have many more instances than others. Table 2 shows the instance numbers of each class for five BRM tasks. Such an imbalanced class problem may bias the prediction results. For example, the model may always predict an instance as one of the majority class. To avoid potential threats brought by imbalanced class data, we adopt the random oversampling strategy to balance the datasets in our study. Oversampling aims to balance uneven datasets by keeping all instances in the majority class and repeatedly sampling an instance from the minority class until the sizes of the two classes are the same. We use functions provided by Python package *imblearn* to do oversampling.

After handling the class imbalanced problem, we apply ML algorithms to the balanced datasets to obtain prediction models for BRM tasks. We use the functions provided by Python package *cuML*[8] to build those models. cuML package provides a series of GPU-accelerated machine learning algorithms, and its APIs mirror Scikit-learn's[9]. Only for those models that encountered the out-of-memory problem on a single NVIDIA GeForce RTX 3090 GPU (with 30G memory), would we train them with the CPU-based Scikit-learn ML APIs. All ML algorithms use the default settings provided by cuML or Scikit-learn. Using default settings improves the replicability of our study and what is more important, makes our findings more general and hence more referenceable in practice. This is because practitioners could not know in advance what optimal settings are for their BRM tasks at hand, in which case choosing a relatively better ML in general cases is very likely to be their first step in building their models. We adopt stratified ten-fold cross-validation during model building. That is, for each dataset, we first divide it into ten mutually exclusive subsets of approximately equal size, with each subset being testing data and the remaining nine subsets as training data. Both the training and testing data preserved the percentage of samples for each class. Then, for each pair of training and testing data (10 pairs in total), we perform the model training and prediction activities. Last, we take the averaged prediction results of ten rounds as the final performance of built BRM models. Note that, for the three quite large products, namely Platform, Core, and Firefox, we do not use all but the latest 50,000 closed/reopened bug reports for model training, which we think is sufficient to build a representative classifier but with much less time for model training. Moreover, we only do oversampling to the original training data of 9 folds. When we test the model, we use the original 1-fold testing data, which is still imbalanced. We think this is more reasonable than using balanced testing data to evaluate built models, as the testing data is imbalanced in real-world situations.

---

[8] http://docs.rapids.ai/api/cuml/stable/
[9] http://scikit-learn.org/stable/

Table 2: Numbers of bug reports for five BRM classification tasks

Project	Product	Fixing Time			Priority			Reassignment		Reopen		Severity	
		0-5 days	5-10 days	>10 days	High	Medium	Low	No	Yes	No	Yes	Severe	Non-Severe
Eclipse	CDT	2,519	710	9,138	390	16,900	62	11,162	6,190	16,327	1,103	2,269	848
	Community	8,077	1,409	9,252	607	23,588	221	21,030	3,386	22,128	2,303	2,986	423
	Equinox	2,756	633	4,650	184	13,785	77	9,849	4,197	13,275	813	1,784	482
	JDT	6,230	2,689	19,547	4,234	51,939	1,721	35,754	22,140	51,522	6,654	5,943	4,102
	Mylyn	1,747	478	2,998	1,955	5,319	655	4,001	3,928	7,193	748	734	1,060
	Orion	2,894	585	3,657	541	9,637	132	7,491	2,819	9,935	413	623	262
	Papyrus	2,201	444	5,827	307	9,666	17	7,163	2,827	9,667	379	1,493	360
	PDE	3,255	893	5,574	485	16,260	103	12,282	4,566	15,714	1,161	1,984	760
	Platform	6,355	2,047	15,654	702	48,553	434	33,548	16,141	46,209	3,791	6,904	2,900
	z_Archived	3,325	1,615	46,863	10,551	65,997	710	14,978	62,280	71,079	6,453	14,230	2,970
Mozilla	Core	8,464	5,095	14,830	8,648	6,774	17,691	31,871	17,141	45,281	4,718	2,010	7,156
	DevTools	1,412	693	14,948	6,588	6,730	3,812	7,878	22,924	29,131	2,007	484	1,053
	Firefox	3,038	1,946	15,666	11,474	4,697	4,751	30,508	19,090	47,541	2,458	2,071	2,223
	Firefox for Android Graveyard	469	196	14,834	3,741	2,186	2,909	5,509	33,637	37,086	2,065	4,328	507
	Firefox OS Graveyard	5,584	2,888	21,145	8,786	1,538	427	55,701	14,831	66,812	3,720	3,733	451
	MailNews Core	892	373	9,639	1,668	4,113	196	3,618	26,291	28,015	1,980	8,181	1,906
	SeaMonkey	1,257	501	19,410	4,013	15,262	863	9,288	83,556	88,501	4,386	22,300	9,221
	Testing	4,801	3,160	10,897	2,581	1,297	10,171	22,376	8,278	28,526	2,391	1,042	927
	Thunderbird	3,080	962	8,049	622	199	304	36,140	16,377	51,156	1,531	9,131	3,992
	Toolkit	3,408	1,680	16,309	5,510	3,118	3,060	20,811	26,716	45,067	2,834	5,279	3,118

3.4.2 Evaluation Metrics

As mentioned previously, the detection of a duplicate bug report is a recommendation task, which recommends a list of bug reports that may be duplicated with a given bug report. In this study, following (Runeson et al., 2007; Hindle et al., 2016), we calculate the cosine similarity between two semantic vectors of two bug reports to get the recommendation list. When evaluating detection performance, we calculate the Recall-N value for the tested bug reports. Here Recall-N refers to the ratio of duplicate bug reports whose master reports are within the suggested recommendation lists of N bug reports, over the total number of duplicate bug reports used in the experiment. In this study, we use N=20 for the duplicate bug detection task.

$$Recall\text{-}N = \frac{\#\ duplicate\ bugs\ correctly\ detected\ in\ top\ N}{total\ \#\ duplicate\ bugs} \quad (1)$$

For the other five classification BRM tasks, by following the practice of prior BRM studies (Xia et al., 2015b; Zhang et al., 2016; Ardimento and Mele, 2020; Jia et al., 2021), we adopt the widely used Precision, Recall, and F1 score to measure the performance of our models for BRM tasks. The calculation of the three metrics is shown in the following formula, where S represents a predicted class label.

$$Precision = \frac{\#\ bugs\ correctly\ predicted\ as\ S}{\#\ bugs\ predicted\ as\ S} \quad (2)$$

$$Recall = \frac{\#\ bugs\ correctly\ predicted\ as\ S}{\#\ bugs\ of\ S} \quad (3)$$

$$F1\text{-}Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

3.5 Research Questions

We try to answer four research questions (RQs) to understand the potential of word embedding models in bug report management tasks.

RQ1. Which machine learning approach performs best for BRM tasks on the whole?

To the best of our knowledge, most existing studies take BRM tasks as classification tasks, which generally apply typical machine learning (ML) algorithms, such as SVM, NB, and RF, to semantic features obtained by classical information retrieval models like VSM, LDA, etc. We have no idea whether the findings on existing studies about suitable ML for BRM tasks still hold under the situation that we use WE models to retrieve bug report semantics. Hence, we plan to answer this RQ with the aim of helping developers choose suitable ML for their BRM tasks. We compare four typical and commonly used ML methods in this paper, including Naive Bayes, Random Forest, Logistic Regression, and Support Vector Machine.

RQ2. Is there a word embedding model that generally outperforms others for BRM tasks?

Semantic retrieval of bug reports is a fundamental step for BRM tasks. For the five WE models (i.e., Word2Vec, GloVe, FastText, ELMo, and BERT) we studied in this paper, we attempt to explore whether there is a dominant WE model that always tends to outperform other WE models for BRM tasks. The answer to this RQ could guide developers in choosing the most suitable WE model for their BRM tasks at hand.

RQ3. Do word embedding models outperform the most used VSM model for BRM tasks?

Before the applications of WE models for BRM tasks, most studies used the simple and effective VSM model to retrieve the semantics of bug reports. Compared to VSM, WE models are relatively complex. The performance of WE models compared to that of VSM is still a question mark. Answering this question could help us understand whether it is beneficial to use WE or the VSM model for various BRM tasks.

RQ4. How will the performance change if word embedding models are fine-tuned with bug data?

To fully exploit the potential of WE models, it is not uncommon for researchers/practitioners to adapt the general pre-trained WE models to their tasks, through further training those models with domain-specific data. This naturally inspires us to explore whether we could obtain better prediction performance of BRM tasks if we further train the general WE models on bug data. In this study, we choose to further train BERT whose architecture is designed to facilitate further training. We use a large dataset of bug reports to further train the general pre-trained BERT. Then, we compare the performance difference between ML models built on feature vectors obtained by general pre-trained BERT and fine-tuned BERT, to answer this RQ.

4 Experiment Results

4.1 RQ1. Which machine learning approach performs best for BRM tasks on the whole?

Approach. To find the most suitable ML algorithm under the situation of using WE models to retrieve bug report semantic features, we conducted a series of experiments. For each BRM classification task (duplicate bug report detection is excluded as it is mainly considered as a recommendation rather than a classification problem), we first use five WE models to retrieve the semantics of bug reports for each product (as described in Section 3.1). Then we apply four ML approaches, namely LR, SVM, RF, and NB, to the feature vectors of five WE models separately to obtain prediction models. Last, we use corresponding testing datasets to evaluate the performance of those prediction models. In other words, we would have 800 prediction results for each BRM classification task, i.e., 20 products * 4 ML algorithms * 10 WE models with different dimension sizes = 800. Similarly, each ML would have 20 * 10 = 200 prediction values for each BRM task.

Since our results do not fulfill the residual normality and sphericity assumptions of parametric ANOVA, we followed the guidelines by (Demšar, 2006) to combine non-parametric Friedman test and Nemenyi post-hoc test to do the statistical comparison of multiple ML over multiple datasets. Friedman test can tell us whether or not there are statistically significant differences among three or more populations. If yes, then the Nemenyi test can be applied to compare the classifier with each other. The Nemenyi test uses the critical difference (CD) between average ranks to define significantly different populations. If the distance between the average ranks of two populations is larger than the CD value, then the two populations are significantly different. Otherwise, the experimental data is not sufficient to reach any conclusions related to the compared approaches. A CD diagram is used to visualize the Nemenyi test results. Such a diagram provides the rank order of compared approaches, the significance of observed differences, etc. Both the Friedman test and the Nemenyi test can only tell us whether there are statistically significant differences among compared approaches. We further use Cliff's delta effect size(Macbeth et al., 2011) to measure how large the difference might be quantitatively. The non-parametric Cliff's delta does not require the data to follow certain distributions like the normal distribution. The effect size of Cliff's Delta is divided into four levels:$|d| < 0.147$ (Negligible, N), $0.147 \leq |d| < 0.333$ (Small, S),$0.333 \leq |d| < 0.474$ (Medium, M) and $|d| \geq 0.474$ (Large, L).

Results. Table 3 and Figure 4 present the results of the Friedman test and the Nemenyi post-hoc test over the F1-scores of four ML algorithms on five BRM classification tasks. In this study, we use a p-value of 0.05 as the significance level threshold. If the p-value of a statistical test is less than 0.05, then we conclude that there is a statistically significant difference between/among compared populations. Related to the CD diagrams, a pair of techniques connected with a bold line indicates that the post-hoc test does not detect a

Table 3: Friedman tests of ML algorithms on five BRM classification tasks

Task	Chi-squared Statistic	p-value
Fixing Time	419.82	1.12E-90
Priority	512.47	9.42E-111
Reassignment	482.23	3.37E-104
Reopen	523.98	3.02E-113
Severity	409.59	1.85E-88

Fixing Time: Bug Fixing Time Prediction;
Priority: Bug Priority Prediction;
Reassignment: Report Field Reassignment Prediction;
Reopen: Re-opened Bug Prediction task;
Severity: Bug Severity Prediction task;
*The same rules applied to other tables and figures.

Table 4: The magnitude of statistically significant F1 score differences between ML algorithms for five BRM classification tasks

| Recall-N	Model Pairs	Cliff's Delta	Difference Magnitude
Fixing Time	RF vs SVM	0.2689	small
Fixing Time	RF vs Logistic	0.43965	medium
Fixing Time	RF vs NB	0.5026	large
Fixing Time	SVM vs Logistic	0.21775	small
Fixing Time	SVM vs NB	0.2603	small
Fixing Time	Logistic vs NB	0.0521	negligible
Priority	RF vs SVM	0.90745	large
Priority	RF vs Logistic	0.99235	large
Priority	RF vs NB	0.9936	large
Priority	SVM vs Logistic	0.34255	medium
Priority	SVM vs NB	0.6328	large
Priority	Logistic vs NB	0.46205	medium
Reassignment	RF vs SVM	0.1682	small
Reassignment	RF vs Logistic	0.29345	small
Reassignment	RF vs NB	0.6159	large
Reassignment	SVM vs Logistic	0.1139	negligible
Reassignment	SVM vs NB	0.4872	large
Reassignment	Logistic vs NB	0.44125	medium
Reopen	RF vs SVM	0.90745	large
Reopen	RF vs Logistic	0.99235	large
Reopen	RF vs NB	0.9936	large
Reopen	SVM vs Logistic	0.34255	medium
Reopen	SVM vs NB	0.6328	large
Reopen	Logistic vs NB	0.46205	medium
Severity	RF vs Logistic	0.17565	small
Severity	RF vs NB	0.6911	large
Severity	SVM vs Logistic	0.25895	small
Severity	SVM vs NB	0.7292	large
Severity	Logistic vs NB	0.56865	large
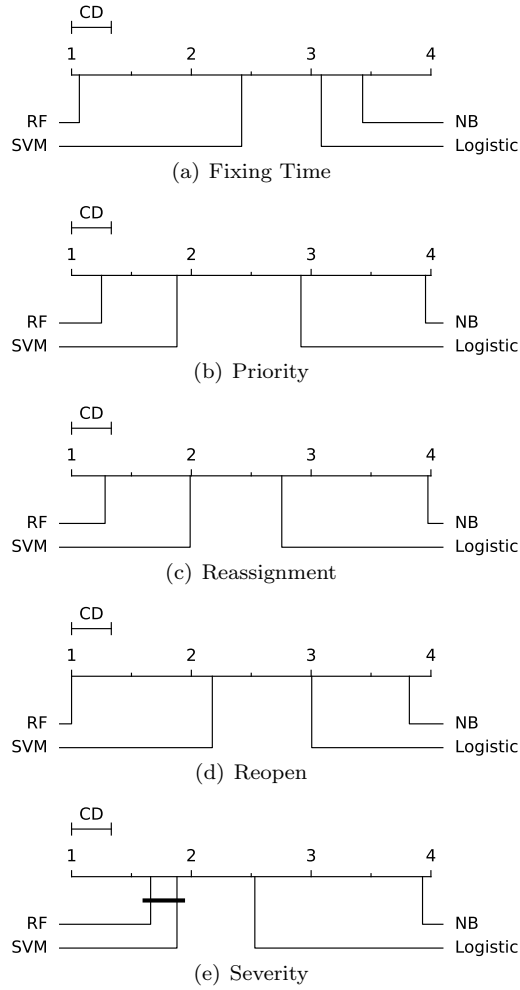
Fig. 4: Comparison of all classifiers against each other with the Nemenyi test. Groups of classifiers that are not significantly different (p-value>0.05) are connected with the bold line(s).

significant difference in F1 scores between them (e.g., the pair of RF and SVM in the bug severity prediction task in Figure 4).

From the p-values in Table 3, we can conclude that there indeed exists a statistically significant difference among the four ML methods for every studied BRM classification task. Further, from the CD diagram in Figure 4, we can find that RF achieved the best performance ranks among four ML algorithms on all five BRM classification tasks. All performance differences between them are statistically significant except the pair of RF and SVM in the bug severity prediction task. To understand the magnitude of performance differences

between these ML approaches, we further calculate their corresponding Cliff's delta effect sizes. The results are shown in Table 4. From the table, we can find that the F1-score difference between RF and SVM ranges from small to large over four BRM tasks; the performance difference over the bug severity prediction task is excluded in that it shows no statistical significance in the CD diagram. The performance differences between RF and LR/NB are also obvious according to Cliff's delta effect sizes, with 6 large, 1 medium, and 1 small magnitude levels respectively on five BRM classification tasks. Similarly, we can conclude that SVM outperforms LR and NB while LR outperforms NB in five BRM classification tasks.

Finding 1. RF performs best among studied traditional ML algorithms, the F1-score difference between RF and the other three ML algorithms ranges from small to large (mostly large) according to Cliff's delta effect sizes.

4.2 RQ2. Is there a word embedding model that generally outperforms others for BRM tasks?

Approach. Similar to RQ1, we use the same statistic tests, i.e., Friedman test, Nemenyi test, and Cliff's delta effect size, to compare the studied WE models over 20 products on six studied BRM tasks. In detail, for the five BRM tasks, we would collect the F1 scores by applying four ML algorithms to feature vectors of bug reports obtained by different WE models. As for the duplicate bug report detection task, for each duplicate bug report, we would first calculate the cosine similarities between the WE vectors of this bug report and all its previous bug reports; then, a rank list is created from those similarity scores, where bug reports with the largest scores ranked first in the list. Based on the rank lists of all duplicated bug reports and the ground truth, i.e., duplicate pairs obtained in Section 3.4.1, we can compute the Recall-N values. We consider five N values in this study, namely 1/5/10/15/20, indicating providing a rank list of the most similar 1/5/10/15/20 bug reports for a given duplicate bug report. When we compare the WE models over the duplicate bug report detection task, we conduct the statistical tests over Recall-1, Recall-5, Recall=10, Recall-15, and Recall-20, respectively. For BRM classification tasks, each WE model would have a population of 80 prediction result samples, i.e., 4 ML algorithms * 20 products, used to perform statistical tests; while for the cosine-similarity-based duplicate bug detection task, the number would be 20 (we conduct experiments on 20 products) at each level of Recall-N.

Results. (a) Comparison of FastText/GloVe with different dimension sizes. As mentioned in Section 3, the official websites provide several dimension options for some pre-trained WE models. For example, users can use GloVe-50/100/200/300 to obtain a GloVe embedding vector with 50/100/200/300 elements separately. To understand the possible effects of dimension sizes over BRM tasks, we compare the prediction performance of using different dimension sizes for pre-trained FastText (100/200/300) and GloVe (50/100/200/300). We first check the Friedman test results and find that all p-values are <0.05

(detailed test results can be found in the Appendix Table 12). Hence, we can conclude that there indeed exists pre-trained FastText/GloVe with a certain dimension size that performs differently than others.

By further checking the CD diagrams in Figure 5, 11, 12, 13 (Fig. 11-13 show a similar trend with Fig. 5, we place them in the Appendix section) we can find that, on five BRM classification tasks, FastText-300 performs best, with FastText-200 being the second best; the only exception is that we cannot conclude whether FastText-200 performs better or not than FastText-100 on the bug fixing time prediction task as the test result is not statistically significant (connected with a bold line). As for GloVe, we can find that except for the connected pair of GloVe-100 and GloVe-50 in the bug report reassignment task (no conclusion can arrive), GloVe models generated larger vectors always perform better than those with smaller dimension sizes on the five BRM classification tasks. As for the duplicate bug report detection task, we cannot tell whether GloVe-300 performs better than Glove-200 or not, but we can tell that GloVe-300 and GloVe-200 perform better than GloVe-50. The differences between pairs (GloVe-200, GloVe-100), (GloVe-100, GloVe-50) also have no statistical significance, indicating no conclusion can be arrived.

> **Finding 2.** On the whole, pre-trained FastText/GloVe with the largest dimension size perform better than other smaller dimension sizes over six studied BRM tasks.

Table 5: Friedman tests of five WE models on five BRM classification tasks and the duplicate BR detection task

Five BRM Classification Tasks			Duplicate BR Detection		
Task	Chi-squared Statistic	p-value	Recall-N	Chi-squared Statistic	p-value
Fixing Time	0.44	9.79E-01	Recall-1	71.12	1.31E-14
Priority	18.83	8.45E-04	Recall-5	75.71	1.40E-15
Reassignment	7.85	9.72E-02	Recall-10	76.63	8.96E-16
Reopen	45.73	2.80E-09	Recall-15	76.63	8.96E-16
Severity	17.21	1.75E-03	Recall-20	76.36	1.02E-15

(b) Comparison of five WE Models. Based on the above analysis, we decided to use GloVe-300 and FastText-300 while comparing the performance of five WE models, that is, FastText-300, GloVE-300, Word2Vec-300, BERT-768, ELMo-1024. Similarly, we resort to the Friedman test results in Table 5 and CD diagrams in Figures 6 and 7, to help us compare these models. From the Friedman test results, we find that nothing can be concluded on the bug fixing time prediction and bug report field reassignment tasks as the tests are not statistically significant. For the remaining BRM classification tasks, we can find that most pairs are connected with bold lines, indicating we cannot determine whether the connected WEs perform the same. Only a few comparisons show statistically significant performance differences, such as the ELMo-1024 performs better than FastText-300 on the bug priority prediction task. As for the duplicate bug report detection task, at different recommendation list
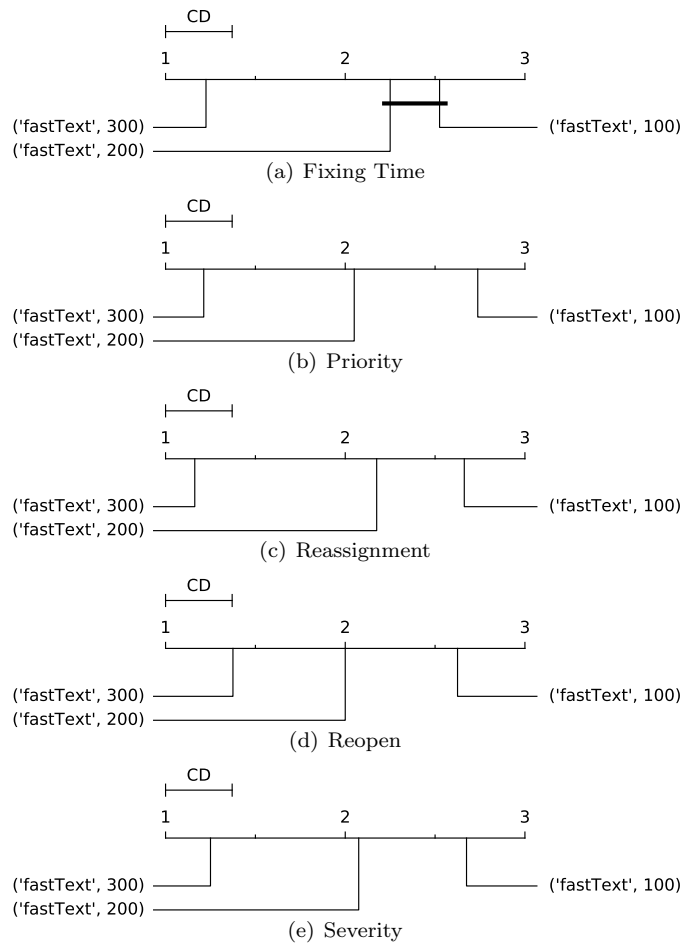
Fig. 5: Comparison of FastText with different dimension sizes on five BRM classification tasks with the Nemenyi test. Groups of FastText-N that are not significantly different (p-value>0.05) are connected with the bold line(s).

sizes, i.e.,1/5/10/15/20, we can find that the group of small WE models that generate smaller embedding vectors, namely FastText-300, GloVe-300, and Word2Vec-300 generally perform better than the relatively larger group of WE models, including BERT-768 and ELMo-1024. But we cannot tell whether one performs better or not than another one within the small WE group or large WE group, as the connected bold lines indicate statistically non-significant differences.
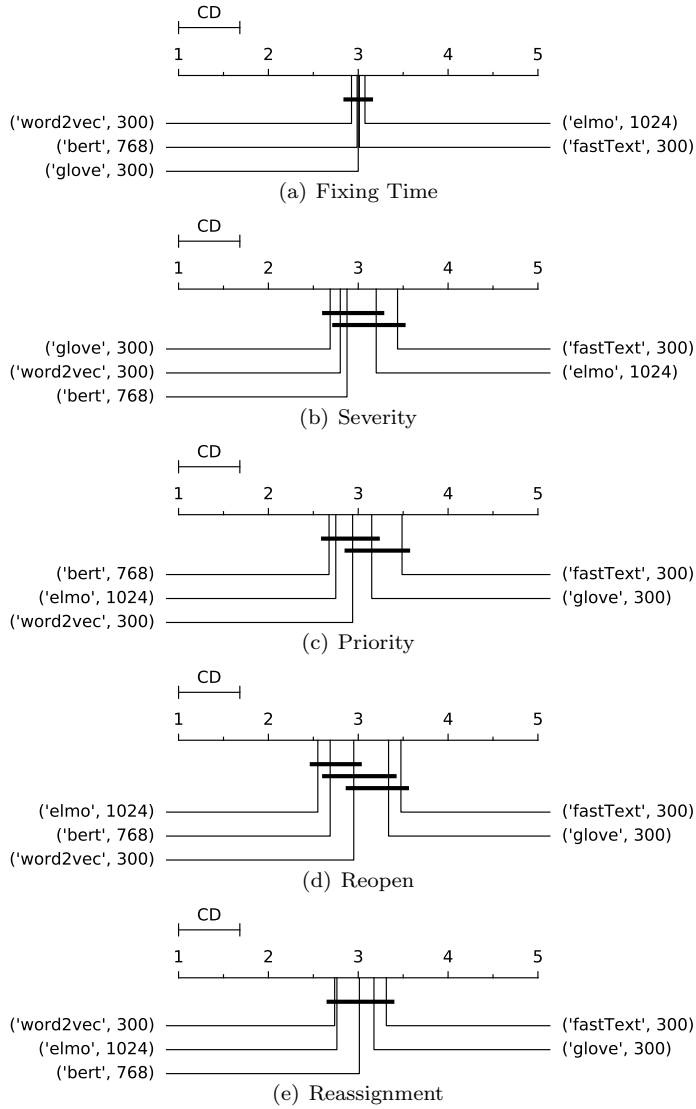
Fig. 6: Comparison of five WE models against each other on five BRM classification tasks with the Nemenyi test. Groups of WE models that are not significantly different (p-value>0.05) are connected with the bold line(s).

Finding 3. In most cases, we did not detect statistically significant performance differences among five WE models in five BRM classification tasks; on the duplicate bug report detection task, small WE models (Fast-Text/GloVe/Word2Vec that generate 300-dimensional embedding vectors) performed better than relatively large WE models (BERT and ELMo that generate 768 and 1024-dimensional vectors separately).

(a) Recall-1

(b) Recall-5

(c) Recall-10

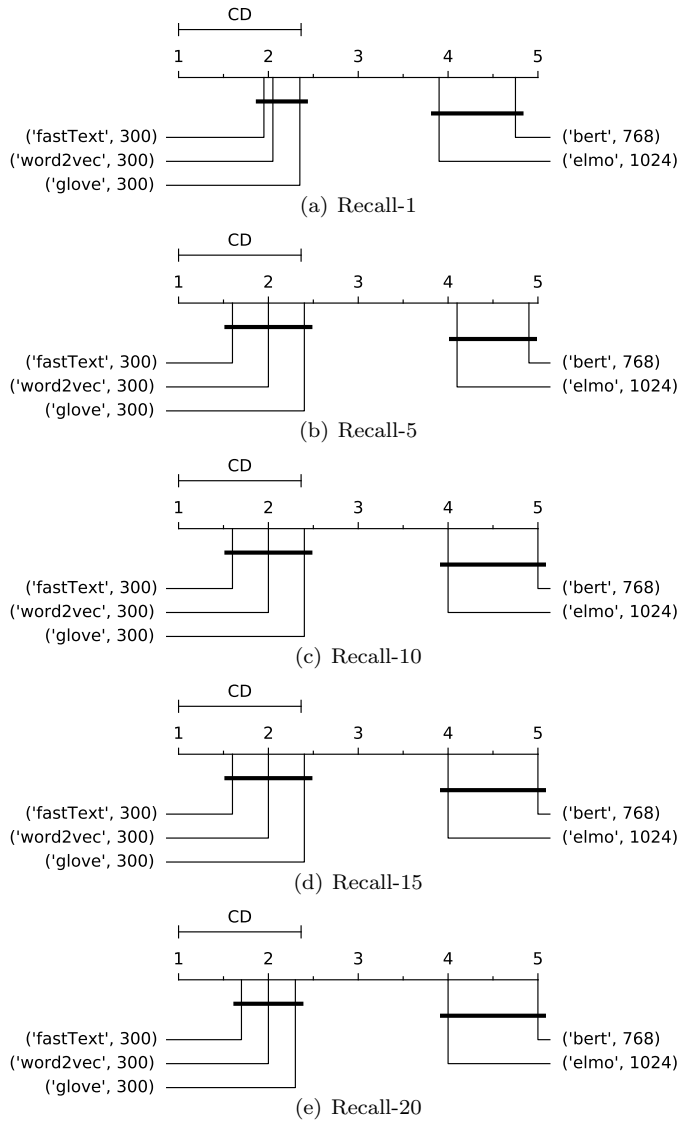(d) Recall-15

(e) Recall-20

Fig. 7: Comparison of five WE models against each other on the duplicate BR detection task with the Nemenyi test. Groups of WE models that are not significantly different (p-value>0.05) are connected with the bold line(s).

4.3 RQ3. Do word embedding models outperform the most used VSM model for BRM tasks?

Approach. To compare the effectiveness between VSM and the word embedding techniques, we build different classification and recommendation models where VSM and the five WE models are used to extract the semantic features of bug reports separately for the six BRM tasks on 20 products from Eclipse

and Mozilla. For the five BRM classification tasks, we chose RF as the classifier as it performed better than other traditional ML algorithms in RQ1. The term weighting strategy of VSM we used is the tf-idf. As suggested in (Demšar, 2006), instead of using the Nemenyi post-hoc test for the case when all approaches are compared to each other, Bonferroni-Dunn test (Dunn, 1961) is more powerful than Nemenyi test in the specific case when all approaches are compared with a control one as it controls the family-wise error in multiple hypothesis testing. Hence, we use the Bonferroni-Dunn test to compare all WE models to the controlled model, namely VSM. The test results can be found in the CD diagrams shown in Figures 8 and 9. In the figures, VSM is the controlled model. For each subfigure, a bold line is drawn to mark an interval where all WE models with ranks outside the marked interval are significantly different from the control VSM at `p-value<0.05`.

Results. From the results in Figure 8, we can find that for three out of five BRM classification tasks (i.e., bug fixing time prediction, bug priority prediction, bug report field reassignment prediction), we cannot arrive at any conclusion about whether WE models perform better or not than VSM, as all statistical tests are not statistically significant (the ranks of all WEs are within the marked interval bold line).

For the bug severity prediction and reopened bug prediction tasks, we can tell that the small WE models, namely FastText-300, GloVe-300, and Word2Vec-300 perform worse than the traditional VSM; while we cannot conclude whether the relatively large WE models, namely Bert-768 and ElMo-1024 perform the same with VSM or not. For the comparisons between VSM and those small WE models that show statistically significant performance differences, we further check the magnitude of these differences by Cliff's delta effect size. Table 6 shows the results. From the table, we can find that the magnitude of F1 score differences between VSM and the three small WE models on the bug severity prediction task are all small, while for the reopened bug prediction task, the difference is negligible according to Cliff's delta effect sizes. As

Table 6: The magnitude of statistically significant F1 score differences between small WE Models and VSM for two BRM classification tasks

Task	Model Pairs	Cliff's Delta	Difference Magnitude
Severity	fastText-300 vs VSM	-0.185	small
Severity	glove-300 vs VSM	-0.270	small
Severity	word2Vec-300 vs VSM	-0.235	small
Reopen	fastText-300 vs VSM	-0.030	negligible
Reopen	glove-300 vs VSM	-0.025	negligible
Reopen	word2Vec-300 vs VSM	-0.020	negligible

for the recommendation task, i.e., duplicate bug report detection, we can find that at all recommendation levels, namely, 1,5,10,15,20, in most cases except the three comparisons of VSM and FastText-300 (no statistically significant RecallN difference is detected), VSM always achieved the best RecallN scores
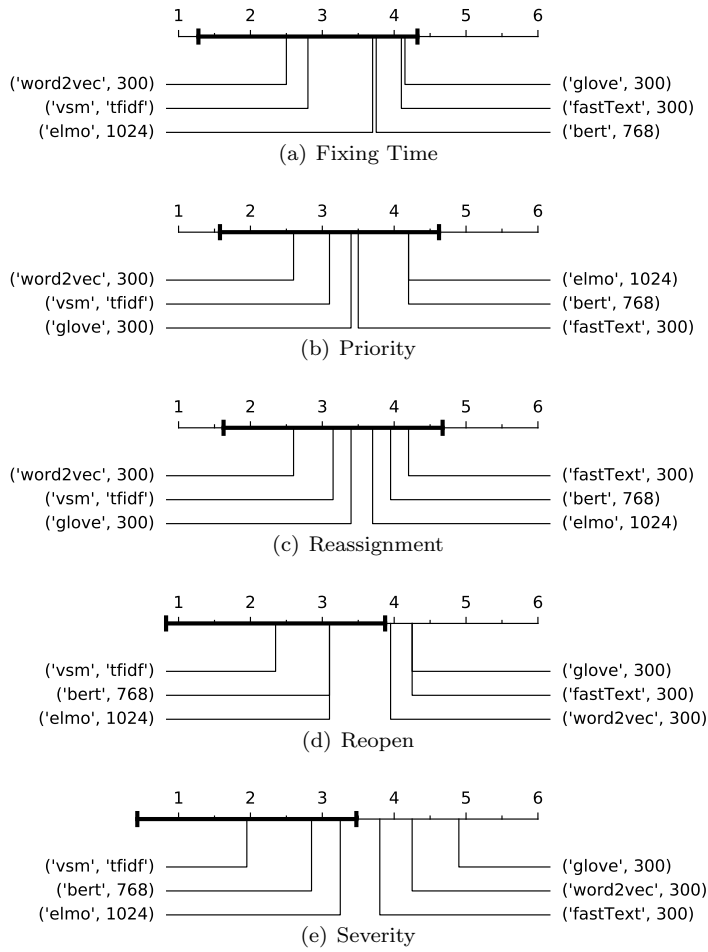
Fig. 8: Comparison of five WE models against VSM on five BRM classification tasks with the Bonferroni-Dunn test. Models with ranks outside the marked interval line are considered significantly different from the VSM.

than WE models. Similarly, we further check the magnitude of RecallN differences for those comparisons with statistical significance. The results are shown in table 7. The table shows that all differences between VSM and other WE models are large based on Cliff's delta effect size values.

Finding 4. Most comparisons of WE models over VSM on five BRM classification tasks do not show statistical significance from the Bonforroni-Dunn tests; for the duplicate bug report detection task, VSM is found to outperform WE models, with a large RecallN difference according to the Cliff's delta effect size.

(a) Recall-1

(b) Recall-5

(c) Recall-10

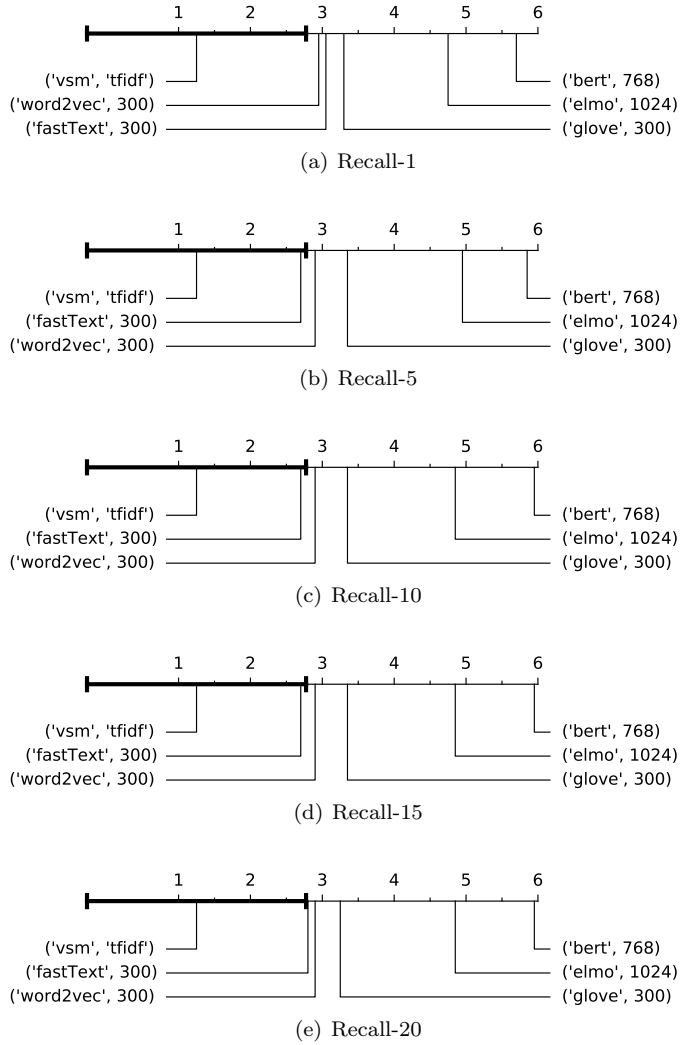(d) Recall-15

(e) Recall-20

Fig. 9: Comparison of five WE models against VSM on the duplicate BR detection task with the Bonferroni-Dunn test. Models with ranks outside the marked interval line are considered significantly different from the VSM.

4.4 RQ4. How will the performance change if word embedding models are fine-tuned with bug data?

Approach. This RQ aims to investigate how the performance of WE models would change when they are fine-tuned on a domain-specific corpus. To this end, we first constructed a bug dataset of more than 3 million bug reports crawled from the Eclipse, Mozilla, and Apache foundations. We extract the summary and description from each bug report and then feed them sentence

Table 7: The magnitude of statistically significant Recall-N differences between WE models and VSM for the duplicate BR detection task

Recall-N	Model Pairs	Cliff's Delta	Difference Magnitude
Recall-1	Bert vs VSM	-0.620	large
Recall-1	ElMo vs VSM	-0.740	large
Recall-1	fastText-300 vs VSM	-0.675	large
Recall-1	glove-300 vs VSM	-0.545	large
Recall-1	word2vec-300 vs VSM	-0.500	large
Recall-5	Bert vs VSM	-0.765	large
Recall-5	ElMo vs VSM	-0.850	large
Recall-5	glove-300 vs VSM	-0.720	large
Recall-5	word2vec-300 vs VSM	-0.690	large
Recall-10	Bert vs VSM	-0.815	large
Recall-10	ElMo vs VSM	-0.875	large
Recall-10	glove-300 vs VSM	-0.770	large
Recall-10	word2vec-300 vs VSM	-0.705	large
Recall-15	Bert vs VSM	-0.840	large
Recall-15	ElMo vs VSM	-0.890	large
Recall-15	glove-300 vs VSM	-0.815	large
Recall-15	word2vec-300 vs VSM	-0.755	large
Recall-20	Bert vs VSM	-0.855	large
Recall-20	ElMo vs VSM	-0.890	large
Recall-20	glove-300 vs VSM	-0.820	large
Recall-20	word2vec-300 vs VSM	-0.795	large

by sentence to the general pre-trained BERT, i.e., the bert-base-uncased model downloaded from HuggingFace [10]. We chose BERT's built-in Masked Language Model (MLM) task during the further training to let the general pre-trained BERT capture the deeper contextual semantics of bug data. With MLM, the model is trained to fill in the missing tokens in a text where certain tokens are masked or replaced with a special token, typically "[MASK]". After we obtained the fine-tuned BERT (named ftBERT), we use ftBERT to extract the semantics of bug reports and then perform the six BRM tasks. We also collect the F1 scores of five BRM tasks and RecallN of the duplicate bug report detection task after adopting ftBERT for semantic extraction. Then, we compare the performance of using ftBERT with that of general pre-trained BERT. As the comparison only involves two models, we use the Wilcoxon signed ranks test to determine whether the F1 score or RecallN difference is statistically significant. The Wilcoxon singed-ranks test is non-parametric. It assumes commensurability of differences but only qualitatively, does not assume normal distributions of samples, and is less affected by outliers than the alternative paired t-test (Demšar, 2006). After conducting the Wilcoxon test, we use the Cliff's delta effect size to identify the magnitude of such a difference.

Results. Table 8 shows the test results and the associated effect sizes over five BRM classification tasks. The table shows that the Wilcoxon signed-ranks test

---

[10] http://huggingface.co/bert-base-uncased

Table 8: Wilcoxon signed ranks tests and the magnitude of F1 score differences between general pre-trained BERT and fine-tuned BERT over five BRM classification tasks

Task	P-value	Cliff's Delta	Difference Magnitude
Fixing Time	2.3334E-05	0.0463	negligible
Priority	0.2124	-0.0072	negligible
Reassignment	0.0037	-0.0278	negligible
Reopen	0.0041	0.0669	negligible
Severity	0.2517	0.0106	negligible

Table 9: Wilcoxon signed ranks tests and the magnitude of Recall-N differences between general pre-trained BERT and fine-tuned BERT on the duplicate BR detection task

Recall-N	P-value	Cliff's Delta	Difference Magnitude
Recall-1	0.0172	0.0650	negligible
Recall-5	0.3144	0.0375	negligible
Recall-10	0.7841	-0.0050	negligible
Recall-15	0.5958	0.0000	negligible
Recall-20	0.2772	-0.0025	negligible

detects a statistically significant F1 score difference between the general pre-trained Bert and the fine-tuned BERT (i.e., ftBERT) on three out of five BRM classification tasks. They are fixing time prediction, reopened bug prediction, and bug report field reassignment prediction tasks. By further referring to Cliff's delta effect sizes, we can say that the pre-trained BERT performs better than the ftBERT, but the difference is negligible.

As for the duplicate bug report detection task shown in Table 9, except the Recall_1, all comparisons do not show statistically significant differences. In other words, we cannot conclude whether the fine-tuned BERT is better than the general pre-trained BERT based on our experimental results.

> **Finding 5.** We only detect a negligible performance difference between fined-tuned BERT and the general pre-trained BERT on three out of six BRM tasks. Other comparisons do not show statistical significance from the Wilcoxon signed-ranks tests.

5 Discussion

In this section, we first discuss the potential effect of hyperparameter tuning of RF on arrived conclusions. Then we introduce the implications of our work. Last, we present the threats to the validity of our study.

5.1 Hyper Parameter Tuning

Motivation. Among the six BRM tasks, five of them are classification tasks, which involve applying a specific ML algorithm to WE-based or VSM-based semantic representations of bug reports. Given that an ML algorithm generally has some hyperparameters whose values could also be set by users, and it is not uncommon for practitioners to tune these hyperparameters to obtain optimal hyperparameter settings for their adopted ML models, we also checked whether the hyperparameter tuning would affect our arrived conclusions in this study. This could help us understand whether hyperparameter optimization plays a role in our arrived sort of counterintuitive results, i.e., with default parameter settings, we did neither find the advantage of using WE over VSM in studied BRM tasks nor the performance improvement of fine-tuning the general pre-trained BERT with bug data.

Approach. Specifically, for five BRM classification tasks[11], we decided to perform hyperparameter tuning on the RF classifier after we obtained the semantic representations of bug reports through the WE or VSM models. RF is an ensemble method that constructs multiple decision trees at training time and leverages their power to make predictions. Its hyperparameters include the number of samples or features used to train a meta tree, the number of meta trees to be constructed, the maximum depth of a tree, the minimum number of samples required to split an internal node, etc. Each hyperparameter is associated with a value range, e.g., for the number of samples used to construct a tree, its value range is (0.0,1.0] (e.g., 0.1 means 10 percent of all samples are used to build a meta tree).

Given the constraints of combinatorial explosion and computational resource limitations, it is generally impractical to test all possible hyperparameter settings when tuning the RF. Hence, we adhere to the common practice that using a well-chosen subset of the hyperparameter space to find the optimal settings for each dataset. Detailedly, inspired by the findings of RF tuning studies that are widely recognized by the research community (Probst et al., 2019b,a; Probst and Boulesteix, 2018), we decided to fine-tune two hyperparameters, namely the number of samples used to build a tree (referred to as sampleNum), and the number of features used to construct a tree (referred to as featureNum), in that they are found to be most important and tunable (may have a significant effect on results). For other less important and relatively untunable hyperparameters (like the number of trees to be constructed (Probst et al., 2019a)), since the settings of sampleNum and featureNum actually have inexplicitly placed constraints on their value options (e.g., affect the depth of a tree (Probst et al., 2019b)), we think it is acceptable to focus on tuning the two hyperparameters.

During hyperparameter tuning, we choose five values for the sampleNum hyperparameter: 0.2*N, 0.4*N, 0.6*N, 0.8*N, and 1.0*N, where N is the num-

---

[11] Duplicate bug report detection is a recommendation task that does not involve applying ML algorithms, hence not considered here.

Table 10: The magnitude of statistically significant F1 score differences between WE Models and VSM for BRM classification tasks after hyperparameter tuning.

Task	Model Pairs	Cliff's Delta	Difference Magnitude
Fixing Time	word2vec-300 vs VSM	0.09	negligible
Reassignment	elmo-1024 vs VSM	-0.035	negligible
Reopen	glove-300 vs VSM	-0.020	negligible
Reopen	fastText-300 vs VSM	-0.025	negligible
Severity	word2Vec-300 vs VSM	-0.130	negligible
Severity	glove-300 vs VSM	-0.185	small
Severity	elmo-1024 vs VSM	-0.125	negligible

ber of training instances. By default, RF constructs each meta tree using all N training instances. For the featureNum hyperparameter, we choose seven values, namely sqrt(M), log2(M), 0.1*M, 0.2*M, 0.3*M, 0.4*M, 0.5*M, where M is the number of instance features (e.g., M would be 300 for 300-dimensional Word2Vec model, and M would be the number of all unique words in the VSM model). sqrt(M) and log2(M) are two recommended value options by the cuML and scikit-learn libraries. In both cuML and scikit-learn, the default value is sqrt(M). Given that it is generally best practice to use only a limited number of features to construct each tree so that RF can ensure its diversity of meta trees (such diversity could help RF improve its predictive performance, reduce overfitting, and effectively handle complex datasets with many features), we decide not to consider larger featureNum (i.e., `>0.5*N and <=1.0*N`). Hence, we would have 35 combinations (5*7=35) of sampleNum and featureNum. For each BRM classification task, we rerun our experiments with each parameter combination on every dataset whose bug reports are represented by five pre-trained WE, the VSM, and the fine-tuned WE (ftBERT), with parameters besides sampleNum and featureNum still using their default settings. In total, we would repeat our whole experiments with RF 245,000 times (35 parameter combinations * 5 BRM tasks * 20 projects * 7 semantic representations * 10 stratified cross-validation).

Results. After we performed hyperparameter tuning, we retrieved the best F1 scores from 35 parameter combinations for each dataset of five BRM tasks. Then, we compared the performance of BRM techniques using WE models and that of using VSM. Similarly, we also checked the performance between BRM techniques using general pre-trained WE and the fine tuned WE with bug data. Details are as follows.

(a) WE vs. VSM after hyperparameter tunning. We used the same statistical method as RQ3 to explore whether WE models show advantages over VSM in five BRM classification tasks after hyperparameter tuning over RF. Fig. 10 shows the results. From the figure, we can find that most comparisons between WE and VSM are still not statistically significant after hyperparameter tuning (their ranks are within the marked interval bold line), just like that by using default hyperparameter settings in Fig. 8.
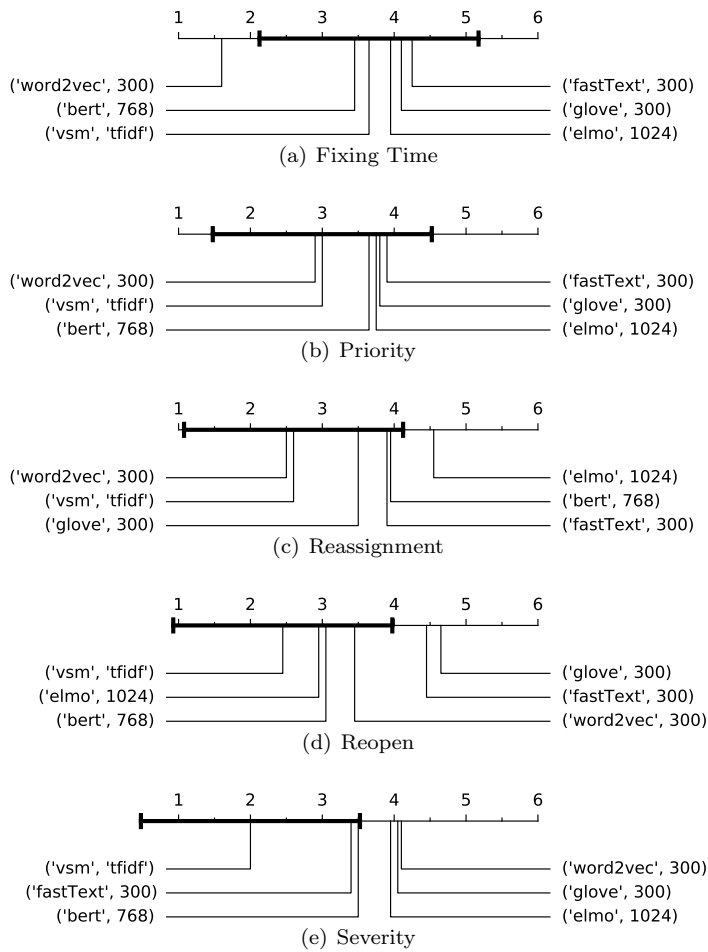
Fig. 10: Comparison of five WE models against VSM on five BRM classification tasks with the Bonferroni-Dunn test after hyperparameter tuning. Models with ranks outside the marked interval line are considered significantly different from the VSM.

Only few performance comparisons between WE models and the VSM show statistical significance, which means we can tell which one is better from the statistical perspective. Specifically, for the bug-fixing time prediction task, Word2Vec outperformed the VSM. For the bug priority prediction task, all comparisons are not statistically significant. For the bug field reassignment prediction task, ELMo performed worse than the VSM. For the reopened bug prediction task, GloVe and FastText performed worse than the VSM. For the bug severity prediction task, three WE models including Word2Vec, GloVe, and ELMo, performed worse than the VSM. To understand how large these

performance differences are, we further computed their Cliff's delta effect size as shown in Table 10. From the table, we can see that only the magnitude of F1 score differences between VSM and GloVe on the bug severity prediction task is small, with all others being negligible according to Cliff's delta effect sizes. In other words, whether we performed hyperparameter tuning or not, we did not find the advantages of using WE models over the VSM model on the five BRM classification tasks.

(b) General pre-trained BERT vs. ftBERT after hyperparameter tuning. After we obtained the embedding vectors of general pre-trained BERT and the fine-tuned BERT, we also tuned RF with different combinations of sampleNum and featureNum to these instance feature vectors to build prediction models with different performances. Then, the models with the best prediction performance are selected to compare the general pre-trained BERT and ftBERT in the five BRM classification tasks. Like in RQ4, we use the Wilcoxon signed ranks test to determine whether the F1 score difference between general pret-trained BERT and the ftBERT is statistically significant or not, and use Cliff's delta effect size to identify the difference magnitude. Table 11 shows the corresponding test results. From the table results with hyperparameter tuning, we can find that all F1 score differences between the general pre-trained BERT and the ftBERT are negligible according to Cliff's delta effect size; this is similar to that of using default hyperparameter settings. But we cannot tell whether these negligible differences are a fact or just caused by accident, as the p-values of their Wilcoxon signed ranks tests are all larger than 0.05 (which means no statistical significance was detected).

Table 11: Wilcoxon signed ranks tests and the magnitude of F1 score differences between general pre-trained BERT and fine-tuned BERT over five BRM classification tasks after hyperparameter tuning.

Task	P-value	Cliff's Delta	Difference Magnitude
Fixing Time	0.8408	0.01	negligible
Priority	0.1327	-0.01	negligible
Reassignment	0.5958	0.00	negligible
Reopen	0.4524	0.02	negligible
Severity	0.1536	0.06	negligible

5.2 Implication

We would recommend practitioners use RF to build their BRM classifiers when pre-trained WE models are used to represent the semantics of bug reports. In RQ1, we find that RF generally outperformed the other ML classifiers (SVM, NB, and LR) based on various WE-based semantic feature representations of bug reports. The performance differences

among them ranged from small to large (mostly large) measured by Cliff's delta effect sizes. Such an advantage, to a certain degree, may be due to the fact that RF is an ensemble method based on a list of meta-decision trees, which can combine the advantages of different meta-classifiers for learning specific semantic aspects of bug reports. Moreover, during our experiments, we found that as an ensemble approach, the training process of RF is actually not slower than, e.g., the second-best SVM. Hence, RF would be a good choice for practitioners to build BRM models based on WE feature vectors of bug reports. Further, if practitioners choose RF as their BRM classifier, it would be suitable for them to use the default parameter settings when they need to further decide which pre-trained WE to use, as the conclusion would not make much difference based on our hyperparameter tuning results in Sect. 5.1.

Practitioners have a relatively high degree of freedom when choosing which WE model to use in their BRM tasks as none of them is dominant, but larger FastText/GloVe is more recommended than their smaller versions once FastText/GloVe is their option. The statistical test results in RQ2 indicate that we cannot conclude whether a dominant WE model exists for five BRM classification tasks. This means practitioners are more free to choose WE models that they can consider more about their concrete task constraints, such as computing power, storage space, or room for future fine-tuning. Further, if practitioners decide to use FastText/GloVe for semantic retrieval in their BRM tasks, we recommend they use the largest ones, i.e., FastText-300/GloVe-300, as they are found to generally perform better than their smaller versions such as FastText-100/GloVe-50. This is also consistent with the statement that for a general pre-trained WE model, the larger dimension generally means the more information it can capture. The FastText-300/GloVe-300 (actually also includes the Word2Vec-300) also perform better than other large-WE models (BERT-768/ELMo-1024) for the duplicate bug report detection task based on our experimental results.

The traditional VSM is still a good option of semantic representation for BRM tasks. From the results of RQ3, regardless of whether they show statistically significant differences or not, we find that the traditional VSM generally gets better performance than WE models when combined with traditional ML methods. We think the reason may lie in the not important or noisy terms. For the VSM model, the tf-idf value may do reveal the importance of a term in a bug report. While in the use of WE models, the importance of key terms is weakened. That is, in general settings of using WE models for bug report semantic extraction, each term is represented as an N-dimensional numeric vector first, and then an average pooling strategy is applied to these vectors to get a final N-dimensional vector for a bug report. As the number of important terms is generally much smaller than that of insignificant terms, averaging them would flatten the importance of key terms but relatively increase the influence of less important terms. Whatever the reason, at least from the performance perspective, the traditional VSM is still a good choice for practitioners to use in their BRM tasks, especially in the duplicate bug report task. Further, it should be noted that, for five BRM classification tasks,

we find that the performance difference between the VSM and five WE models is quite small. Hence, if practitioners focus more on other aspects than performance such as the computation cost, we think it would be a good choice to use pre-trained WE models (in our experiments, the sparse, high-dimensional semantic vector of the VSM costs much more training time than the smaller N-dimensional vectors of WE models (N ranged from 50 to 1024)).

Further efforts are required to help WE models better learn the semantics of bugs. In RQ4, we find that using the fine-tuned BERT with bug data to represent the semantics of bug reports did not achieve better BRM performance than the general pre-trained BERT. This may lie in the knowledge gap between general website contents and domain-specific bug data, which is still not learned well by using the built-in objective task to further train the general pre-trained BERT. Or it may be the quality of the textual content of bug reports is not high and the embedded noisy information adversely affects the fine-tuning process. In the future, one valuable attempt would be to test or develop new objective tasks to fine-tune general pre-trained WE models to better capture the domain knowledge of bug reports. Meanwhile, we think it would be interesting and valuable to combine key term identification techniques (which help reduce noisy bug information) with WE models to perform semantic retrieval of bug reports.

5.3 Threats to Validity

Internal Validity The experimental design method used in the empirical study does have an impact on the results. In our work, we try our best to follow the literature of previous experiments for each step in the experiment. In order to make the experimental results more reliable, researchers may repeat the whole process many times, such as Bennin et al. (Bennin et al., 2019), Giger et al.(Giger et al., 2010) (they repeated 10 times). In our experiment, we do not repeat each BRM task on the same open-source software multiple times. Nevertheless, we use the following strategies for experiments instead: 1) the top 10 products in each open source project are selected for experiments, which is equivalent to conducting 10 repeated experiments on that project; 2) four machine learning classification methods are used for each WE technique, which equals to conducting 4 repeated experiments for that WE technique; 3) last, we perform 10-fold cross-validation during model building for each BRM classification task. The above-repeated experiments to some extent ensure the reliability of our observations in the study.

  External Validity Our experiments are conducted on all open-source projects from Eclipse and Mozilla with varied scale sizes. We cannot guarantee that our arrived conclusions could be applicable to other OSS projects, non-open-source projects, or commercial projects. However, considering that all selected experimental projects are typical software products with active developments among developers and are popular among users, this makes us believe that our findings still shed some light on the adoption of word embed-

ding techniques in BRM tasks. We encourage future research to replicate our studies in bug reports from other projects to further improve the generalizability of our findings. Another threat is that we only use the oversampling strategy to solve the data imbalance problem. Whether the conclusions could hold for other imbalanced-data-handling strategies such as SMOTE is still unknown. Further consideration arises from our methodology in constructing experimental datasets, we perform average pooling to word embedding vectors to obtain the final instance features. Actually, besides the commonly-used average pooling (Deshmukh et al., 2017; Messaoud et al., 2022), there also exist other pooling strategies such as max pooling. We cannot guarantee that our conclusions still hold when tested with other pooling strategies. We encourage researchers to replicate our studies to further improve the generalizability of our work in the future.

6 Related Work

Among various kinds of bug report management technologies, a common process is to extract the textual semantics of bug reports. The extracted semantics would be used as an important source of features that are fed into e.g., a classification model or recommendation model, to help build models to finish a series of BRM tasks. Hence, it is full of important to explore semantic extraction approaches for bug reports. Our study is also a study that focuses on investigating the potential of word embedding models in semantic feature extraction of bug reports. Based on the way in semantic extraction, we divide existing studies into two parts. One is to introduce existing BRM studies that use traditional information-based approaches to represent the semantics of bug reports, such as VSM, LDA, etc. The other one is to introduce BRM studies that try to adopt word embedding techniques to extract the semantics of bug reports. They are as follows.

IR-Based Semantic Extraction To the best of our knowledge, a majority of bug report management tasks generally use traditional IR-based approaches to represent the textual semantics of bug reports, including VSM, Latent Dirichlet Allocation (LDA), and Latent Semantic Indexing (LSI), etc. VSM uses a numeric vector to represent a bug report. The length of the vector is the number of unique words in a corpus. Each element the vector represents a unique word and its numeric value represents the weight (e.g., term frequency) of the word in the bug report. As a classic information retrieval method, VSM is used in many BRM tasks and those tasks often use the term frequency-inverse document frequency (TF-IDF) to weigh the terms of bug reports, those tasks include bug localization (Wang et al., 2014; Tantithamthavorn et al., 2018; Liu et al., 2019), duplicate bug detection (Runeson et al., 2007; Jalbert and Weimer, 2008; Wang et al., 2008; Tian et al., 2012), bug priority prediction (Tian et al., 2013; Alenezi and Banitaan, 2013; Tian et al., 2015; Izadi et al., 2022), and other tasks (Giger et al., 2010; Xia et al., 2015b,a; Zhang et al., 2022). LSI is an indexing and retrieval method that can identify

the relationship between the terms and concepts contained in an unstructured collection of text by using mathematical techniques such as Singular Value Decomposition (SVD). It is used in earlier, for example, bug localization studies (Poshyvanyk et al., 2006, 2007) or as a baseline or complementary part to later studies (Lukins et al., 2008; Chawla and Singh, 2014). LDA is a topic model that represents a document with a distribution of topics. Its representation is a normalized vector in the topic space by converting a bag-of-words document from a word space into a topic space. It is usually applied in duplicate bug detection task (Nguyen et al., 2012; Han et al., 2012; Hindle et al., 2016; Aggarwal et al., 2017).

WE-Based Semantic Extraction With the development of word embedding technologies in the NLP area, more and more researchers also adopt WE techniques to do semantic extraction in their BRM tasks at hand. Existing BRM studies mainly use Word2Vec, GloVe, FastText, etc (Budhiraja et al., 2018a; Jia et al., 2021; Ciborowska and Damevski, 2022). For Word2Vec (supporting skip-gram and CBOW), since existing studies of Word2Vec(Mikolov et al., 2013a,b) showed that the skip-gram model was better than the CBOW, researchers mostly used the skip-gram model for experiments (at the time it was mostly called skip-gram rather than Word2Vec). Yang et al. (Yang et al., 2016) propose an approach that computes and combines two similarity scores based on word embedding vectors by using the skip-gram model and TF-IDF vectors to recommend similar bug reports. Budhiraja et al. also use the skip-gram model to extract semantics for duplicate bug detection task (Budhiraja et al., 2018a,b). With the development of other word embedding techniques besides Word2Vec, some researchers also try to use more word embedding technologies and compare them with traditional technology in their BRM tasks. Cheng et al. (Cheng et al., 2020) applied Wor2Vec to bug localization task and compared it with TF-IDF. Ciborowska el al. (Ciborowska and Damevski, 2022) compare BERT and VSM in bug localization task. Ardimento et al. (Ardimento and Mele, 2020; Ardimento, 2022) used fine-tuned BERT to predict bug fixing time and conducted a comparative experiment between BERT and DistilBERT. Kumar et al. (Kumar et al., 2021) applied CBOW, skip-gram, GloVe, Word2Vec, FastText, BERT and GPT to predict bug severity and also compared it with TF-IDF. Jia et al.(Jia et al., 2021) use FastText to get embedding vectors for the bug report severity prediction task.

To better mine the advantages of word embedding models and neural networks, some researchers propose to combine them to perform their BRM tasks (Sepahvand et al., 2020; Izadi et al., 2022). Xiao et al. (Xiao et al., 2017) and Guo et al. (Guo et al., 2020) use Word2Vec to extract the semantic information of a bug report and then feed the semantic vector to Convolutional Neural Networks (CNN) for bug report management downstream tasks. Deshmukh et al. (Deshmukh et al., 2017) propose a retrieval and classification model using Siamese CNN and Long Short Term Memory (LSTM) for accurate detection and retrieval of duplicate and similar bugs. They mainly use Word2Vec and GloVe to extract semantics from bug reports.

7 Conclusion

In this paper, we conduct an empirical study to explore the potential of pre-trained word embedding models in six BRM tasks. Our whole work aims to answer four research questions, related to the best-performance machine learning algorithm under various word embedding models, the best-performance word embedding model under various machine learning algorithms, the performance difference between word embedding models and the vector space model in semantic extraction, and the value of fine-tuning general pre-trained WE models with bug data. Based on experiments over a set of OSS products, we obtain a list of actionable findings that developers could refer to in choosing suitable techniques for building relevant bug report management models. Our study also indicates that it would be rewarding for researchers to conduct more cross-domain research in BRM tasks and even other software engineering tasks involving textual semantic extraction.

Data availability. The datasets and code scripts for replication are available in the BRMWEStudy repository[12].

Conflict of Interest. The authors declared that they have no conflict of interest.

References

Aggarwal K, Timbers F, Rutgers T, Hindle A, Stroulia E, Greiner R (2017) Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process 29(3):e1821

Alenezi M, Banitaan S (2013) Bug reports prioritization: Which features and classifier to use? In: 2013 12th International Conference on Machine Learning and Applications, IEEE, pp 112–116

Anvik J (2006) Automating bug report assignment. In: Proceedings of the 28th International Conference on Software Engineering, pp 937–940

Ardimento P (2022) Predicting bug-fixing time: Distilbert versus google bert. In: International Conference on Product-Focused Software Process Improvement, Springer, pp 610–620

Ardimento P, Mele C (2020) Using bert to predict bug-fixing time. In: 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems, IEEE, pp 1–7

Arokiam J, Bradbury JS (2020) Automatically predicting bug severity early in the development process. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results, pp 17–20

Bennin KE, Keung JW, Monden A (2019) On the relative value of data resampling approaches for software defect prediction. Empirical Software Engineering 24(2):602–636

Bertram D, Voida A, Greenberg S, Walker R (2010) Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated

---

[12] https://github.com/SurfGitHub/WE4BRM

teams. In: Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, pp 291–300

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp 308–318

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics 5:135–146

Budhiraja A, Dutta K, Reddy R, Shrivastava M (2018a) Dwen: deep word embedding network for duplicate bug report detection in software repositories. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp 193–194

Budhiraja A, Reddy R, Shrivastava M (2018b) Lwe: Lda refined word embeddings for duplicate bug report detection. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp 165–166

Chawla I, Singh SK (2014) Automatic bug labeling using semantic information from lsi. In: 2014 Seventh International Conference on Contemporary Computing, IEEE, pp 376–381

Cheng S, Yan X, Khan AA (2020) A similarity integration method based information retrieval and word embedding in bug localization. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security, IEEE, pp 180–187

Choetkiertikul M, Dam HK, Tran T, Ghose A (2017) Predicting the delay of issues with due dates in software projects. Empirical Software Engineering 22(3):1223–1263

Ciborowska A, Damevski K (2022) Fast changeset-based bug localization with bert. In: Proceedings of the 44th International Conference on Software Engineering, pp 946–957

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. The Journal of Machine learning research 7:1–30

Deshmukh J, Annervaz K, Podder S, Sengupta S, Dubash N (2017) Towards accurate duplicate bug retrieval using deep learning techniques. In: 2017 IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 115–124

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805

Dunn OJ (1961) Multiple comparisons among means. Journal of the American statistical association 56(293):52–64

Giger E, Pinzger M, Gall H (2010) Predicting the fix time of bugs. In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, pp 52–56

Guo S, Zhang X, Yang X, Chen R, Guo C, Li H, Li T (2020) Developer activity motivated bug triaging: via convolutional neural network. Neural Processing Letters 51(3):2589–2606

Habayeb M, Murtaza SS, Miranskyy A, Bener AB (2017) On the use of hidden markov model to predict the time to fix bugs. IEEE Transactions on Software Engineering 44(12):1224–1244

Han D, Zhang C, Fan X, Hindle A, Wong K, Stroulia E (2012) Understanding android fragmentation with topic analysis of vendor-specific bugs. In: 2012 19th Working Conference on Reverse Engineering, IEEE, pp 83–92

Hewett R, Kijsanayothin P (2009) On modeling software defect repair time. Empirical Software Engineering 14:165–186

Hindle A, Alipour A, Stroulia E (2016) A contextual approach towards more accurate duplicate bug report detection and ranking. Empirical Software Engineering 21(2):368–410

Hinton GE, et al. (1986) Learning distributed representations of concepts. In: Proceedings of the Eighth Annual Conference of the Cognitive Science Society, Amherst, MA, vol 1, pp 1–12

Huo D, Ding T, McMillan C, Gethers M (2014) An empirical study of the effects of expert knowledge on bug reports. In: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 1–10

Islam MS, Hamou-Lhadj A, Sabor KK, Hamdaqa M, Cai H (2021) Enhmm: On the use of ensemble hmms and stack traces to predict the reassignment of bug report fields. In: Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, IEEE, pp 411–421

Izadi M, Akbari K, Heydarnoori A (2022) Predicting the objective and priority of issue reports in software repositories. Empirical Software Engineering 27(2):1–37

Jalbert N, Weimer W (2008) Automated duplicate detection for bug tracking systems. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, IEEE, pp 52–61

Jia Y, Chen X, Xu S, Yang G, Cao J (2021) Ekd-bsp: Bug report severity prediction by extracting keywords from description. In: 2021 8th International Conference on Dependable Systems and Their Applications, IEEE, pp 42–53

Kanwal J, Maqbool O (2012) Bug prioritization to facilitate bug report triage. Journal of Computer Science and Technology 27:397–412

Kim D, Tao Y, Kim S, Zeller A (2013) Where should we fix this bug? a two-phase recommendation model. IEEE Transactions on Software Engineering 39(11):1597–1610

Kim J, Yang G (2022) Bug severity prediction algorithm using topic-based feature selection and cnn-lstm algorithm. IEEE Access 10:94643–94651

Kumar L, Kumar M, Murthy LB, Misra S, Kocher V, Padmanabhuni S (2021) An empirical study on application of word embedding techniques for prediction of software defect severity level. In: 2021 16th Conference on Computer Science and Intelligence Systems, IEEE, pp 477–484

Lamkanfi A, Demeyer S (2013) Predicting reassignments of bug reports-an exploratory investigation. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, pp 327–330

Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: 2010 7th IEEE Working Conference on Mining Software Repositories), IEEE, pp 1–10

Lamkanfi A, Demeyer S, Soetens QD, Verdonck T (2011) Comparing mining algorithms for predicting the severity of a reported bug. In: 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, pp 249–258

Liu G, Lu Y, Shi K, Chang J, Wei X (2019) Mapping bug reports to relevant source code files based on the vector space model and word embedding. IEEE Access 7:78870–78881

Lukins SK, Kraft NA, Etzkorn LH (2008) Source code retrieval for bug localization using latent dirichlet allocation. In: 2008 15Th Working Conference on Reverse Engineering, IEEE, pp 155–164

Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psychologica 10(2):545–555

Marks L, Zou Y, Hassan AE (2011) Studying the fix-time for bugs in large open source projects. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, pp 1–8

Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. In: 2008 IEEE International Conference on Software Maintenance, IEEE, pp 346–355

Messaoud MB, Miladi A, Jenhani I, Mkaouer MW, Ghadhab L (2022) Duplicate bug report detection using an attention-based neural language model. IEEE Transactions on Reliability

Mi Q, Keung J, Huo Y, Mensah S (2018) Not all bug reopens are negative: A case study on eclipse bug reports. Information and Software Technology 99:93–97

Mikolov T, Chen K, Corrado G, Dean J (2013a) Efficient estimation of word representations in vector space. arXiv:1301.3781

Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013b) Distributed representations of words and phrases and their compositionality. Advances in Neural Information Processing Systems 26

Nguyen AT, Nguyen TT, Nguyen TN, Lo D, Sun C (2012) Duplicate bug report detection with a combination of information retrieval and topic modeling. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp 70–79

Panjer LD (2007) Predicting eclipse bug lifetimes. In: Fourth international workshop on mining software repositories (MSR'07: ICSE workshops 2007), IEEE, pp 29–29

Pennington J, Socher R, Manning CD (2014) Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing, pp 1532–1543

Peters ME, Neumann M, Iyyer M, Gardner M, Clark C, Lee K, Zettlemoyer L (2018) Deep contextualized word representations. arXiv:1802.05365

Poshyvanyk D, Marcus A, Rajlich V, Gueheneuc YG, Antoniol G (2006) Combining probabilistic ranking and latent semantic indexing for feature identification. In: 14th IEEE International Conference on Program Comprehension, IEEE, pp 137–148

Poshyvanyk D, Gueheneuc YG, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Transactions on Software Engineering 33(6):420–432

Probst P, Boulesteix AL (2018) To tune or not to tune the number of trees in random forest. Journal of Machine Learning Research 18(181):1–18

Probst P, Boulesteix AL, Bischl B (2019a) Tunability: Importance of hyperparameters of machine learning algorithms. Journal of Machine Learning Research 20(53):1–32

Probst P, Wright MN, Boulesteix AL (2019b) Hyperparameters and tuning strategies for random forest. Wiley Interdisciplinary Reviews: data mining and knowledge discovery 9(3):e1301

Ramay WY, Umer Q, Yin XC, Zhu C, Illahi I (2019) Deep neural network-based severity prediction of bug reports. IEEE Access 7:46846–46857

Rodrigues IM, Aloise D, Fernandes ER, Dagenais M (2020) A soft alignment model for bug deduplication. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp 43–53

Roy NKS, Rossi B (2014) Towards an improvement of bug severity classification. In: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE, pp 269–276

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: 29th International Conference on Software Engineering (ICSE'07), IEEE, pp 499–510

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp 345–355

Sahin SE, Tosun A (2019) A conceptual replication on predicting the severity of software vulnerabilities. In: Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering, pp 244–250

Sepahvand R, Akbari R, Hashemi S (2020) Predicting the bug fixing time using word embedding and deep long short term memories. Institution of Engineering and Technology Software 14(3):203–212

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto Ki (2010) Predicting re-opened bugs: A case study on the eclipse project. In: 2010 17th Working Conference on Reverse Engineering, IEEE, pp 249–258

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto Ki (2013) Studying re-opened bugs in open source software. Empirical Software Engineering 18(5):1005–1042

Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume

1, pp 45–54

Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp 253–262

Tagra A, Zhang H, Rajbahadur GK, Hassan AE (2022) Revisiting reopened bugs in open source software systems. Empirical Software Engineering 27(4):92

Tantithamthavorn C, Abebe SL, Hassan AE, Ihara A, Matsumoto K (2018) The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization. Information and Software Technology 102:160–174

Tian Y, Sun C, Lo D (2012) Improved duplicate bug report identification. In: 2012 16th European Conference on Software Maintenance and Reengineering, IEEE, pp 385–390

Tian Y, Lo D, Sun C (2013) Drone: Predicting priority of reported bugs by multi-factor analysis. In: 2013 IEEE International Conference on Software Maintenance, IEEE, pp 200–209

Tian Y, Lo D, Xia X, Sun C (2015) Automated prediction of bug report priority using multi-factor analysis. Empirical Software Engineering 20:1354–1383

Umer Q, Liu H, Illahi I (2019) Cnn-based automatic prioritization of bug reports. IEEE Transactions on Reliability 69(4):1341–1354

Van Nguyen T, Nguyen AT, Phan HD, Nguyen TD, Nguyen TN (2017) Combining word2vec with revised vector space model for better code retrieval. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, IEEE, pp 183–185

Vieira RG, Mattos CLC, Rocha LS, Gomes JPP, Paixão M (2022) The role of bug report evolution in reliable fixing estimation. Empirical Software Engineering 27(7):164

Wang S, Lo D, Lawall J (2014) Compositional vector space models for improved bug localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 171–180

Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th international conference on Software engineering, pp 461–470

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007), IEEE, pp 1–1

Xia X, Lo D, Wang X, Yang X, Li S, Sun J (2013) A comparative study of supervised learning algorithms for re-opened bug prediction. In: 2013 17th European conference on software maintenance and reengineering, IEEE, pp 331–334

Xia X, Lo D, Wen M, Shihab E, Zhou B (2014) An empirical study of bug report field reassignment. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, pp 174–183

Xia X, Lo D, Shihab E, Wang X (2015a) Automated bug report field re-assignment and refinement prediction. IEEE Transactions on Reliability 65(3):1094–1113

Xia X, Lo D, Shihab E, Wang X, Zhou B (2015b) Automatic, high accuracy prediction of reopened bugs. Automated Software Engineering 22:75–109

Xiao Y, Keung J, Mi Q, Bennin KE (2017) Improving bug localization with an enhanced convolutional neural network. In: 2017 24th Asia-Pacific Software Engineering Conference, IEEE, pp 338–347

Xiao Y, Keung J, Bennin KE, Mi Q (2019) Improving bug localization with word embedding and enhanced convolutional neural networks. Information and Software Technology 105:17–29

Yang A, Wang Q, Liu J, Liu K, Lyu Y, Wu H, She Q, Li S (2019) Enhancing pre-trained language representations with rich knowledge for machine reading comprehension. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pp 2346–2357

Yang G, Zhang T, Lee B (2014) Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In: 2014 IEEE 38th Annual Computer Software and Applications Conference, IEEE, pp 97–106

Yang X, Lo D, Xia X, Bao L, Sun J (2016) Combining word embedding with information retrieval to recommend similar bug reports. In: 2016 IEEE 27Th International Symposium on Software Reliability Engineering, IEEE, pp 127–137

Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 689–699

Ye X, Bunescu R, Liu C (2015) Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. IEEE Transactions on Software Engineering 42(4):379–402

Yuan W, Xiong Y, Sun H, Liu X (2021) Incorporating multiple features to predict bug fixing time with neural networks. In: 2021 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 93–103

Zhang H, Gong L, Versteeg S (2013) Predicting bug-fixing time: an empirical study of commercial software projects. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 1042–1051

Zhang T, Chen J, Yang G, Lee B, Luo X (2016) Towards more accurate severity prediction and fixer recommendation of software bugs. Journal of Systems and Software 117:166–184

Zhang T, Yu Y, Mao X, Lu Y, Li Z, Wang H (2022) Fense: A feature-based ensemble modeling approach to cross-project just-in-time defect prediction. Empirical Software Engineering 27(7):1–41

Zhang T, Han D, Vinayakarao V, Irsan IC, Xu B, Thung F, Lo D, Jiang L (2023) Duplicate bug report detection: How far are we? ACM Transactions on Software Engineering and Methodology 32(4):1–32

Zhang W, Challis C (2020) Automatic bug priority prediction using dnn based regression. In: Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery: Volume 1, Springer, pp 333–340

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering, IEEE, pp 14–24

Zhou X, Zhang Y, Cui L, Huang D (2020) Evaluating commonsense in pretrained language models. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 34, pp 9733–9740

Zimmermann T, Nagappan N, Guo PJ, Murphy B (2012) Characterizing and predicting which bugs get reopened. In: 2012 34th International Conference on Software Engineering, IEEE, pp 1074–1083

Zou W, Lo D, Chen Z, Xia X, Feng Y, Xu B (2018) How practitioners perceive automated bug report management techniques. IEEE Transactions on Software Engineering 46(8):836–862

Appendix

The Friedman test results and three CD graphs mentioned in the results part of Section 4: RQ2-(a) are as follows:

Table 12: Friedman tests of FastText and GloVe with different dimension sizes on five BRM classification tasks and the duplicate BR detection task

Five BRM Classification Tasks				
	FastText		**GloVe**	
Task	**Chi-squared Statistic**	**p-value**	**Chi-squared Statistic**	**p-value**
Fixing Time	73.67	1.00E-16	133.99	7.44E-29
Priority	93.32	5.42E-21	138.31	8.72E-30
Reassignment	91.2	1.57E-20	155.54	1.67E-33
Reopen	63.7	1.47E-14	121.91	2.97E-26
Severity	82.42	1.26E-18	141.61	1.69E-30
Duplicate BR Detection				
	FastText		**GloVe**	
Recall-N	**Chi-squared Statistic**	**p-value**	**Chi-squared Statistic**	**p-value**
Recall-1	40	2.06E-09	60	5.87E-13
Recall-5	40	2.06E-09	60	5.87E-13
Recall-10	40	2.06E-09	60	5.87E-13
Recall-15	40	2.06E-09	60	5.87E-13
Recall-20	40	2.06E-09	60	5.87E-13

CD
| 1   2   3 |

('fastText', 300)
('fastText', 200)   ('fastText', 100)

(a) Recall-1

CD
| 1   2   3 |

('fastText', 300)
('fastText', 200)   ('fastText', 100)

(b) Recall-5

CD
| 1   2   3 |

('fastText', 300)
('fastText', 200)   ('fastText', 100)

(c) Recall-10

CD
| 1   2   3 |

('fastText', 300)
('fastText', 200)   ('fastText', 100)

(d) Recall-15

CD
| 1   2   3 |

('fastText', 300)
('fastText', 200)   ('fastText', 100)
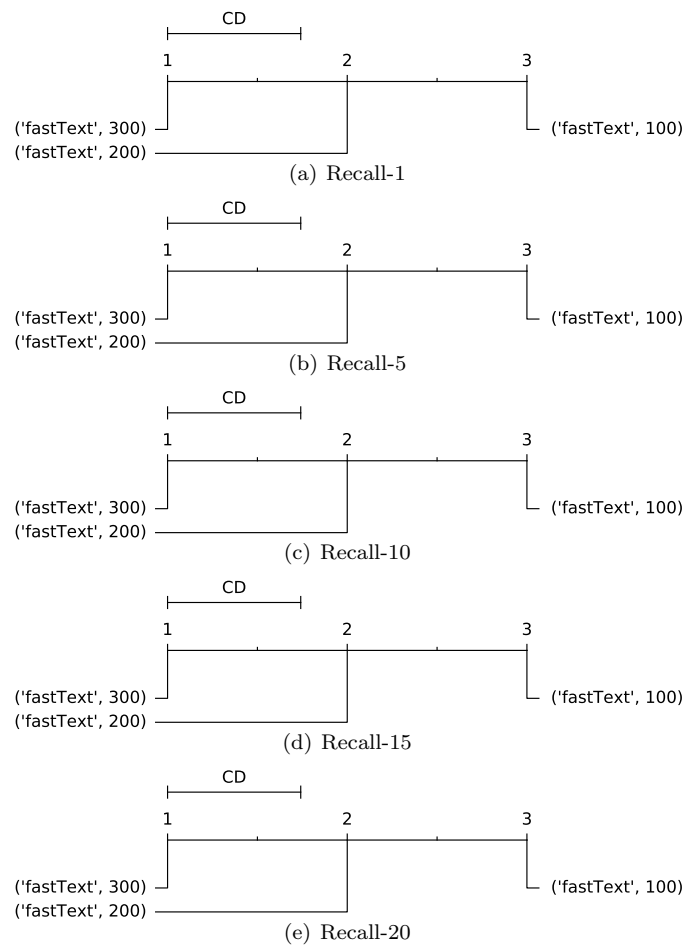
(e) Recall-20

Fig. 11: Comparison of FastText with different dimension sizes on the duplicate BR detection task with the Nemenyi test. Groups of FastText-N that are not significantly different (p-value>0.05) are connected with the bold line(s)
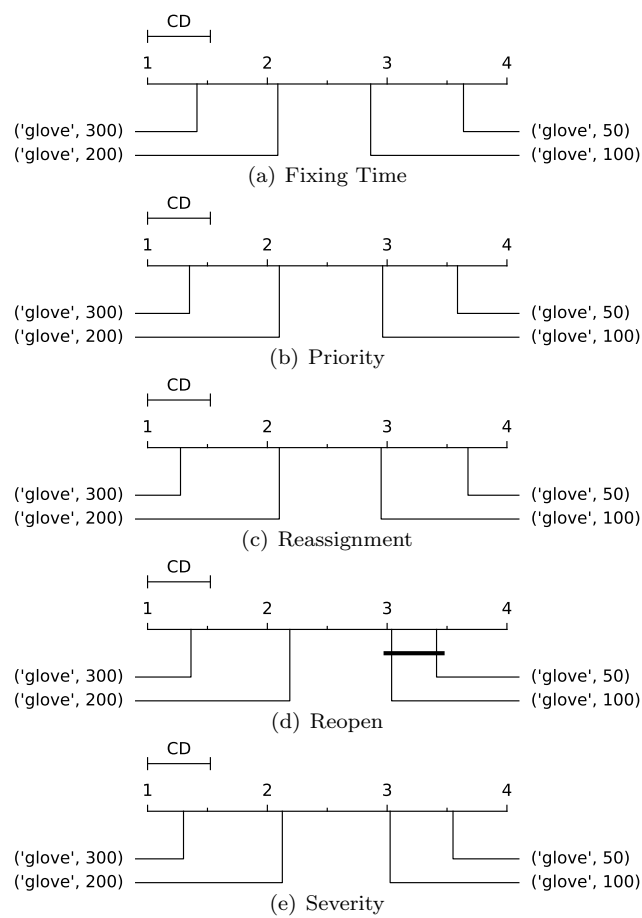
Fig. 12: Comparison of GloVe with different dimension sizes on five BRM classification tasks with the Nemenyi test. Groups of GloVe-N that are not significantly different (p-value>0.05) are connected with the bold line(s)

(a) Recall-1

(b) Recall-5

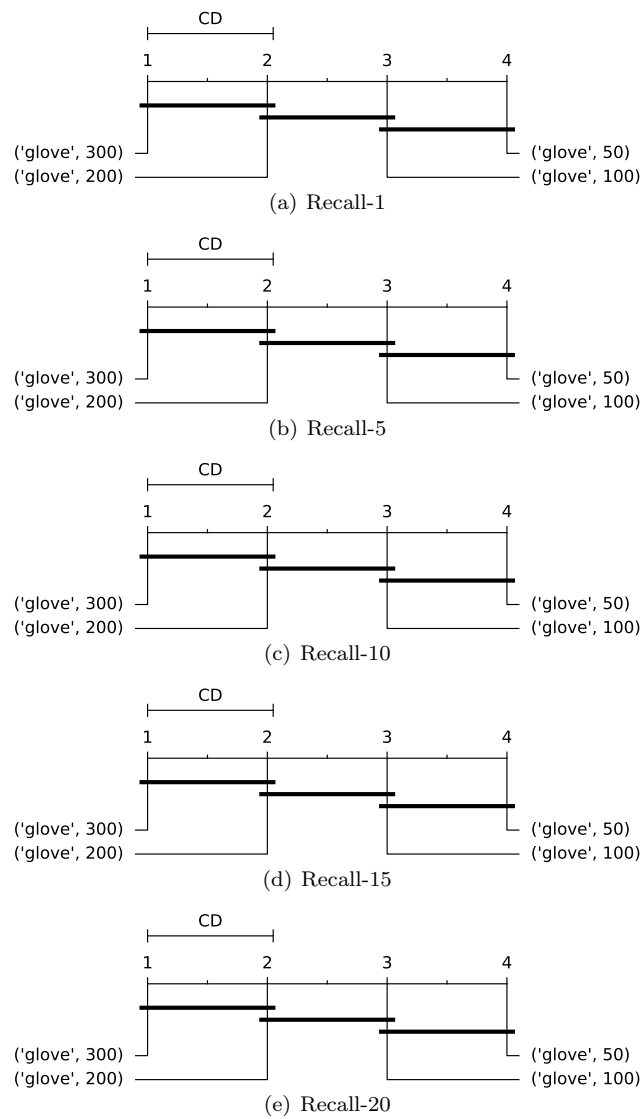(c) Recall-10

(d) Recall-15

(e) Recall-20

Fig. 13: Comparison of GloVe with different dimension sizes on the duplicate BR detection task with the Nemenyi test. Groups of GloVe-N that are not significantly different (p-value>0.05) are connected with the bold line(s)