The Journal of Systems & Software 198 (2023) 111607

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Does Deep Learning improve the performance of duplicate bug report detection? An empirical study☆

Yuan Jiang [a], Xiaohong Su [a,*], Christoph Treude [b], Chao Shang [a], Tiantian Wang [a]

[a] School of Computer Science and technology, Harbin Institute of Technology, Harbin, China
[b] School of Computing and Information Systems, University of Melbourne, Melbourne, VIC, Australia

ARTICLE INFO

Article history:
Received 5 May 2022
Received in revised form 13 November 2022
Accepted 2 January 2023
Available online 6 January 2023

Keywords:
Duplicate bug report detection
Deep learning
Information retrieval
Similarity measure
Realistic evaluation

ABSTRACT

Do Deep Learning (DL) techniques actually help to improve the performance of duplicate bug report detection? Prior studies suggest that they do, if the duplicate bug report detection task is treated as a binary classification problem. However, in realistic scenarios, the task is often viewed as a ranking problem, which predicts potential duplicate bug reports by ranking based on similarities with existing historical bug reports. There is little empirical evidence to support that DL can be effectively applied to detect duplicate bug reports in the ranking scenario. Therefore, in this paper, we investigate whether well-known DL-based methods outperform classic information retrieval (IR) based methods on the duplicate bug report detection task. In addition, we argue that both IR- and DL-based methods suffer from incompletely evaluating the similarity between bug reports, resulting in the loss of important information. To address this problem, we propose a new method that combines IR and DL techniques to compute textual similarity more comprehensively. Our experimental results show that the DL-based method itself does not yield high performance compared to IR-based methods. However, our proposed combined method improves on the MAP metric of classic IR-based methods by a median of 7.09%–11.34% and a maximum of 17.228%–28.97%.

1. Introduction

With the increasing scale and complexity of software systems, bugs are highly prevalent. In order to improve the reliability and maintainability of software systems, bug tracking systems (BTSs) have been widely adopted to track bug-related information in software products. Although BTSs are of great benefit to software developers, they also encounter problems, such as duplicate bug reports from different bug reporters pointing to the same bug. Assigning duplicate bug reports to different developers to fix is not only a waste of resources but also introduces inconsistencies when independent developers fix the same bug asynchronously.

To solve this problem, triagers need to manually identify and label whether the new bug report is a duplicate of an already reported bug. However, this process is tedious and time-consuming. According to our statistics, during the three-year period from January 1, 2018 to January 1, 2021, a total of 7542 bug reports are reported to Eclipse BTS, of which 39% are duplicates and invalid. Therefore, manually identifying duplicate bug reports is a challenging task for triggers.

Previous studies have shown that automatically retrieving historical bug reports that are similar to the newly reported ones can speed up the detection process of duplicate bug reports by narrowing the search space of possible candidates. Therefore, most existing methods make use of information retrieval (IR) techniques for the automatic detection of duplicate bug reports (Sun et al., 2010, 2011; Wang et al., 2008; Jalbert and Weimer, 2008; Lin and Yang, 2014; Hindle et al., 2016), which construct a query for a given bug report and treat all historical bug reports in BTS as a collection of documents, and then calculate the relevance score between query and each document to retrieve duplicate candidates. In this context, the relevance score is defined as the likelihood of two bug reports being duplicates of each other. All historical bug reports are ranked by their relevance to the query bug report, and bug reports with higher relevance are closer to the top of the ranked list. Although these IR-based methods have proven very useful in detecting duplicate bug reports with satisfactory results, they only exploit lexical similarity as relevance score, and ignore semantic similarity. Wang et al. (2008) first mentioned that using lexical similarity in duplicate bug report detection cannot match phrases with the same meaning but different expressions, such as "retain" and "still running". Therefore, researchers have argued that it is difficult to determine whether

☆ Editor: Dr Earl Barr.

* Corresponding author.
E-mail addresses: jiangyuan@hit.edu.cn (Y. Jiang), sxh@hit.edu.cn (X. Su), christoph.treude@unimelb.edu.au (C. Treude), 19S103131@stu.hit.edu.cn (C. Shang), wangtiantian@hit.edu.cn (T. Wang).

https://doi.org/10.1016/j.jss.2023.111607
0164-1212/

Table 1
Bug reports 90421 and 91188 from OpenOffice.

Field	Bug report 1	Bug report 2
Bug_id	90421	91188
Summary	In vertical writing Japanese fonts get invisible on Ubuntu 8.04	Writer will not show Chinese Vertical Characters in correct direction
Description	In vertical writing Japanese fonts disappear or get invisible with OpenOffice.org 2.4 and 3.0 Beta on Ubuntu 8.04.	Vertical Chinese characters were not correctly displayed. 1. Characters were not displayed. 2. After press "BOLD", characters displayed but 90 degrees clockwise. You can check on youtube. http://www.youtube.com/watch?v=m4MMNv_pjNA
Product	Writer	Writer
Component	code	Viewing
Op_sys	Linux	Linux
Version	OOo 2.4.0	OOo 2.4.0
Priority	P3	P3
Severity	Trivial	Trivial

a pair of bug reports is duplicate by using lexical similarity alone (He et al., 2020). As an example, we use two duplicate bug reports in the OpenOffice project as shown in Table 1 to explain the problem from a practical point of view.

From Table 1, we can observe: (1) these two bug reports implicitly represent the same bug, i.e., some vertical characters/fonts were not displayed correctly in the Writer; (2) the textual fields (i.e., *summary* and *description*) of the two bug reports are almost a complete lexical mismatch, i.e., sharing very few words with each other; (3) some synonyms or phrases, such as "disappear" → "not displayed" in *Bug Report 1* and *2* respectively reflect semantically near-identical content; (4) the categorical fields of bug reports such as product, component, priority and version, are the same between the two duplicate bug reports.

We believe that an automatic method should be able to more precisely detect duplicate bug reports by taking into account the semantic relations between bug reports. A few DL-based methods (e.g., Deshmukh et al. (2017), He et al. (2020), Budhiraja et al. (2018a,b) and Poddar et al. (2019)) have been proposed to extract semantic features from bug reports, and then use a similarity layer (e.g., a full-connection neural network) to compute semantic similarity between two bug reports and suggest possible duplicates. However, does DL actually improve the performance of duplicate bug report detection? The conclusions are less clear from the experiments carried out in related work, due to the following reasons: (1) the evaluation datasets used are too small to cover the entire lifecycle of the studied BTSs, which is not enough to prove that the proposed method achieves a significantly superior performance compared to classic IR-based methods. (2) most existing DL-based methods are evaluated on their ability to predict whether a pair of bug reports is duplicate, which may not be representative of their practical performance in a ranking scenario, i.e., ranking all possible duplicates of a given bug report. (3) since most existing methods are evaluated on small-scale datasets, duplicates are often removed due to their master bug report falling outside the evaluated time intervals. Thus, it is still unclear how well DL-based methods perform without ignoring any bug reports.

Due to the above limitations of previous work, this paper performs an empirical study to comprehensively evaluate whether DL-based methods offer distinct advantages in duplicate bug report detection. For this purpose, we evaluate according to two aspects: (1) examining the performance of DL-based methods in isolation; (2) examining the performance of a mixed method combining DL and IR techniques. To the best of our knowledge, we are the first to study the performance of the combined method on the domain of duplicate bug report detection, which is one of the main contributions of this work. In addition, to ensure the rationality and objectivity of the experimental results, we select three large real-world datasets covering the entire lifecycle of the corresponding projects to study the performance differences of different methods. This is another important contribution of this work.

To verify whether DL models can achieve good performance by themselves for duplicate bug report detection, we first design two deep learning models (i.e., siamese and triplet networks) based on different textual feature extractors to model the semantic relations between bug reports. Then, we experimentally evaluate the performance of the proposed models on the duplicate bug report detection task, and compare them with the state-of-the-art DL-based detection methods on three large-scale, open-source software projects with more than 1,000,000 bug reports in total. Finally, we compare the DL-based model, which yields the maximum performance in the last step, with IR-based methods to demonstrate its effectiveness in the ranking scenario. The main advantage of the above setting is that it allows us to clearly distill (1) which DL-based model exhibits superior performance in detecting duplicate bug reports when evaluated on complete datasets, and (2) whether the best-performing DL-based model outperforms classic IR-based methods when applied to a ranking scenario that is closer to that realistic scenario.

However, no matter whether the lexical or semantic similarity calculated by IR or DL-based methods respectively, it only measures the relevance of pairs of bug reports from one aspect or one level. Concretely, lexical similarity evaluates the degree of term-based matching, while semantic similarity measures the relatedness in the semantic content. In addition, as shown in Table 1, categorical similarity also plays an important role in accurately evaluating the relevance between two bug reports, as demonstrated empirically by Rakha et al. (2017). Therefore, in this paper, we also aim to demonstrate whether combining these three kinds of similarities can more comprehensively characterize the semantic relatedness between bug reports. To this end, we try to learn a ranking function to combine those three types of features (lexical similarity, semantic similarity, and categorical similarity) in a linear combination fashion and compute the final score of a given bug report with respect to each historical bug report. The weights in the ranking function can be learned automatically based on previously solved bug reports using a learning to rank technology. Experimental results on three large datasets demonstrate that semantic similarity provides complementary information that further improves ranking performance when compared to popular IR-based method.

The main contributions of this paper include:

- We investigate which DL-based model is more appropriate for extracting semantic features from textual fields of bug reports. The experimental results show that BERT is better for duplicate bug report detection.
- We investigate which feature is the most important in the duplicate bug report detection task. Empirical studies on large-scale datasets show that "lexical similarity" is

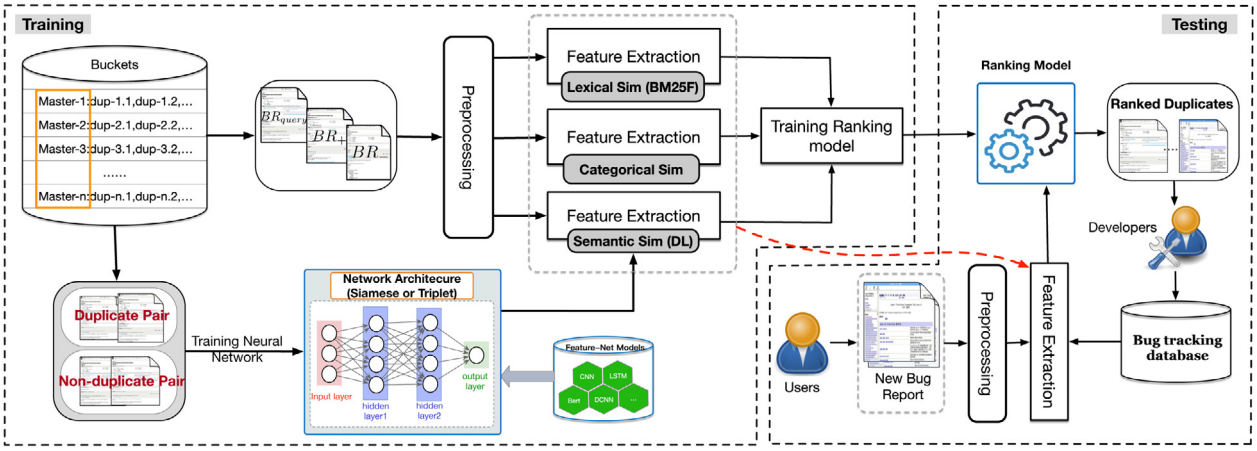

Fig. 1. The general framework for detecting duplicate bug reports in the ranking scenario.

more important than "semantic similarity" in determining whether a pair of bug reports are duplicate or not.

- We investigate whether or not combining these three types of similarities can characterize the semantic relatedness between bug reports in a more comprehensive way. Experimental results show that using all features works better than using each type of feature individually.
- We conduct extensive experiments on benchmark datasets and our newly collected datasets from three projects to study the performance of different methods. Since the bug reports in our datasets span the entire lifecycle of projects, the main conclusions that drawn from the experiments can represent real-life/practical situations.

The paper is organized as follows. Section 2 presents our empirical study methods. Section 3.2 describes our empirical study data. Section 3 presents our empirical study setup. In Section 4, we conduct extensive experiments and provide experimental results. Section 5 presents a discussion, and Section 6 introduces related work. Section 7 offers the conclusion and future work.

2. Empirical study methods

As described in Section 1, DL-based approaches address the task of duplicate bug report detection in two ways: (1) they make decisions directly on each pair of bug reports to detect possible duplicate bug reports (He et al., 2020), and (2) they first use a deep neural network to learn a semantic vector representation for each bug report, and then rank all historical bug reports according to their similarity scores with the new bug report to detect possible duplicate bug reports (Deshmukh et al., 2017). The first way can be formalized as a binary classification task, where the input is a pair of bug reports and the output is a similarity score to indicate whether the input bug reports are duplicates of each other. The detailed detection process of DL-based methods in the classification scenario is depicted in Section 2.1.3. The second way can be viewed as a ranking problem where the goal is to return a rank list of historical bug reports that may have reported the same bug as the given input bug report. Since DL-based models can be used alone or in combination with IR-based methods in such a ranking problem, we propose a general framework as shown in Fig. 1 to explain how DL-based methods work in the ranking scenario. As can be seen in Fig. 1, the proposed framework can be further divided into two main phases: training and testing. During the training phase, the model is built based on historical bug reports with known ground truth. During the testing phase, the trained model is used to rank possible duplicate bug reports with newly-submitted bug reports.

For training, we first organize the bug reports in the bug tracking database into a list of buckets which is a kind of data structure similar to hash-map (Sun et al., 2010). There is only one master bug report per bucket, which is the first one reported, and the others are all duplicate with the master bug report. Second, we extract training samples in the form of $\{BR_{query}, BR^+, BR^-\}$ triplets from buckets where BR_{query} refers to a query bug report, BR^+ is duplicate with BR_{query}, and BR^- is non-duplicate with BR_{query}. Third, for each bug report in triplets, we collect and preprocess its textual information (i.e., *summary* and *description*) as well as the categorical information (e.g., product, component, version and type). Fourth, various types of similarities between two bug reports are calculated, which serve as features for duplicate bug report detection. Finally, based on these features, a ranking model is trained to compute the global matching score between a pair of bug reports with the ranking based objective, aiming at ranking positive pairs (i.e., BR_{query} and BR^+) higher than negative pairs (i.e., BR_{query} and BR^-).

For testing, given a newly-submitted bug report, we first apply the same preprocessing and feature extraction in the training phase, and then feed the calculated similarities into the trained ranking model to predict the relevance score between the new bug report and each historical bug report in the bug tracking database. Next, all historical bug reports are ranked in descending order of predicted scores, with the expectation that the most likely duplicate ones (with the new bug report) are closer to the top of the list. Finally, the triager manually examines the ranked list to determine if the new bug report has already been reported. If so, the triager needs to label the new bug report as a duplicate and link it to the bug report describing the same bug issue and reported by other users.

2.1. Studied features

In total, we calculate 9 similarities between a pair of bug reports, which can be used as features for duplicate bug report detection. These similarities can be divided into three unique dimensions: lexical similarity, semantic similarity and categorical similarity. Table 2 summarizes the similarities of different categories. The reasons for using these similarities as features to detect duplicate bug reports are as follows:

- Lexical and categorical similarities have been widely adopted in prior studies, which describe whether a pair of bug reports are duplicates of each other from different perspectives.

Table 2
Studied similarities of bug reports.

Dimension	Notation	Definition	Rationale
Lexical similarity	$L_{unigram}$	Lexical similarity between the *summary* or *description* field of two bug reports based on unigram model	Duplicate bug reports tend to be represented with the same terms
	L_{bigram}	Lexical similarity between the *summary* or *description* field of two bug reports based on bigram model	
Categorical similarity	$Prod_{sim}$	Categorical similarity between the product field of two bug reports	The issue described by duplicate bug reports tend to occur in the same product
	$Comp_{sim}$	Categorical similarity between the component field of two bug reports	The issue described by duplicate bug reports tend to occur in the same component
	$Type_{sim}$	Categorical similarity between the type field of two bug reports	The issue described by duplicate bug reports tend to have the same issue type
	$Prio_{sim}$	Categorical similarity between the priority field of two bug reports	The issue described by duplicate bug reports tend to have the same priority
	$Vers_{sim}$	Categorical similarity between the version field of two bug reports	The issue described by duplicate bug reports tend to occur in the same version of software
Semantic similarity	S_{ss}	Semantic similarity between the *summary* field of two bug reports	Duplicate bug reports tend to represent the same semantics in the *summary* field
	S_{dd}	Semantic similarity between the *description* field of two bug reports	Duplicate bug reports tend to represent the same semantics in the *description* field

- Semantic similarity learned by deep learning techniques is often used to estimate how well the semantic content of a document matches the input query. We conjecture that semantic similarity also plays an important role in determining whether a bug report is a duplicate by providing an accurate measure of bug semantics.

Semantic similarity is mainly calculated by DL-based methods, which is the focus of this paper. Lexical and categorical similarities are mainly calculated by IR-based methods, e.g., $BM25F_{ext}$ and REP, which will be detailed in the latter part of this section. The reasons for using IR-based methods are that (1) they are popular and representative in the field of duplicate bug report detection; (2) their effectiveness in real scenarios has empirically been demonstrated in previous work (Rakha et al., 2017; Sun et al., 2011). Therefore, employing these IR-based methods as classical baselines enables convincing comparisons with DL-based methods. In addition to the above methods, some other techniques, e.g., machine learning and topic model, have also been applied to detect duplicate bug reports. However, previous work (Rakha et al., 2017) has empirically found that most of these methods are not publicly available or too complex to be implemented, which brings difficult to make a fair comparison in this paper. Nonetheless, this does not affect our comparison studies as our aim is to demonstrate the effectiveness of DL-based methods.

2.1.1. Lexical similarity

Lexical similarity is one of the most important dimensions for predicting duplicate bug reports. This is because the duplicate bug report detection task can often be transformed into an IR problem where lexical similarity is used as a metric to determine the degree of duplication of a pair of bug reports based on their textual descriptions. To measure lexical similarity, many ranking models (e.g. Boolean model and Vector Space Model (VSM) Manning et al., 2008, Liu (2011)) based on probabilistic ranking principles have been proposed in the field of IR and have been effectively applied in practice. One of the most well-known probabilistic ranking models is *BM25F* (Hermawati et al., 2017), which allows the use of machine learning algorithms to learn the weights of different fields or terms in each field, resulting in more accurate text similarity between bug reports. Although *BM25F* model is a strong baseline for traditional IR tasks and seems a natural fit for duplicate bug report detection, one of its major drawbacks is that it considers only the importance of different terms in the document and not the terms in the query. The reason for such designs is that in IR tasks, the query usually consists of several keywords without repeated terms. However, for duplicate bug report detection, the query is a bug report that contains a long text *summary* and *description*. Therefore, following previous work (Sun et al., 2011; Jiang et al., 2020), we use a new variant of *BM25F* (i.e. $BM25F_{ext}$) to calculate the lexical similarity between two bug reports by considering the weights of terms in both query and documents. The calculation formulas of $BM25F_{ext}$ are as follows:

$$IDF(t) = log\frac{N}{N_{dt}} \tag{1}$$

$$TF_D(d,t) = \sum_{f=1}^{K} \frac{weight_f * occurrences(d[f],t)}{1 - b_f + \frac{b_f * length_f}{average\_length_f}} \tag{2}$$

$$TF_Q(q,t) = \sum_{f=1}^{K} weight_f * occurrences(q[f],t) \tag{3}$$

$$BM25F_{ext} = \sum_{t \in q \cap d} IDF(t) * \frac{TF_D(d,t)}{k_1 + TF_D(d,t)} * \frac{(k_3+1) * TF_Q(q,t)}{k_3 + TF_Q(q,t)} \tag{4}$$

where N represents the total number of bug reports, N_{dt} represents the number of bug reports containing a given term t, $weight_f$ is the field weight, b_f is the parameter that determines the scaling of the field length, $d[f]$ represents the bag of terms in the fth field of bug report d, $length_f$ is the size of $d[f]$, $average\_length_f$ is the average size of all $d[f]$ of all bug reports, K represents the number of textual fields and its default value is 2 which represents two textual fields used in duplicate bug report detection, k_1 and k_3 are the scaling parameters to control the values of $TF_D(d,t)$ and $TF_Q(q,t)$ respectively, $occurrences(d[f],t)$ is the frequency of a term t that appears in field f of bug report d. *IDF* (Inverse Document Frequency) is a global weight of a given term across all bug reports, which is the logarithmic of N divided by N_d. $TF_D(d,t)$ represents the frequency of the term t appearing in historical bug reports. $TF_Q(q,t)$ is an extension of BM25F, which represents the frequency of the term t appearing in a query bug report q.

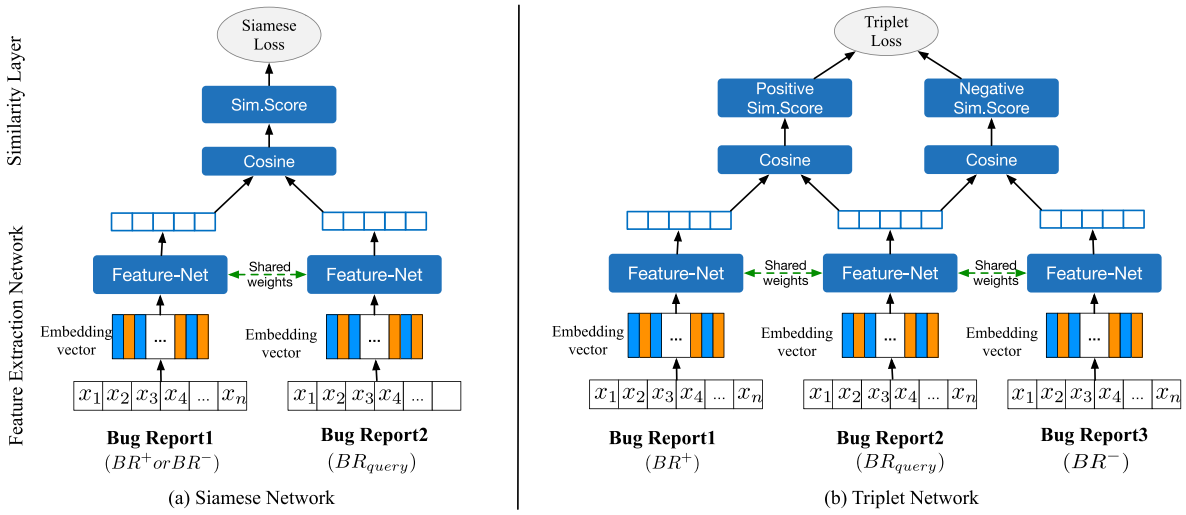

Fig. 2. The architecture of the DL-based duplicate bug report detection framework.

The above formulas calculated based on unigrams yield the first lexical similarity feature, i.e., $L_{unigram}$. If we represent bug reports using bigrams, and then use the same formulas to calculate the similarity between bug reports, we can get another lexical similarity feature, i.e., L_{bigram}.

2.1.2. Categorical similarity

Actually, duplicate bug reports are not only similar in the textual field, but also in some categorical fields including *product*, *component*, *type*, *priority*, *version* etc. Therefore, in addition to textual similarity, categorical similarity, which is calculated based on non-textual information (i.e., categorical fields) of bug reports, is also widely used for duplicate bug report detection (Sun et al., 2011).

As shown in Table 2, there are five categorical similarities involved. $Prod_{sim}$ compares if two bug reports are in the same *product*. In a bug report, the *product* field stores the product affected by the reported bug. The value of $Prod_{sim}$ is 1 if two bug reports belong to the same product, and 0 otherwise.

$Comp_{sim}$ determines whether two bug reports are in the same *component*. In a bug report, the *component* field specifies in which architecture layer or in which sub-module within an architecture layer the reported bug occurred (Alipour et al., 2013). The value of $Comp_{sim}$ is 1 if two bug reports have the same component, and 0 otherwise.

$Type_{sim}$ compares the *type* of two bug reports. If two bug reports have the same bug type, they are more likely to be duplicates. Similarly, we set the $Type_{sim}$ to 1 if equality holds for the type field of two bug reports, and 0 otherwise.

$Prio_{sim}$ and $Vers_{sim}$ compare the differences in the priority and version fields of two bug reports respectively. Since the values of these two fields are continuous, following Sun et al. (2011), the reciprocal of the distance between the *priority* or *version* fields of two bug reports is assigned to the corresponding categorical similarity (i.e., $Prio_{sim}$ or $Vers_{sim}$).

The calculation formulas of the above categorical similarities are as follows:

$$Prod_{sim} = \begin{cases} 1, & if\ d.prod = q.prod \\ 0, & otherwise \end{cases} \tag{5}$$

$$Comp_{sim} = \begin{cases} 1, & if\ d.comp = q.comp \\ 0, & otherwise \end{cases} \tag{6}$$

$$Type_{sim} = \begin{cases} 1, & if\ d.type = q.type \\ 0, & otherwise \end{cases} \tag{7}$$

$$Prio_{sim} = \frac{1}{1 + |d.prio - q.prio|} \tag{8}$$

$$Vers_{sim} = \frac{1}{1 + |d.vers - q.vers|} \tag{9}$$

2.1.3. Semantic similarity

Lexical similarity is commonly used in traditional IR tasks based on explicit text matching (Jones et al., 2000a,b). However, merely using the lexical similarity between two bug reports is insufficient to comprehend their semantic relationships. To bridge the semantic gap, deep learning has been introduced in the field of duplicate bug report detection to measure the semantic similarity of bug reports. A major benefit of using deep learning is that it can automatically extract semantic features from input bug reports without relying on hand-engineering features based on expert domain knowledge. In this paper, we argue that lexical and semantic similarities compensate each other in textual similarity measures and hence ensembling these two similarities can more comprehensively capture the semantic relatedness between bug reports.

Since semantic similarity measures the relatedness in the embedded semantic space (Bengio et al., 2003; Mikolov et al., 2013a; Fan et al., 2017), we first need to map the textual *description* of bug reports into this semantic space and then calculate semantic similarities between these vectors. Inspired by the widespread success of existing DL-based embedding methods, in this paper, we present a general DL-based framework for learning deep semantic representation of bug reports, as shown in Fig. 2, which mainly consists of the following components.

- ***Feature-Net*** transforms any bug reports into feature embeddings for high-level representations.
- ***Similarity Layer*** evaluates the probability that two bug reports are duplicates by measuring the similarity of their vector representations.
- The framework offers two different ways (i.e., one is ***siamese*** and another is ***triplet*** network architecture) to detect duplicate bug reports using similarity scores of pairs of bug reports.

For a newly submitted bug report, we denote it as a query (BR_{query}). The goal of DL-based methods is to detect whether the BR_{query} is duplicate or non-duplicate in the classification or ranking scenario. In both cases, the Feature-Net plays an important role in the duplicate bug report detection since it affects the

quality of extracting reliable information from bug reports. Thus, Feature-Net should be able to generate low-dimensional vectors while preserving the semantics of bug reports. In this study, we employed a broad range of neural networks as the candidate Feature-Net, including convolutional networks (i.e., CNN LeCun et al., 1998) and recurrent neural networks (e.g., BiLSTM Hochreiter and Schmidhuber, 1997), which are popular models widely used in NLP tasks within a decade. Besides, with the development of transformer structures, various transformer based models (e.g., BERT (Bidirectional Encoder Representations from Transformers) Devlin et al., 2018) have recently been proposed to model natural language texts. There is plenty of evidence that BERT exhibits strong performance on many NLP tasks. Therefore, we also chose BERT as a candidate for Feature-Net, and compared it to LSTM or CNN based networks and existing DL-based models (e.g., DCNN He et al., 2020 and Deshmukh et al., 2017).

Our key insight is that by comparing the similarities of pairs of bug reports, we can train a siamese or triplet network (Kaya and Bilge, 2019). The trained network not only acts as a predictor to determine whether two bug reports are duplicates of each other in the classification scenario, but also acts as a feature extractor to produce the representations required for the semantic similarity calculation in the ranking scenario.

As illustrated in Fig. 2, the siamese network is built on two Feature-Nets, whereas the triplet network is built on three Feature-Nets. In a siamese or triplet network, different Feature-Nets share the exact same architecture and parameters. That is, any changes to one Feature-Net also have an impact on the others. For each Feature-Net, it takes the word2vec embeddings of all tokens contained in a bug report as input and outputs a feature representation $f(BR, \theta_f)$ defined by different network f, where θ_f denotes the network parameters. Taking the CNN network as an example, $f(BR, \theta_f)$ can be formalized as follows:

$$f(BR, \theta_f) = f_L \circ ... f_2 \circ f_1(BR) \tag{10}$$

where f_i represents a composition function, which consists of four cascade-operations: convolution operation, a batch normalization, an activation operation (such as ReLU) and a pooling operation. L is the number of convolutional layers of CNN.

A similarity layer on top of Feature-Nets is used to measure the pairwise similarity score between bug reports based on their representation. In metric learning, there are three most commonly used ways to understand the similarity relationship between samples (Kaya and Bilge, 2019; Nguyen et al., 2018). Specifically, fully-connected layers, as used in Wang et al. (2014), support accurate calculation of similarity scores but incur high computational costs. Euclidean distance (Schroff et al., 2015) is another widely used similarity function that satisfies speed requirements but fails to deal effectively with high-dimensional sparse data. In our framework, we employ cosine similarity, for its cost-effective, robust and good performance (Xie et al., 2018), as the similarity function in the similarity layer to construct a low-dimensional target space (Nguyen et al., 2018). Given a pair of bug reports BR_i and BR_j, the similarity function $g(f(BR_i), f(BR_j))$ defined by $g : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$, is computed as follows:

$$g(f(BR_i), f(BR_j)) = \frac{f(BR_i) \cdot f(BR_j)}{\|f(BR_i)\| \left\|f(BR_j)\right\|} \tag{11}$$

The function g calculates the similarity between two bug reports, ranging from 0 to 1. If the input is a pair of duplicate reports, g returns a large value, called positive score g_+. Ideally, the cosine similarity of a pair of duplicate bug reports should be 1. Conversely, the similarity of a non-duplicate pair calculated by g is smaller, called negative score g_-.

Since model architectures of siamese and triplet networks are different, two specialized training strategies are developed to learn them respectively. The training of the triplet network is performed by feeding triplet examples, each of which includes three instances: $\{BR_{query}, BR^+, BR^-\}$ where BR^+ is a positive bug report that is duplicate with BR_{query}, and BR^- is a negative bug report that is different from BR_{query} in the reported bug. The training process of the siamese network is similar to that of the triplet network, differing in that siamese takes as input a pair of bug reports (i.e., $\{BR_{query}, BR^+\}$ or $\{BR_{query}, BR^-\}$) rather than a triplet example.

The hinge loss is used as the loss function for training the triplet network, and the parameters θ_f of Feature-Net are learned by maximizing the hinge-loss between the positive score g_+ and the negative score g_-:

$$L = max(0, m + g_- - g_+) \tag{12}$$

where m is a margin. Instead of setting a threshold to determine whether or not the measured pairs of bug reports are duplicates, we utilize the hinge-loss function to ensure that positive scores are always greater than negative scores with a certain margin m.

The cross-entropy loss is used as the loss function for training the siamese network, and the parameters θ_f of Feature-Net are learned by maximizing the similarity value for a duplicate pair and minimizing it for a non-duplicate pair:

$$L = -ylog(v) - (1 - y)log(1 - v) \tag{13}$$

where v is the similarity score computed by the siamese network for a pair of input bug reports, and $y \in \{0, +1\}$ is the ground-truth label, i.e., $+1$ for duplicate pairs and 0 for non-duplicate pairs.

Due to the different importance of *summary* and *description* on duplicate bug report detection, each textual field is handled separately as most previous approaches did (Deshmukh et al., 2017; Rakha et al., 2017). We denote the feature embedding of *summary* and *description* in BR_{query} and BR_{doc} (i.e., BR^+ or BR^-) as V_{qs}, V_{qd}, V_{ds}, V_{dd} respectively, which are all produced by the trained Feature-Net. Taking the semantic similarity between V_{qs} and V_{ds} as an example, it is calculated as follows:

$$S_{ss} = \frac{V_{qs} \cdot V_{ds}}{\left\|V_{qs}\right\| \left\|V_{ds}\right\|} \tag{14}$$

2.2. Approach

In this empirical study, we consider three types of similarities: lexical similarity, categorical similarity and semantic similarity, which characterize the relatedness between bug reports from three different perspectives. Then, based on these computed similarities, we learn a global score function to determine whether or not the newly submitted bug report is duplicate. Previous research (Sun et al., 2011; Rakha et al., 2017) has shown that combining both lexical and categorical similarities can improve detection performance when compared to utilizing only one similarity measure. However, there is not enough evidence to indicate that combining semantic similarity with other traditional similarities will further improve the performance of the detection model over long periods of time with a significant margin. Therefore, in this paper, we aim to demonstrate whether or not combining these three types of similarities can characterize the semantic relatedness between bug reports in a more comprehensive way. To this end, we follow the idea that is used in Sun et al. (2011), where the author combines different duplicate scores (i.e., similarities) using a supervised learning to rank approach. The reason for employing learning to rank is that it uses machine learning algorithms to build effective ranking score functions from training data, and has been widely used in the past duplicate bug report

detection studies (Liu, 2011). The formula to compute the global similarity score ($Dup_{score}(d, q)$) between bug reports is as follows:

$$Dup_{score}(d, q) = \sum_{i=1}^{9} w_i * sim_i \quad (15)$$

where w_i is the weight of the ith similarity value Sim_i, which can be learned from the training data with known duplicated labels. Once the w_i is fixed, the model is learned, and can be used for detection.

Although there are three types of similarities in total as shown in Table 2, each of them can be used standalone as the feature of the ranking model to calculate the global similarity score by Eq. (15). For example, some of the previous studies are based on the lexical similarities (e.g., *BM25F* Sun et al., 2011; Rakha et al., 2017, $BM25F_{ext}$ Sun et al., 2011) or the combination of lexical and categorical similarities (e.g., REP Sun et al., 2011; Rakha et al., 2017), whereas other recent work focuses on leveraging semantic similarities for duplicate bug report detection (e.g., DL-based methods Deshmukh et al., 2017; He et al., 2020). Unlike previous work that only considers one type of similarity, our work is the first to explore the combinations of three types of similarities to boost detection performance compared to both IR- and DL-based methods. For convenience, we refer to the proposed method as CombineIRDL throughout this paper.

3. Empirical study setup

3.1. Research questions

The main research questions (RQ) focused in this paper are as follows:

RQ1: How does the performance of the DL-based approaches on benchmark datasets with different scales? Most existing approaches are evaluated on small-scale datasets with bug reports collected from a short period of time. This type of evaluation (called *classical evaluation* in the literature Rakha et al., 2017) tends to overstate model performance in a real-world production setting due to the assumption that the time-interval between duplicate bug reports and their master report is short. Therefore, in this RQ, we spontaneously wonder whether or not such a phenomenon occur during the evaluation process of DL-based methods. To answer this, we first construct two datasets of different sizes for each studied BTSs and then assess the performance of DL-based methods on different sized datasets.

RQ2: How well do the DL-based approaches perform compared to existing popular IR-based approach? The DL-based approaches address the duplicate bug report detection task in two ways: (1) they make decisions for any pair of bug reports to indicate whether they are duplicates (Lin et al., 2016), and (2) they first use deep neural networks to learn from the textual fields of bug reports to generate low-dimensional semantic vector representations, and then compare representations of bug reports to identify possible duplicate bug reports that are similar to a new bug report. The first way can be formalized as a binary classification via DL which is analyzed in RQ1, while the second way can be formulated as a ranking problem that is closer to the realistic situation. In this RQ, we aim to study whether the state-of-the-art DL-based approaches perform better than other popular IR-based methods by comparing their ranking performance?

RQ3: How does combining DL with IR impact the performance of the duplicate bug report detection task? In RQ3, we investigate whether it is possible to further improve the performance of detection methods by combining DL with IR techniques. The reasons for this combination are two-fold: (1) Deep semantic information is crucial for accurately evaluating the textual similarity between two bug reports, and DL has been proven to be very effective at extracting such information from bug reports; (2) The lexical and categorical features calculated by IR-based methods also have been previously shown to achieve good performance for finding duplicate bug reports.

RQ4: How effective is the approach of combining IR and DL when applied to three large unpublished datasets over recent time periods? Three benchmark datasets from 1998 to 2010 (11 years before this study) are widely used in previous studies to validate the effectiveness of various detection methods. Even though the experimental results on these datasets illustrate the potential of the IR-based and DL-based approaches for solving the duplicate bug report detection task, we still have no insight into whether these approaches lead to desired performance on the reported bug reports that occurred in recent time periods. Therefore, additional empirical studies are needed to answer the above question. In RQ4, we first expand the benchmark datasets by collecting additional bug reports in BTSs of three open-source projects from January 1, 2011 to December 31, 2020, and then perform extensive experiments on the new datasets to evaluate and compare the performance of IR-based baseline and the proposed combined approach CombineIRDL.

3.2. Studied datasets

3.2.1. Benchmark DataSets

Experiments are conducted on three datasets collected from OpenOffice,[1] Eclipse,[2] and Mozilla,[3] where OpenOffice is a popular software system while Eclipse and Mozilla are two popular software foundations. The Mozilla Foundation is a free software community that uses, develops, spreads and supports Mozilla products, such as Firefox, Thunderbird and Bugzilla. The Eclipse Foundation is an open-source organization, and is home to over 375 open-source projects, including runtimes, tools, and frameworks for a wide range of technical fields. OpenOffice is a discontinued open-source office suits, owned by Apache Foundation, which provides a word processor, a spreadsheet, a presentation application and a drawing application for office purposes.

Although the datasets collected from the above sources are widely used in prior work, no standardized data size (or time period) exists for their evaluation. In this study, the datasets we adopt as benchmarks are the ones contributed by Rakha et al. (2017). The reasons we choose the datasets are threefold. (1) All three datasets span a period over decades, ensuring that a large amounts of bug reports can be used for method evaluation; (2) The datasets of these projects are also adopted by previous studies, which allows for more convenient comparisons between different methods; and (3) All three datasets are from well-known and large-scale open-source projects, which are statistically representative and relevant to the real-world situations. The statistical information of all the datasets is detailed in Table 3, where "Time Range" is the time range for the bug reports, "# of Reports" is the total number of bug reports and "# of Duplicates" is the number of duplicated bug reports in the dataset. As

[1] https://www.openoffice.org/.
[2] https://www.eclipse.org/.
[3] https://foundation.mozilla.org/en/.

Table 3
The statistics of the benchmark datasets.

Project	Time Range	# of Reports	# of Duplicates
Eclipse	10/10/01–12/31/10	332,600	40,232
Mozilla	04/07/98–12/31/10	552,451	135,163
OpenOffice	10/16/00–12/31/10	116,043	19,119

shown in Table 3, these datasets contain a total of 1,001,094 bugs reported from April 7, 1998 to December 31, 2010. Among the 1,001,094 bugs, 194,514 are duplicates.

3.2.2. Newly collected datasets

As can be seen from Table 3, most bug reports are submitted before December 31, 2010, which may be too old to simulate the present situation as the development organization and process have changed significantly over the years. To further verify the effectiveness of different methods over the most recent periods, we first conduct data collection to augment benchmarks and then report the experimental results of different methods on the new datasets. For data collection, we first implement a web crawler to collect all bug reports from January 1, 2011 up to December 31, 2020 in OpenOffice, Eclipse and Mozilla BTSs. Then, we store each collected bug report as an XML file, which has a set of fields such as *bug_id*, *summary*, *description*, *product*, *component*, *op_sys*, *version*, *priority*, *severity* and others. Finally, if the resolution field of the bug report is "*DUPLICATE*" it will be labeled as duplicate and non-duplicate otherwise. More details about the newly collected datasets can be found in Section 4.4. All these Datasets for Eclipse, OpenOffice and Mozilla are freely and openly available to the public via the link (https://sites.google.com/view/dbddl).

3.2.3. Bug report preprocessing

As textual fields of bug reports, including *summary* and *description*, may contain some irrelevant information for duplicate bug report detection, we apply the following preprocessing steps to clean up the textual fields. First, we tokenize the text into a list of terms. Second, we remove URLs, stack trace, hex code, and non-alphabetic characters. Finally, we convert the remaining terms to lower-case and then transform them to their root forms using NLTK Porter Stemmer. Considering the tremendous number of bug reports and for saving the query and computational time, we adopt the distributed search engine Elasticsearch (ES) to manage and store the preprocessed bug reports in our datasets.

3.3. Performance evaluation

3.3.1. Evaluation metrics

In this section, we include the most widely-used metrics, i.e., precision, recall, F1-measure, *Recall Rate@k* and MAP to measure the performance of the different approaches for the duplicate bug report detection task.

When evaluating DL-based methods in the classification scenario, we use well-known metrics including precision, recall and F1-measure to measure and compare their overall performance on different datasets, which is consistent with previous work (Deshmukh et al., 2017; He et al., 2020). $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F1\text{-}measure = 2 * \frac{precision*recall}{precision+recall}$, where TP is true positives, FP is false positives, FN is false negatives and TN is true negatives.

When treating the duplicate bug report detection task as a ranking problem, i.e., searching historical bug reports to find the original master report for a given bug report, we use the same two evaluation metrics as Rakha et al. (2017) to compare the ranking performance of different methods including IR-based, DL-based and the hybrid method (i.e., the proposed CombineIRDL). The two metrics are defined as follows.

MAP: The mean of the average precision for all queries, which is the most commonly used IR metric for evaluating the power of retrieval approaches (Youm et al., 2017; Akbar and Kak, 2019). MAP is important if a developer is interested in finding the true master report by reading the returned list one by one in descending order of similarity (Wang and Lo, 2016). A higher MAP value indicates more effective of the retrieval approach. The original formula is defined as follows:

$$MAP = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{|K|}\sum_{k\in K} Prec@k(Q_q) \tag{16}$$

where Q is the set of query bug reports, K is the set of positions of the relevant bug reports in the rank list, and $Prec@k(Q_q)$ is the precision of relevance at position k in the returned bug report list. The definition of $Prec@k(Q_q)$ is:

$$Prec@k(Q_q) = \frac{\#\ the\ relevant\ docs\ for\ Q_q\ in\ topk}{k} \tag{17}$$

Due to the specialized nature of the duplicate bug report detection task, i.e., only one master report relevant to a given query bug report, $|K|$ is equal to 1, and $k \in |K|$ is the index of the right master bug report in the candidate list. Thus, *MAP* can be simplified as:

$$MAP = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{k} \tag{18}$$

Recall Rate@k: The percentage of bug reports whose master can be detected in the returned bug report list. Following previous work (Rakha et al., 2017), the value of k is set to 5 or 10. Unlike MAP, *Recall Rate@k* only considers the top-k ranked results when evaluating retrieval approaches. It is also important since prior studies (Kochhar et al., 2016; Huo et al., 2019) has shown that more than 95% of practitioners only check the top 10 results.

$$Recall\ Rate@k = \frac{N_{detected}}{N_{total}} \tag{19}$$

where $N_{detected}$ is the number of bug reports whose master can be successfully detected in the *top-k* candidate reports, and N_{total} is the number of query bug reports.

3.3.2. Statistical tests

Most previous work conducts experiments on datasets belonging to three projects (Mozilla, Eclipse and OpenOffice) collected over a specific time period. For example, the testing periods of the three projects in the literature (Sun et al., 2011) range from 1 to 2 years. However, only performing evaluation over a period of time may introduce randomness or bias, and hence some results could not be generalized to the whole lifetime of a studied BTS. To better demonstrate whether the comparisons between the two classes of approaches vary significantly with the time period, we follow recent work (Rakha et al., 2017) to first randomly select 100 chunks of data that cover the lifetime of each dataset, and then iteratively conduct experiments on each chunk of data to collect a set of evaluation results in terms of various metrics (i.e., *Recall Rate@k* and *MAP*). Finally, we carry out a statistical analysis of the experimental results using *Mann–Whitney U* test to verify the statistical significance of our conclusions. The reason for using the *Mann–Whitney U* test is that, it is a non-parametric statistical test approach that does not require the population to follow a normal distribution, but only require samples to be randomly selected, and observation values are independent. For example, when performing comparisons between IR-based and DL-based methods, we define the following hypotheses:

Table 4
Configuration search space of various deep learning models.

Hyper-parameter	Value	Description
Embedding size (Word2Vec)	[60, 120, 200]	The embedding size of the Word2vec is set to 60, 120 or 200
Widths of convolutional filters (CNN, DC-CNN He et al., 2020)	[(1,2,3), (2,3,4), (3,4,5)]	Each tuple, e.g., (1, 2, 3), represents that we use three types of convolutional filters whose widths are 1, 2 and 3
The number of filters (CNN, DC-CNN He et al., 2020)	[50, 100, 150, 200]	The number of filters for CNN is set to 50, 100, 150 or 200
RNN layer (BiLSTM, Deshmukh et al. (2017))	[1, 2]	The number of layers for RNN is set to 1 or 2.
RNN hidden size (BiLSTM, Deshmukh et al. (2017))	[50, 100, 150, 200]	The hidden size in each RNN unit is set to 50, 100, 150 or 200.
RNN cell number (BiLSTM, Deshmukh et al. (2017))	[50, 100, 150, 200]	The number of cells for each RNN layer is set to 50, 100, 150 or 200.
Initial learning rate	[2e−02, 2e−03, 2e−04, 2e−05]	The initial learning rate is set to 2e−02, 2e−03, 2e−04, 2e−05
Mini-batches of size	[8, 16, 32, 64, 128]	The network is trained with mini-batches of 8, 16, 32, 64 or 128.
Epoch	[5, 15, 20, 30, 50]	The epochs for training networks are set to 5, 15, 20, 30 or 50

- H_0: The IR-based approaches and DL-based approaches have no significant performance difference.
- H_1: The IR-based approaches and DL-based approaches have significant differences in performance.

Under the null hypothesis and 5% level of significance, our specification test will reject the null hypothesis (i.e., H_0) with a probability (i.e., p-value) that, in the limit, does not exceed the significance level of the test. Additionally, cliff's delta is also calculated to measure the probability that the values from one group of the evaluation metrics $X = \{x_1, x_2, ..., x_m\}$ are greater than the other $Y = \{y_1, y_2, \ldots, y_n\}$, which can be used as *effect size* to quantify the difference between two distributions. The definition of cliff's delta is as follows:

$$\delta(i,j) = \begin{cases} +1, & x_i > y_j \\ -1, & x_i < y_j \\ 0, & x_i = y_j \end{cases} \tag{20}$$

$$\delta = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}\delta(i,j) \tag{21}$$

The formula (20) is used to calculate the difference between a pair of metric values which are sampled from two different groups, and the formula (21) is used to calculate the *effect size* by averaging over $m * n$ results obtained from formula (20). Obviously, the value of δ is between -1 and $+1$, and when it is close to $+1$ or -1, indicating a more significant group difference in the evaluation metric; otherwise, when δ is close to 0, the difference between two groups is not significant.

Moreover, we use the kurtosis (Joanes and Gill, 1998) measure to determine whether two or more detection approaches have a stable performance on datasets over different periods of time. The kurtosis is often used to explain the peakedness of a population. The higher the kurtosis, the more agreement that samples in the distribution have on the average of the distribution. Specifically, all the normal distributions have the same kurtosis, i.e., 3. The formulas for calculating kurtosis used in this study are provided by Cramir (1946), which can be defined as follows:

$$G_2 = \frac{K_4}{K_2^2} = \frac{n-1}{(n-2)(n-3)}\{(n+1)g_2 + 6\}$$

where

$$\begin{aligned} K_4 &= \frac{n^2\{(n+1)m_4 - 3(n-1)m_2^2\}}{(n-1)(n-2)(n-3)} \\ K_2 &= \frac{n}{n-1}m_2 \\ g_2 &= \frac{m_4}{m_2^2} - 3 \\ m_r &= \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^r \end{aligned} \tag{22}$$

where m_r is the r-order sample moment of samples of size n, m_4 is the fourth sample moment, m_2 is the second sample moment, x_i is the ith value in the set of samples, $\bar{x}$ is the sample mean, n is the total number of samples.

3.4. Hyperparameter adjustment

Since most models are sensitive to hyperparameter values, a proper hyperparameter tuning is required to yield good performance on benchmark datasets. Existing methods for duplicate bug report detection can be roughly divided into two categories, namely DL-based methods and IR-based methods. For DL-based methods, we follow the previous work by Guo et al. (2017) to tune the hyperparameters of deep neural networks by grid searching on the validation dataset. Specifically, we first identify optimal or near-optimal hyperparameters that may lead to the desired performance, based on the default values of the original paper and practical observation. Next, we design a search space around the identified values such that only the most probable optimal values of hyperparameters are included in the space. Finally, we perform a grid search on the search space and pick the best performing model with a good set of hyperparameters on the validation set. Table 4 describes the search space we used to tune deep neural networks.

To reproduce the experimental value of this paper, we offer a default configuration of parameters that are likely to work out of the box. The embedding of words are trained on training datasets using Word2Vec (Mikolov et al., 2013b) with Skip-gram algorithm and the embedding size is set to 200. The number of hidden layers for each RNN is set to 2, and each hidden layer dimension is set to 200. The number of convolutional filters used is 150 with kernel size of Akbar and Kak (2019), Alduailij and Al-Duailej (2015) and Alipour et al. (2013). All of the activation functions in these models are set to tanh function. As for the Bert model, we use the default architecture and hyperparameter settings (i.e., 12-layer, 768-hidden, 12-heads, 110M parameters) in order to reuse their pre-trained models (Devlin et al., 2018). In addition, we train DL models for a total of 30 epochs using the Adam optimizer (Kingma and Ba, 2014) with a learning rate of 0.002, a dropout of 0.5 and a batch size of 64. Our approach is implemented with the Pytorch library in Python with an Intel(R) Core i7@3.2 GHz, 192 GB DDR4-RAM and 1 GeForce RTX 2080 GPU.

The automated IR-based methods retrieve duplicate candidates from BTSs for query bug reports through a scoring function Dup_{score} that assigns each historical bug report a similarity score to represent their likelihood of being duplicates. The higher the value of similarity, the higher the probability of the query bug report and the historical bug report being duplicates. RankNet

is often used for learning the parameters of the scoring function (Burges et al., 2005). The simplified version of RankNet cost function RNC proposed by Taylor et al. (2006) is as follows:

$$RNC(I) = log(1 + e^{Y}) \tag{23}$$

$$Y = Dup_{score}(BR_{query}, BR^{-}) - Dup_{score}(BR_{query}, BR^{+}) \tag{24}$$

where $Dup_{score}(BR_{query}, BR^{+})$ refers to the score of a pair of duplicate bug reports, and $Dup_{score}(BR_{query}, BR^{-})$ corresponds to the score of a pair of non-duplicate bug reports. The parameters in Dup_{score} are optimized by using the RankNet method, which mainly include feature weights (i.e., $w_1 \sim w_9$) in Eq. (15) and the parameters (i.e., $weight_f$, b_f, k_3 and k_1) of the $BM25F_{ext}$ in Eqs. (2)∼(4).

3.5. Validation settings

To evaluate the ranking performance of different models, we followed the same evaluation protocol with Rakha et al. (2017), which conducts performance evaluation in a realistic setup where no data is ignored, making the performance comparison fair. Realistic evaluation differs from classical evaluation in that classical evaluation ignores the bug reports that are submitted before the evaluated time period, while realistic evaluation does not. In addition, we randomly select 100 different chunks of data from each studied dataset and further divide each chunk into tuning and testing data. Specific details about the implementation are presented in the later sections (i.e., Sections 5.2∼5.3). As a result, there are 100 effectiveness values for each studied BTS, and we perform a statistical significance test to determine whether the differences in performance measures between different methods are statistically significant.

However, if we only compare the performance among DL-based models, we can use a simpler strategy to evaluate the performance of these models, which does not require randomly selecting 100 chunks from each studied dataset and conducting extensive experiments on each chunk. This is because a deep learning model with a huge number of parameters requires a large amount of data for adequately training. Generally, hyperparameter optimization and model training come with huge costs (time and resources). Therefore, we partitioned the studied dataset into mutually exclusive training and testing sets with a proportion of 80% and 20%, which is a common practice in the deep learning literature.

4. Results

4.1. Answer question 1: How does the performance of the DL-based approaches on benchmark datasets with different scales?

Motivation. When evaluating IR-based models on small-scale datasets, their performance may be overestimated, as demonstrated in the literature (Rakha et al., 2017). However, we are unaware of whether such problems hold in the evaluation of DL-based methods. This is mainly because DL-based methods focus on predicting whether or not a pair of bug reports are duplicates, which differs from IR-based methods by ranking all possible duplicates of the given bug reports. Therefore, in this RQ, we evaluate the performance of previous DL-based methods on datasets of different scales, and demonstrate whether detection models trained on large-scale datasets have statistically significant differences in evaluation metrics when compared to models trained on small-scale datasets. In addition, we are interested in which model is more effective for extracting semantic features from bug reports, considering that the performance of DL-based methods depends on its ability to extract deep semantic features.

Approach. To answer this question, we conduct the following experiment. First, for each project, we construct two datasets of different sizes, the smaller dataset, which has the same evaluated time periods as previous work (Sun et al., 2011), and the larger dataset, which is large enough to fully cover the life cycle of each studied BTS under evaluation. Each instance in datasets is represented by a triplet in the form of (BR_{query}, BR^{+}, BR^{-}). Second, we build two deep network frameworks (i.e., siamese and triplet networks) based on various textual feature extractors for duplicate bug report detection, and evaluate these models using multiple evaluation metrics. Finally, we perform statistical testing on the experimental results to analyze whether there are statistically significant differences in evaluation metrics when training detection models on benchmark datasets with different scales.

In this study, we use three commonly used deep learning models BiLSTM, CNN and BERT as Feature-Nets to extract the semantic features from bug reports, and construct a triplet or siamese network for duplicate bug report detection. As a result, we can obtain three baselines:

- *BiLSTM or CNN-based Detection Model.* CNN and LSTM are the two main types of neural networks, both of which can automatically extract features instead of manually designed features (Lee et al., 2017). In particular, CNN can extract local features through convolutional filters and perform well on semantic parsing tasks in NLP (Kim, 2014), whereas LSTM can capture long-range dependencies across sequences by incorporating gating mechanisms, thereby outperforming traditional supervised learning methods on serialization tasks in NLP. Considering that CNN and LSTM are state-of-the-art semantic feature extractors and have been widely explored to handle various NLP tasks (Yin et al., 2017), we examine the performance differences of the two alternative networks in the duplicate bug report detection task.
- *BERT-based Detection Model.* BERT leverages a multi-layer multi-head self-attention together with a positional word embedding to construct an effective text representation, which has achieved great success in many NLP tasks such as text classification over the past few years (Devlin et al., 2018; Lu et al., 2020). Since the semantic features of bug reports are mainly derived from the textual fields of bug reports and BERT has strong semantic extraction ability in natural language processing, we design a method to integrate the triplet or siamese framework and BERT for duplicate bug report detection, which can be served as a strong baseline in our comparison experiments.

Different networks (i.e., triplet and Siamese networks) have different learning processes depending on the number of Feature-Nets. Specifically, for the triplet network, triplets in the form of (BR_{query}, BR^{+}, BR^{-}) sampled from the constructed databases are employed as the input. However, for the siamese network, a pair of bug reports, i.e., (BR_{query}, BR^{+}) or (BR_{query}, BR^{-}), are taken as input, which is constructed by removing BR^{-} or BR^{+} from the triplet (BR_{query}, BR^{+}, BR^{-}). The total number of triples and pairs extracted from three datasets are shown in Table 5.

In addition, we also compare the performance of the above baselines with those proposed by Deshmukh et al. (2017) and He et al. (2020). The details of these state-of-the-art methods are as follows:

- *BiLSTM+CNN+MLP based Detection Model.* The model is a state-of-the-art DL-based duplicate bug report detection approach proposed by Deshmukh et al. (2017), in which BiLSTM and CNN are used to extract semantic features from

Table 5
The statistics of the constructed datasets used for evaluating DL-based models.

	Small scale dataset			Large scale dataset		
	Time Range	# of triplets	# of pairs	Time Range	# of triplets	# of pairs
Eclipse	01/01/08–12/31/08	8762	17,524	10/01/01–12/31/08	79,948	159,896
Mozilla	01/01/10–12/31/10	17,367	34,734	04/07/98–12/31/10	246,282	492,560
OpenOffice	01/01/08–12/31/10	9181	18,362	10/16/00–12/31/10	38,225	76,450

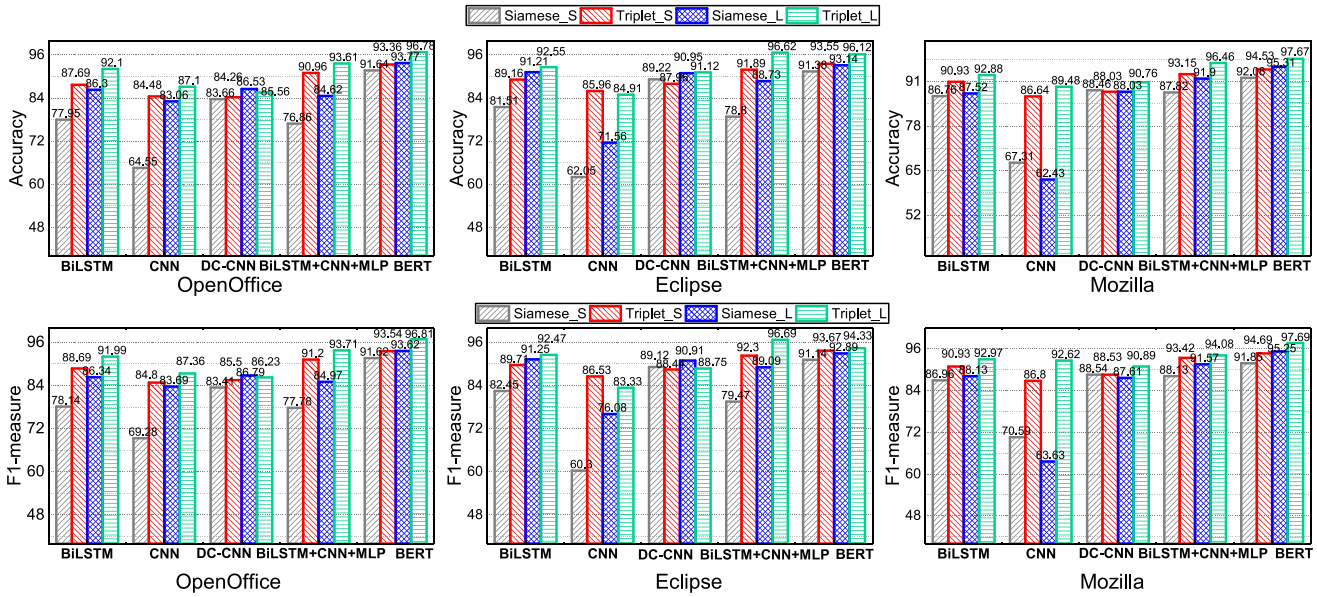

Fig. 3. Accuracy and F1-measure results of different feature-Nets trained in siamese and triplet forms.

summary and *description* of bug reports, respectively, and MLP is adopted to encode categorical information to accurately and comprehensively represent the bug reports. Since it is one of the most representative work in the field of research on duplicate bug report detection, we adopt it as a baseline detection method.

- *Dual-Channel Convolutional Neural Network (DC-CNN) based Detection Model.* DC-CNN is a type of CNN that is used to extract features from the original input in the form of the dual-channel matrix. Given a pair of bug reports, a new dual-channel matrix can be constructed based on the single-channel metrics of two bug reports. Therefore, the DC-CNN can naturally act as the feature extractor to capture the correlated semantic relationships between bug reports by convolving on the dual-channel matrix. In our experiments, we adopt DC-CNN as a baseline as it has proved to be more effective and efficient than the CNN in determining whether or not a pair of bug reports are duplicates.

Results. Table 6 presents the classification results of the three baselines and the two state-of-the-art methods, in which each row is the result of a specific combination of Feature-Net (Model) and learning architecture (Learning Framework), the bold indicates those with the highest value of metrics for each project under different sized datasets. To compare and analyze the effectiveness of all DL-based models more intuitively, we report the accuracy and F1-measure of three projects with large- and small-scale datasets in histogram form as shown in Fig. 3, where *Siamese_S* and *Siamese_L* represent models trained in the siamese way on small and large datasets respectively, while *Triplet_S* and *Triplet_L* represent models trained in the triplet way on small and large datasets respectively. From Table 6 and Fig. 3, we can obtain some notable conclusions:

(1) For different projects, the highest F1-measure is achieved by models trained on large-scale datasets. A paired *Mann–Whitney U* test further confirms that the detection model performs significantly better (p-value < 0.05) on large-scale datasets than on small-scale datasets. This is evident from the fact that most current deep neural networks with a large number of learning parameters require a large amounts of manually labeled data to produce satisfactory results. Therefore, models trained on large-scale datasets tends to perform better than those trained on small-scale datasets.

(2) For OpenOffice and Mozilla, BERT-based models provides the best prediction results, with F1-measure of 96.81% and 97.69%, respectively, on two projects with large scale datasets. This is due to the fact that BERT is able to capture complex contextual information using multiple bidirectional self-attention modules. However, for the Eclipse project, Deshmukh et al.'s approach (i.e., BiLSTM+CNN+MLP) performs the best, slightly outperforming BERT-based models on the F1-measure.

(3) Our results show that the BiLSTM method performs better than the CNN method, which indicates the BiLSTM is more effective than CNN in capturing bug semantics from textual fields of bug reports. A possible explanation for this phenomenon is that the long-term global dependencies of the text learned by the hidden states of BiLSTM help to uncover the meaning of bug reports more accurately.

(4) Interestingly, Deshmukh et al.'s approach (i.e., BiLSTM + CNN + MLP) achieves stronger results than the state-of-the-art method DC-CNN. This is not consistent with the findings reported in the original work (He et al., 2020). This is because, in their original study, DC-CNN is only compared to the BiLSTM+CNN+MLP model trained in the siamese form. However, as we found by experiments, most models exhibit varying degrees of performance improvement by adopting the triplet training form. As a consequence, DC-CNN may outperform the BiLSTM+CNN+MLP method

Table 6
Experimental results of DL-based models trained in siamese and triplet forms.

Project	Model	Learning Framework	Small scale dataset Acc	Prec	Recall	F1-measure	Large scale dataset Acc	Prec	Recall	F1-measure
OpenOffice	BiLSTM	Siamese	77.95	77.43	78.87	78.14	86.30	86.08	86.61	86.34
		Triplet	87.69	82.04	**96.51**	88.69	92.10	93.36	90.66	91.99
	CNN	Siamese	64.55	61.12	79.96	69.28	83.06	80.70	86.92	83.69
		Triplet	84.48	83.07	86.60	84.80	87.10	85.62	89.17	87.36
	DC-CNN	Siamese	83.66	84.72	82.14	83.41	86.53	85.13	88.52	86.79
		Triplet	84.26	79.26	92.81	85.50	85.56	82.41	90.43	86.23
	BiLSTM+CNN+MLP	Siamese	76.86	74.77	81.05	77.78	84.62	83.09	86.95	84.97
		Triplet	90.96	88.84	93.68	91.20	93.61	92.29	95.17	93.71
	BERT	Siamese	91.64	**91.84**	91.39	91.62	93.77	95.90	91.45	93.62
		Triplet	**93.36**	90.95	96.30	**93.54**	**96.78**	**96.08**	**97.54**	**96.81**
Eclipse	BiLSTM	Siamese	81.51	78.45	86.87	82.45	91.21	90.86	91.64	91.25
		Triplet	89.16	85.36	94.52	89.71	92.55	93.35	91.62	92.47
	CNN	Siamese	62.05	63.20	57.65	60.30	71.56	65.65	90.44	76.08
		Triplet	85.96	83.16	90.18	86.53	84.91	93.11	75.41	83.33
	DC-CNN	Siamese	89.22	89.90	88.36	89.12	90.95	91.32	90.50	90.91
		Triplet	87.96	88.44	87.33	87.88	91.12	88.75	94.18	91.38
	BiLSTM+CNN+MLP	Siamese	78.80	77.02	82.08	79.47	88.73	86.35	92.01	89.09
		Triplet	91.89	87.91	**97.15**	92.30	**96.62**	94.70	**98.76**	**96.69**
	BERT	Siamese	91.38	**93.73**	88.70	91.14	93.14	**96.32**	89.70	92.89
		Triplet	**93.55**	91.97	95.43	**93.67**	96.12	91.94	96.85	94.33
Mozilla	BiLSTM	Siamese	86.76	85.65	88.31	86.96	87.52	83.99	92.70	88.13
		Triplet	90.93	90.96	90.90	90.93	92.88	91.82	94.14	92.97
	CNN	Siamese	67.31	64.15	78.46	70.59	62.43	61.65	65.75	63.63
		Triplet	86.64	85.82	87.80	86.80	89.48	85.93	94.42	89.98
	DC-CNN	Siamese	88.46	87.91	89.17	88.54	88.03	90.83	84.61	87.61
		Triplet	88.03	84.93	92.46	88.53	90.76	89.69	92.11	90.89
	BiLSTM+CNN+MLP	Siamese	87.82	85.94	90.44	88.13	91.90	90.63	92.52	91.57
		Triplet	93.15	89.85	97.29	93.42	97.07	95.89	96.13	94.08
	BERT	Siamese	92.08	**94.57**	89.29	91.85	95.31	96.57	93.96	95.25
		Triplet	**94.53**	91.97	**97.58**	**94.69**	**97.67**	**96.96**	**98.42**	**97.69**

when trained in the siamese form, and vice versa when trained in the triplet form.

When duplicate bug report detection is treated as a binary classification task, the experimental results show that deep learning models perform well on three studied BTSs, especially on large scale datasets. Additionally, among the compared models, BERT achieves the overall best performance as it can better capture the deep semantics embedded in the textual fields of bug reports

4.2. Answer question 2: How well do the DL-based approaches perform compared to existing popular IR-based approach?

Motivation. Most existing DL-based methods are only evaluated in the classification scenario such as our experiments in Section 4.1, which may not represent their practical performance in the ranking scenario. Although there are a few studies (Deshmukh et al., 2017) that attempt to evaluate their solutions in both scenarios, almost none of them use the full dataset of the studied BTSs. Therefore, there is no clear evidence that DL-based methods are significantly better than the classical IR-based methods. To fill this gap, in this RQ, we study whether DL-based approaches outperform other popular IR-based methods by comparing their ranking performance.

Approach. For performance comparison with IR-based baselines on benchmark datasets, we follow the realistic evaluation settings in Rakha et al. (2017) to first randomly select 100 different chunks of data from each studied BTS and then divide each chunk into tuning and testing data. Here the tuning data is actually the training data in the sense that it is used to train a ranking model. We call it tuning data in order to be consistent with previous work (Rakha et al., 2017). The bug reports within a single chunk span a period of one year. Thus, there may be some overlap between different chunks. For example, the chunks Feb 2, 2009 to Feb 2, 2010 and Feb 3, 2009 to Feb 3, 2010 are considered as two different chunks (Rakha et al., 2017). Since the start times of 100 chunks across the lifecycle of each BTS are randomly generated, this guarantees that our experimental evaluations cover the whole lifespan of each BTS. The data structure of each chunk for tuning and testing IR-based methods is shown in Fig. 4.

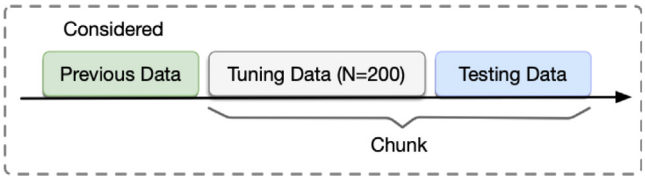

Fig. 4. Data organization of each chunk to tune and test IR-based methods.

As seen in Fig. 4, all the parameters described in formula (4) and (15) are tuned only on the tuning dataset consisting of 200 duplicate bug reports. The considered duplicates in the testing set are those that are marked as duplicates and have corresponding masters in previously reported bug reports. The reasons for using a certain number of bug reports as tuning data in this section are as follows: (1) According to previous research results (Rakha et al., 2017), the detection performance of IR-based models is not significantly affected by the choice of tuning data, especially for OpenOffice and Eclipse projects; (2) when the tuning data contains a certain number of bug reports, the comparative experimental results of different methods throughout this paper

are not affected by the choice of tuning data; (3) previous research (Rakha et al., 2017) has shown that, for different projects, the IR-based detection models can achieve the best performance in most cases when the number of tuning dataset is fixed to $N = 200$.

Various IR-based approaches employ IR techniques to calculate the similarity between the newly-submitted bug report and all historical bug reports in BTS. The most successful IR-based method is REP, which provides better relevant results by making full use of textual and categorical information via learning a ranking model. In addition, REP has publicly available implementation whose effectiveness has empirically been demonstrated in previous work (Rakha et al., 2017). Therefore, we adopt REP as a baseline for fair comparison studies. Lexical and categorical similarities in REP are calculated using the same equations described in Sections 2.1.1 and 2.1.2, respectively. Besides REP, we also compare the performance of the DL-based methods with the existing advanced document similarity approach $BM25F_{ext}$. The reason for using $BM25F_{ext}$ as a baseline is that it only uses lexical similarities in detecting duplicate bug reports, so we can easily determine which feature (i.e., lexical similarity or semantic similarity) is better by comparing it with DL-based methods in the ranking scenario.

Since, as we stated in RQ1, there are multiple DL-based methods for duplicate bug report detection, comparing IR-based approaches with each DL-based model is a complex and time-consuming task, especially when the datasets are very large and each dataset is further divided into 100 chunks. Therefore, we choose the DL-based model (i.e., BERT) that performs best in terms of F1-measure across different projects in the classification scenario as the semantic generator to produce a semantic vector for each historical bug report.

To evaluate BERT in a realistic setup, we need to train the model with historical bug reports. However, the tuning dataset is relatively small, with only 200 duplicate bug reports. Although we observe from Rakha et al. (2017) that a size of 200 is sufficient to tune the weights of ranking models in IR-based methods, it is not enough to build robust deep neural networks with a large number of parameters. Therefore, we propose a three-step procedure for learning an effective semantic vector generator to simulate practical usage of DL-based models. More specifically, we first use some of the oldest bug reports to fine-tune a BERT model that has been pre-trained on a large corpus of bug report in a self-supervised fashion. Then, the trained model can learn semantic vectors of unseen bug reports one by one. Finally, the trained model can be updated as soon as the unseen bug reports have been confirmed to be duplicates or non-duplicates with previous-reported bug reports. The advantage of this procedure is that it prevents the network from cheating by peeking into the future when computing the vector representation of each bug report. After generating the semantic vectors for all bug reports, we follow the experimental setup described above to conduct an empirical study to verify whether the DL-based method helps to improve the effectiveness of duplicate bug report detection.

Results. Figs. 5 and 6 show the distribution of performance metrics of different methods when conducting experiments on 100 randomly selected chunks in the realistic evaluation for each project. Table 7 reports the results of the *Mann–Whitney U* statistical test between different methods. Table 8 presents the kurtosis of the observed performance on different metrics. From the results, we can make the following important conclusions:

(1) As the results show, the performance gap between the DL-based detection method (named DLD in Figs. 5 and 6 for simplicity) and REP or $BM25F_{ext}$ is significant in most metrics, according to the *Mann–Whitney U* test (p-value < 0.05) and the Cliff's delta (*effect size* > 1), for all studied BTSs, which means the DL-based approach (i.e., triplet architecture based on BERT Feature-Net used in this section) performs the worst in the ranking metrics compared to REP (Fig. 5) and $BM25F_{ext}$ (Fig. 6). For example, the DL-based method lowers the performance measured in terms of MAP by a median of 45%–57% percent when compared to the REP method. The low performance of the DL-based approach in the ranking scenario is unexpected. This is because, although we would like semantically duplicate bug reports to be close in the embedding space represented by the DL-based model, this space may not have enough dimensions to generate a proper representation, leading to similar bug reports with the same technical topics to cluster together. That means, when calculating the cosine similarity between semantic vectors to find duplicate candidates for a given bug report, a large number of bug reports with the same topic as the given bug report are likely to be ranked higher. However, most similar bug reports are usually not duplicates of each other, which affects the performance of the DL-based detection method. To support this finding, we also conduct a in-depth case study in Section 5.2.

(2) Among the IR-based methods, REP performs better than $BM25F_{ext}$ on three projects when evaluating in the realistic scenario. This shows that categorical similarities of the REP method are crucial for good performance in detecting duplicate bug reports. Therefore, categorical information should not be disregarded when calculating the similarity between bug reports using IR-based techniques.

(3) As can be seen from Fig. 5 and Table 8, *Recall Rate@k* and MAP display large variation across the evaluated chunks for all the studied BTSs. More specifically, the kurtosis of MAP is 3.243 for REP and 2.462 for the DL-based method on average, which indicates that the same approach (i.e., REP and $BM25F_{ext}$) has large differences for different evaluated chunks with different time periods. This is because the chunks from more recent periods tend to search for massive historical bug reports, making automated approaches more difficult to effectively detect duplicate bug reports. To find evidence to support this analysis, in our experiments, we present the metrics of different approaches across all chunks over the evaluated time period, as shown in Fig. 7. From the results, it can be seen that the chunks with the earliest years achieve the best average performance in terms of various metrics for different methods. This is consistent with previously reported results (Rakha et al., 2017).

As can be seen, the DL-based model itself cannot provide high accuracy for all studied BTSs, and its performance is much lower than that of the IR-based methods REP and $BM25F_{ext}$. This clearly demonstrates the effectiveness of classic IR-based methods for duplicate bug report detection. In addition, this in turn verifies that DL-based methods still have room for further improvement regarding ranking performance.

4.3. Answer question 3: How does combining DL with IR impact the performance of the duplicate bug report detection task?

Motivation. Existing methods rely on textual similarity (i.e., lexical or semantic similarity) to detect duplicate bug reports. However, they lack a comprehensively assessment of textual similarity between bug reports, which is crucial for accurately retrieving duplicate bug reports. Therefore, in this RQ, we investigate whether combining DL with IR techniques can help improve the ranking performance of currently popular detection methods.

Approach. In this section, we also conduct experiments on three large-scale, open-source projects, i.e., OpenOffice, Eclipse and Mozilla. The experimental setup is the same as in Section 4.2. More concretely, we first build a dynamically updated Feature-Net as described in Section 4.2, to generate semantic vectors

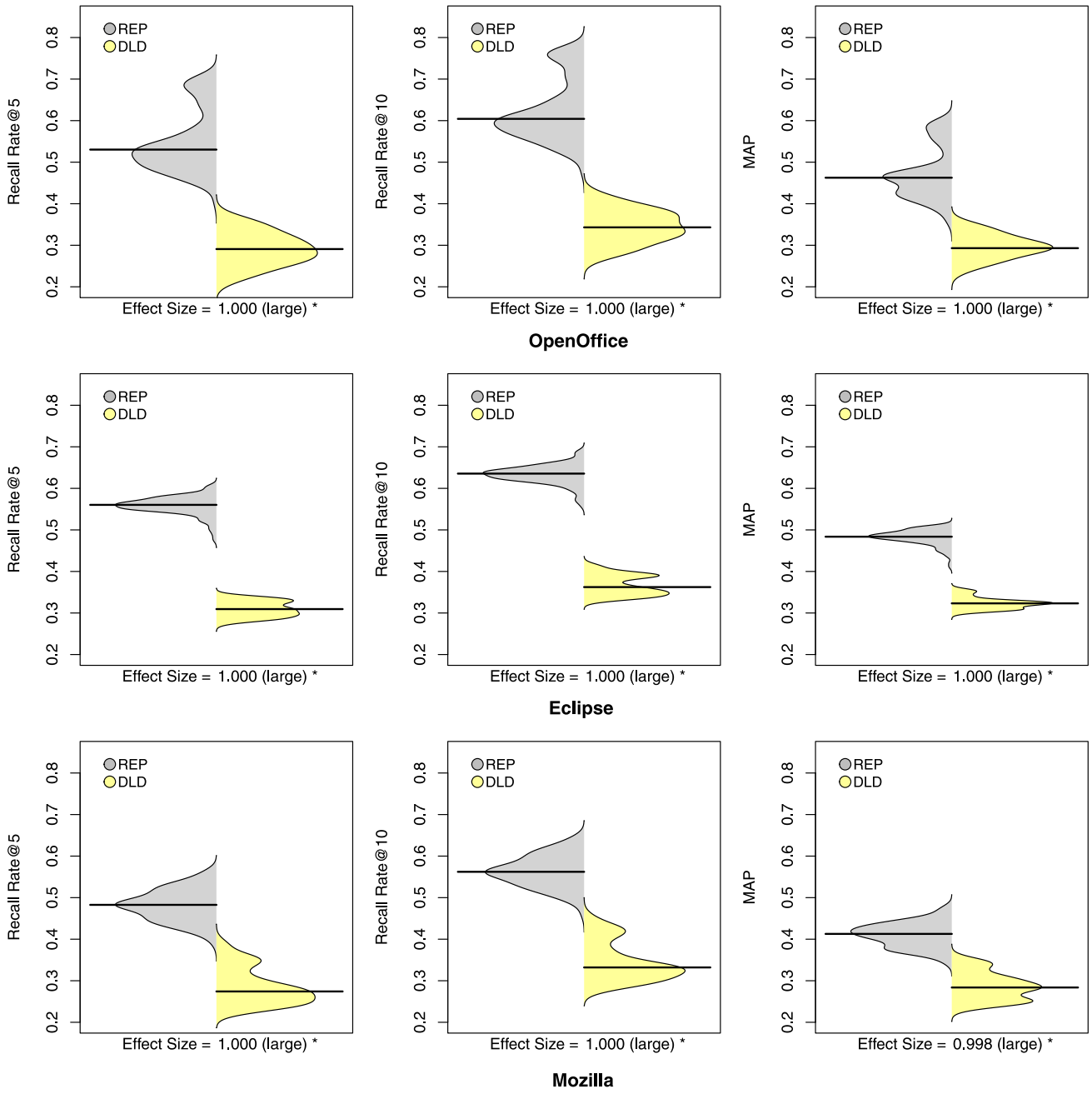

Fig. 5. Performance comparisons of REP and the DL-based method (i.e., DLD) on 100 randomly selected chunks of data using the realistic evaluation.

Table 7
Mann–Whitney U statistical test (P-value) between the DL-based method (i.e., DLD) and IR-based methods (i.e., $BM25F_{ext}$ and REP).

	OpenOffice			Eclipse			Mozilla		
	Recall Rate@5	*Recall Rate@10*	MAP	*Recall Rate@5*	*Recall Rate@10*	MAP	*Recall Rate@5*	*Recall Rate@10*	MAP
DLD vs. $BM25F_{ext}$	3.257e−34	2.562e−34	2.562e−34	2.561e−34	2.562e−34	2.562e−34	1.407e−33	3.899e−34	2.562e−34
DLD vs. REP	2.562e−34	2.562e−34	2.562e−34	2.561e−34	2.562e−34	2.562e−34	2.640e−34	2.562e−34	2.562e−34

Table 8
Kurtosis comparison of the $BM25F_{ext}$, REP and DL-based method in the realistic evaluation.

Kurtosis	OpenOffice			Eclipse			Mozilla		
	DLD	$BM25F_{ext}$	REP	DLD	$BM25F_{ext}$	REP	DLD	$BM25F_{ext}$	REP
Kurtosis (*Recall Rate*@5)	**2.174**	**2.707**	**2.550**	**1.799**	10.004	4.676	**2.393**	**2.303**	**2.425**
Kurtosis (*Recall Rate*@10)	**2.093**	**2.566**	**2.491**	**1.767**	11.651	4.226	**2.318**	**2.112**	**2.587**
Kurtosis (MAP)	**2.409**	**2.552**	**2.511**	**2.848**	10.419	4.802	**2.128**	**2.555**	**2.416**

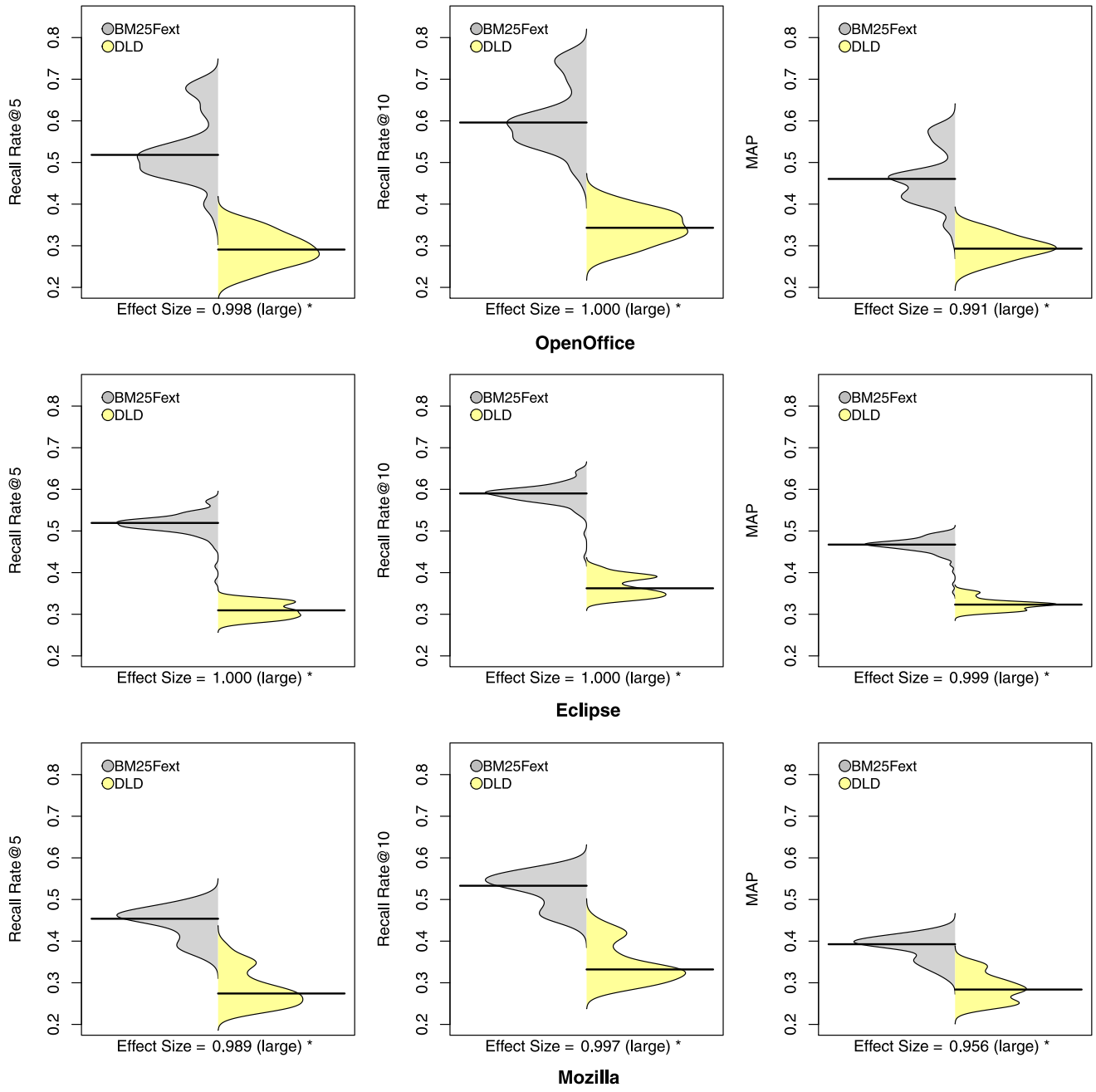

Fig. 6. Performance comparisons of $BM25F_{ext}$ and the DL-based method (i.e., DLD) on 100 randomly selected chunks of data using the realistic evaluation.

for bug reports one by one. Second, we randomly select 100 chunks of data for each BTS to ensure that the studied datasets cover the entire project lifecycle. Thirdly, we use the tuning data consisting of the first 200 duplicate bug reports in each chunk to automatically learn the parameters of the ranking model, and take the remaining bug reports as testing data to compare the performance of different models in the realistic evaluation. Since we are concerned with how much semantic features help to detect duplicate bug reports, we mainly compare the performance between REP and the combined method CombineIRDL, where REP employs lexical and categorical similarities to evaluate the overall similarity between bug reports, while CombineIRDL focuses on improving REP by combining with additional semantic similarity. To determine whether or not the two methods perform significantly differently in terms of various metrics, we perform a statistical test (*Mann–Whitney U* test) on the distribution of experimental results. In addition, we also calculate Cliff's delta to measure the *effect size*, i.e., quantify the amount of difference between the two methods.

Results. Fig. 8 presents the distribution of performance metrics of REP and CombineIRDL when conducting experiments on 100 randomly selected chunks in the realistic evaluation for each studied BTS. Table 9 reports the results of the *Mann–Whitney U* statistical test between REP and CombineIRDL. Table 10 presents the kurtosis on the observed performance in terms of various metrics for REP and CombineIRDL. Fig. 9 presents the cumulative number of master bug reports detected by the REP or CombineIRDL method on 100 chunks of data under the realistic evaluation. From the results, we can draw the following conclusions:

(1) For all projects, CombineIRDL outperforms REP, and the statistical test shows that these improvements are significant

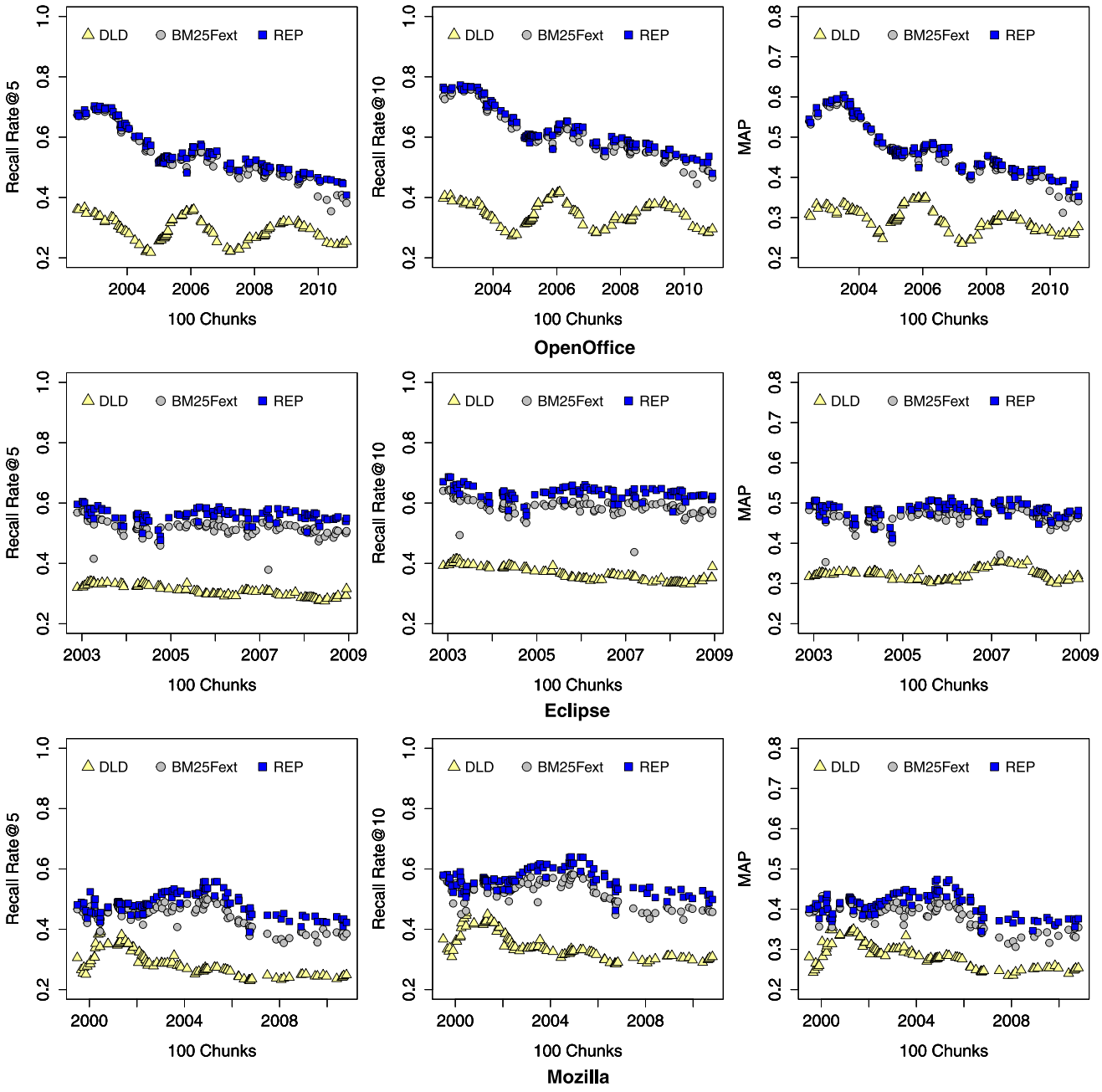

Fig. 7. The *Recall Rate*@5, *Recall Rate*@10, and MAP yielded by the DL-based method (i.e., DLD), $BM25F_{ext}$ and REP using the realistic evaluation over the evaluated time period on 100 chunks of data.

Table 9
Statistical analysis results of the *Mann–Whitney U* test (p-value) between REP and CombineIRDL.

	OpenOffice	Eclipse	Mozilla
Recall Rate@5	5.731e−10	5.803e−31	1.121e−22
Recall Rate@10	2.660e−10	4.490e−31	3.085e−23
MAP	4.180e−15	9.409e−31	8.596e−26

(*effect size* > 0.474) with the exception of OpenOffice in the MAP metric. This demonstrates the usefulness of semantic features for duplicate bug report detection. The experimental results also prove that combining semantic and lexical features can more accurately measure the textual semantic similarity between bug reports.

(2) For different projects, CombineIRDL has varying degrees of performance boost. For example, CombineIRDL achieves an 11.3 performance boost in terms of median MAP over the REP method on the Eclipse project, while it only achieves a 7.1 performance boost on the OpenOffice project. That means the importance of semantic features is tend to be different. This could be due to the fact that the quality of bug reports varies greatly from project to project (Wen et al., 2016). Extracting semantic information from bug reports with low-quality textual contents is difficult. Therefore, more noise could lead to more bias in accurately calculating semantic similarity.

(3) We note from the distribution of *Recall Rate@5*, *Recall Rate@10* and MAP in Fig. 8 that the range of performance values exhibits large variations when evaluating on 100 chunks for each studied BTS. It is consistent with our observation in Section 4.2. As explained earlier, the chunks from the latest years require

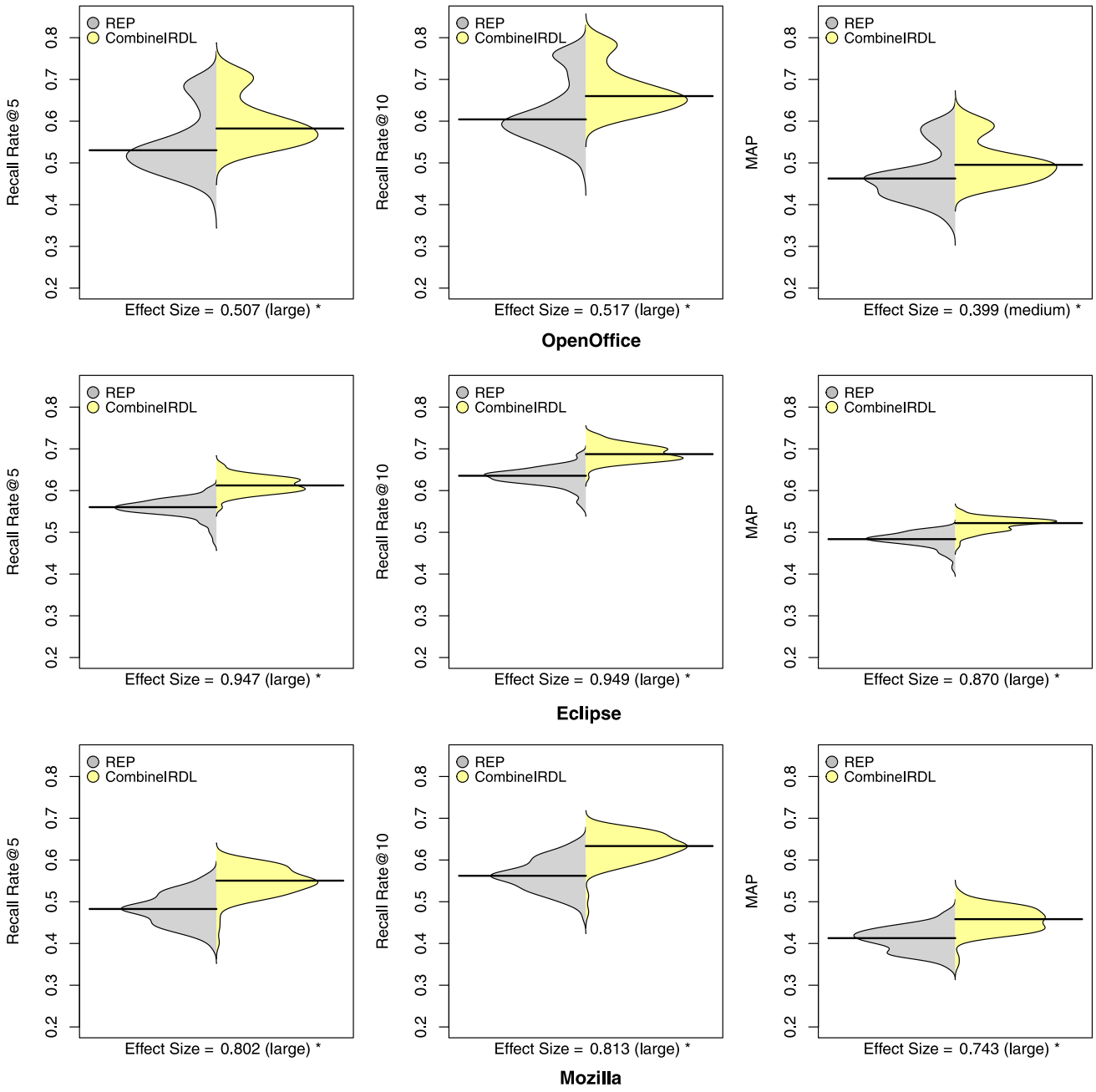

Fig. 8. Performance comparisons of the REP and CombineIRDL approaches on 100 randomly selected chunks of data using the realistic evaluation.

Table 10
Kurtosis comparison of REP and CombineIRDL in the realistic evaluation.

Kurtosis	OpenOffice		Eclipse		Mozilla	
	REP	CombineIRDL	REP	CombineIRDL	REP	CombineIRDL
Kurtosis (*Recall Rate*@5)	**2.550**	**2.474**	4.676	3.075	**2.425**	5.282
Kurtosis (*Recall Rate*@10)	**2.491**	**2.719**	4.226	**2.889**	**2.587**	6.471
Kurtosis (MAP)	**2.511**	**2.523**	4.802	3.648	**2.416**	3.824

searching a large number of historical bug reports to find the possible ones that duplicate with the query bug report. Because of this, the chunks from the earliest years generally lead to better results than those from the latest years.

(4) As can be seen in Fig. 9, the number of duplicate bug reports whose master can be successfully detected by CombineIRDL is greater than that of REP. We also noticed that the number of duplicate bug reports detected by different methods has a very wide range at different periods of time. Specifically, in the early stages of the project lifecycle, the performance between different methods did not differ significantly compared to the later stages. This also emphasizes the importance of performing experiments

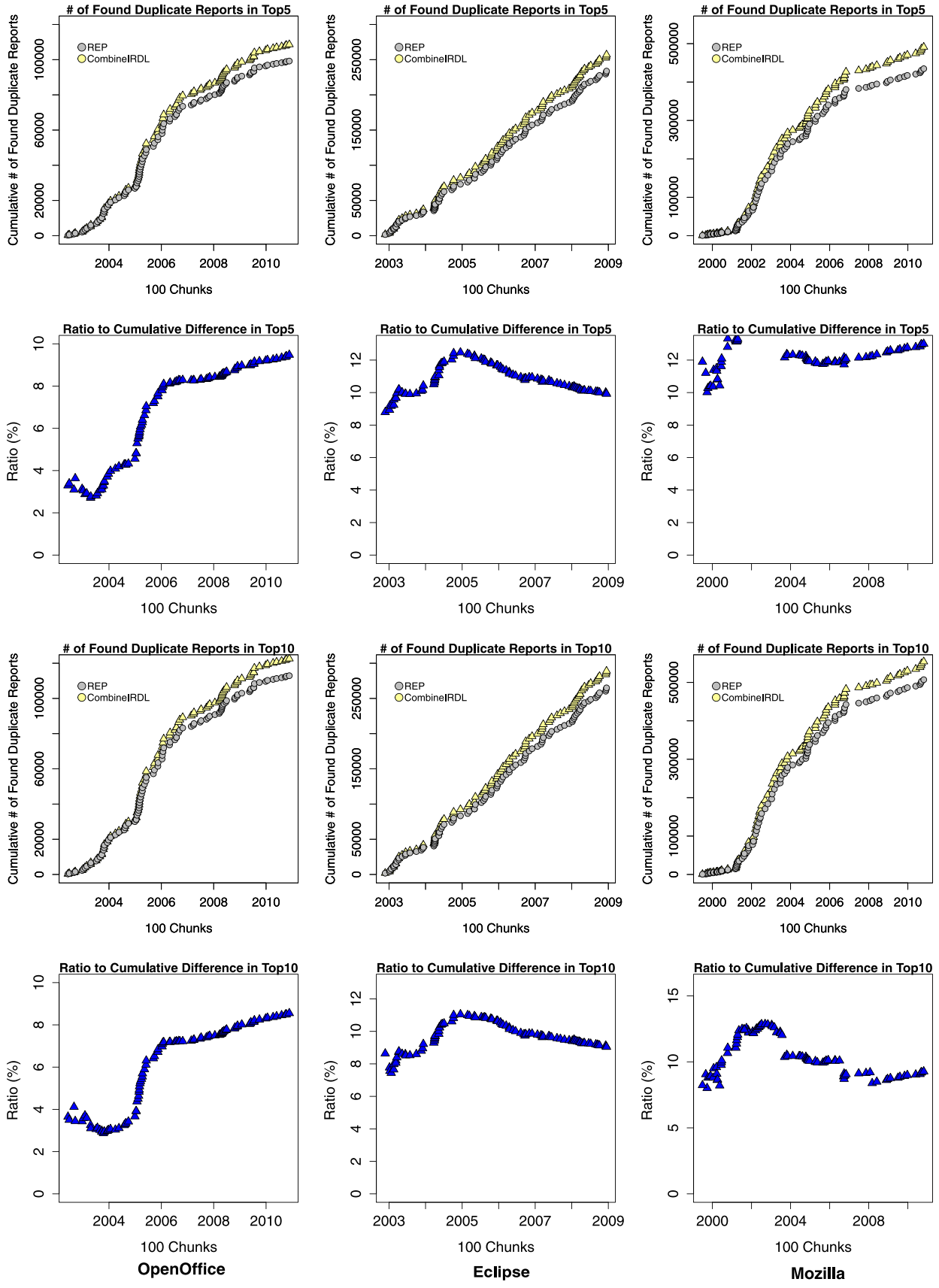

Fig. 9. The cumulative number of found duplicate reports using the realistic evaluation for REP and CombineIRDL approaches over the evaluated time period on 100 chunks of data.

on the entire life span of projects, instead of performing experiments on a randomly selected time period as most previous work did.

Combining the IR-based and DL-based method helps improve the overall detection performance. Specifically, CombineIRDL boosts MAP by on average of 3.03%, with a max average difference of 5.6% on the Mozilla project.

4.4. Answer question 4: How effective is the approach of combining IR and DL when applied to three large unpublished datasets over recent time periods?

Motivation. In RQ3, we have demonstrated the effectiveness of the proposed combined approach CombineIRDL on three publicly available benchmark datasets when compared to classic IR-based methods. However, the bug reports in these three benchmark datasets were submitted before December 31, 2010 (11 years ago before we conducted the study), which may be too old to simulate the present situations as the development organization and process have changed significantly over time. Therefore, we still have no insight into whether the proposed approach CombineIRDL can achieve the desired generalization performance on the bug reports from recent time periods. Therefore, further studies should be performed to answer the above question.

Approach. To answer this RQ, we first conduct a data collection to do data augmentation and then conduct extensive experiments on the newly collected datasets to evaluate the performance of different approaches, including the IR-based method REP and the proposed CombineIRDL in the ranking scenario. For data collection, we first implement a web crawler to collect all bug reports from January 1, 2011 up to December 31, 2020 in OpenOffice, Eclipse and Mozilla BTSs. Each collected bug report is organized as an XML file, which has a set of fields such as *bug_id*, *summary*, *description*, *product*, *component*, *op_sys*, *version*, *priority*, *severity* and others. Next, if the resolution field of the bug report is "*DUPLICATE*" then it will be labeled as duplicate and non-duplicate otherwise. Finally, to ensure the quality of the generated dataset, we filter bug reports that do not have a corresponding master report in the dataset referred by their dup_id field. Table 11 summarizes the statistics of our newly collected datasets. From the table, we notice that the total number of bug reports in the newly collected datasets is still large, especially for Mozilla (953,066 bug reports in total). In addition, we also show statistics on the number of bug reports that must be searched for each bug report in the newly collected datasets, as shown in Fig. 10. As can be seen from the figure, the search space for historical bug reports is huge when experimenting with the newly collected dataset. For example, for bug reports whose bug ids range from 622,321 to 1,684,626 in the collected Mozilla dataset, the number of historical bug reports needed to be searched ranges from 600,000 to 1,500,000.

Targeted at the vast search space of historical bug reports for each testing bug report in the new datasets under realistic evaluation, we choose a year as the unit time of each chunk and divide the new datasets into nonoverlapping chunks. This setting differs from that we used in Sections 4.2 and 4.3, which allows two consecutive chunks to have "overlapping time period". By doing so, we can greatly reduce the number of chunks used for testing to save a lot of computational effort, while ensuring that different methods are evaluated on the entire datasets. It is worth noting that, for Mozilla and Eclipse projects, we follow the data preparation as described above to generate chunks of bug reports that span the entire lifecycle of our collected datasets, while for the OpenOffice project, we only generate chunks covering the time period from January 2011 to June 2014. This is because after 2014 the OpenOffice project reported too few bug reports (less than 200 in the number of duplicate bug reports) to tune and validate the IR-based methods. We will share the databases as benchmarks for future work.

Table 11
The statistics of the newly collected datasets.

Project name	Time Range	# of Reports	# of Duplicates
Eclipse	01/01/11–12/31/20	224,565	15,882
Mozilla	01/01/11–12/31/20	953,066	104,917
OpenOffice	01/01/11–12/31/20	12,062	1413

Results. Fig. 11 shows the distribution of performance metrics of REP and CombineIRDL when conducting experiments on the newly collected datasets in the realistic evaluation. From Fig. 11, we can make the following observations.

(1) CombineIRDL outperforms the classic IR-based method REP in terms of all measures. For example, compared with REP, the proposed CombineIRDL achieves an average performance promotion of 13% regarding MAP. This is mainly due to the fact that IR and DL have distinct advantages in measuring the similarity between bug reports from different aspects. Therefore, by combining both of them, we can achieve the best overall results.

(2) Both methods produce larger variances for different projects. This is due to the fact that detection methods may be sensitive to the text quality of bug reports. Higher-quality bug reports may be easier to be retrieved and then contribute to a better performance than the lower-quality ones. A larger difference in quality among different bug reports indicates a higher variance of performance on the evaluation set. In addition, software development for different projects is done by different teams, each of which has its own writing style and diction for describing bugs (Gegick et al., 2010). Therefore, potential differences in the writing style of bug reports may also lead to a larger variance in performance on the evaluation set.

(3) On the Mozilla project, the superiority of CombineIRDL's performance is more obvious than on other projects. The reason for this may be that semantic similarity plays an extremely important role in duplicate bug report detection for large projects with a significant number of bug reports. Specifically, as stated before, DL-based methods work by automatically discovering the feature space in order to cluster and group bug reports that describe the same bug. This indicates, if two bug reports are quite similar, they should be close to each other in the feature space. Although non-duplicate bug reports from the same technical topic are more likely to be clustered in the same group, it can still greatly reduce the search space for potential duplicate bug reports. This phenomenon seems more pronounced on larger-scale datasets, since the number of bug reports that address the same technical topics does not increase too much with the size of the entire dataset increases.

Extensive experiments on three newly collected datasets validate the effectiveness of the proposed method CombineIRDL. Furthermore, we also demonstrate that semantic similarity has the ability to distinguish bug reports with different semantic meanings, which can greatly reduce the search space of possible duplicate bug reports, especially for large-scale datasets, and thus help to boost the overall performance.

5. Discussion

5.1. What lessons we can learn for future duplicate bug report detection research?

In this study, we perform extensive experiments on three large-scale open-source datasets to demonstrate the practical

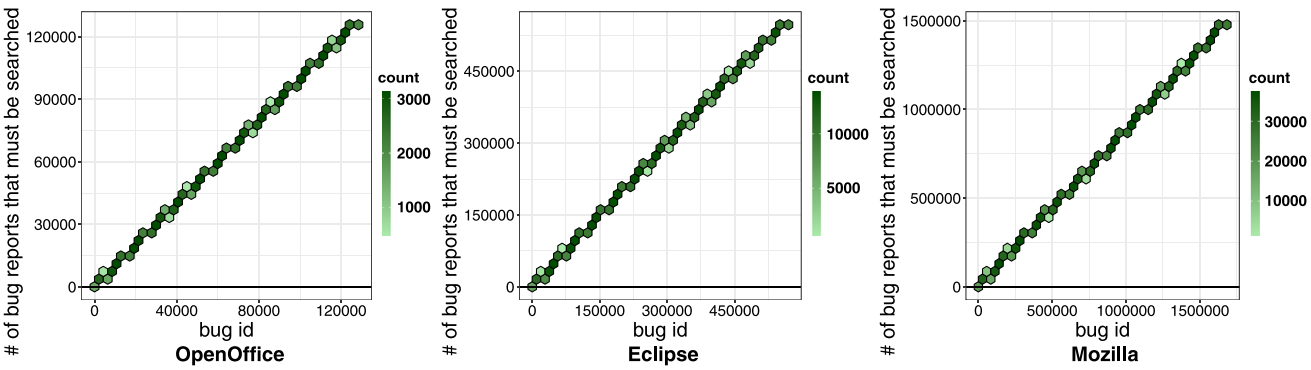

Fig. 10. The number of historical bug reports that must be searched for each bug report with a unique bug id in the realistic evaluation.

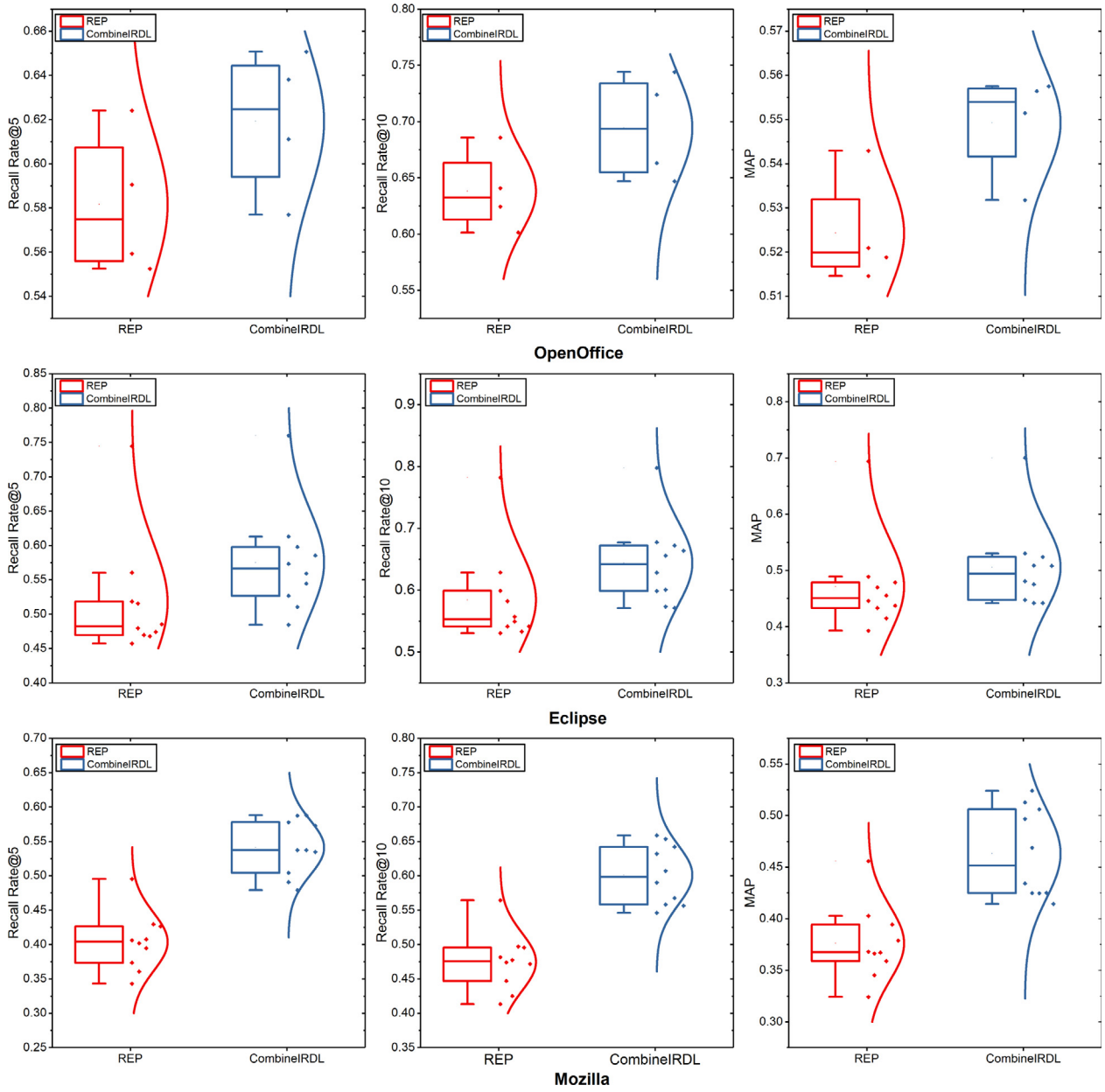

Fig. 11. Boxplots showing the performance of the REP and CombineIRDL methods on newly collected datasets in the realistic evaluation.

effectiveness of DL-based methods compared to other state-of-the-art IR-based methods. From our study, we learned three lessons for future research.

Evaluate the performance of detection models on the entire dataset. Many approaches have been proposed to detect duplicate bug reports. However, most of them are evaluated on the

datasets over a period of time. This setting may introduce randomness or bias such that some results could not be generalized to the lifetime of each studied BTS. For example, as suggested by the experiments in Section 4.3, there are no significant differences in the performance between REP and CombineIRDL in the early stage of studied BTSs, but the differences get increasingly large over time. Therefore, differences in dataset construction may mislead researchers to over- or under-estimate the performance of some techniques. To address this problem, we recommend that researcher's studies should be evaluated on the entire dataset, rather than on a random selected sub-dataset.

Evaluate the performance of DL-based methods in both ranking and classification scenarios. The experimental results are not consistent with previous work (Deshmukh et al., 2017; He et al., 2020), suggesting that DL-based methods can provide better results than the classic IR-based methods. This is because previous work focuses mostly on evaluated their proposed DL-based methods in the classification scenario. However, we observe that higher performance in the classification scenario does not necessarily correlate with better performance in the ranking scenario. More detailed explanations can be found in Section 4.2. Therefore, we suggest that future research should focus more explicitly on improving the performance of DL-based methods on the entire datasets in two evaluation scenarios (i.e., the classification and ranking scenarios).

Deal with duplicate bug report detection by simultaneously considering the lexical and semantic similarity. The basic idea of automatically identifying possible duplicate bug reports is to evaluate their semantic relevance to historical bug reports in the bug repository. Most of the existing approaches are IR-based and DL-based, which calculate the textual similarity between bug reports from different aspects. More concretely, IR-based methods focus on exploiting lexical matching between bug reports to identify duplicate bug reports, while DL-based methods accomplish this detection by measuring the semantic similarity between the feature vectors of bug reports. Although both types of methods have achieved promising performance, each of them being applied alone fails to comprehensively evaluate textual similarity between bug reports, and thus there is still much room for improvement. To alleviate this problem, we propose CombineIRDL, which combines the benefit of IR and DL to capture semantic relatedness between bug reports in a more comprehensive way. Our extensive experiments also demonstrate that the combined method outperforms using either of them separately. Furthermore, we notice that the combined method can reduce the variance of using a single model to some extent, which may also account for the improved performance. Therefore, we suggest that researchers should take into consideration both the lexical and semantic similarity of textual fields when detecting duplicate bug reports.

5.2. In-depth case study

As we detailed in Section 4.2, similar bug reports with the same topic as the query bug report may be ranked higher than the true master report in the list produced by the DL-based method. To support this finding, we take a pair of OpenOffice bug reports in Table 12 as an example to explore a further in-depth case study. From our results, we can observe that if we take bug report 89219 as a query and retrieve its duplicates (i.e., 83441) from historical bug reports, bug report 83441 ranks 31 in the returned list produced by the DL-based method (i.e., BERT-based model). The ranking performance seems far from satisfactory. To find out the reason behind this, we carried out a complementary analysis, creating a topic model over the first 100 bug reports in the ranked list returned by the DL method. The learned topics from 100 bug reports using the LDA model (Alduailij and Al-Duailej, 2015) are shown in Fig. 12.

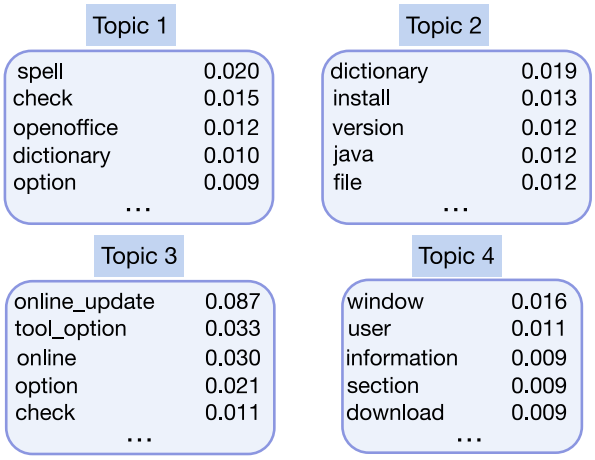

Fig. 12. Top key words for bug report topics learned by an LDA model with 4 topics on the first 100 bug reports retrieved by the DL-based method for query bug report 89219.

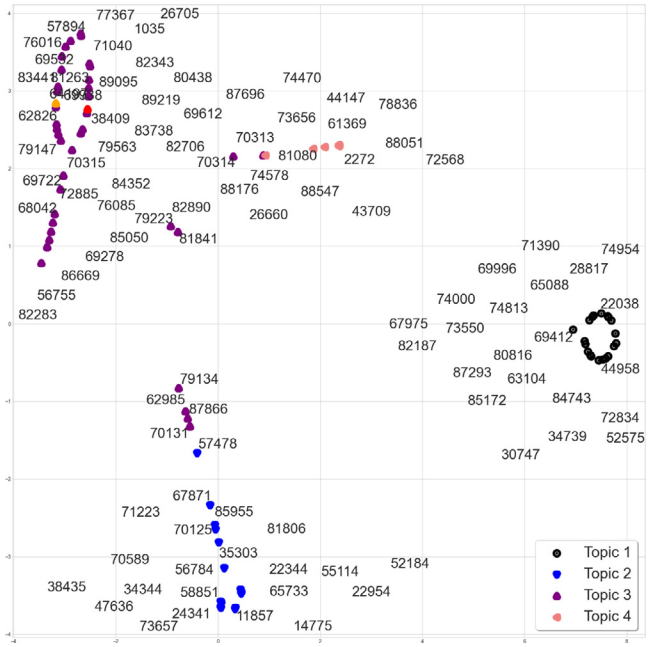

Fig. 13. A 2D visualization of topic features of the top-100 bug reports in the returned list produced by the DL-based method for query bug report 89219 using t-SNE. The legend presents the topics of projected bug reports. For the convenience of distinction, the query bug report is colored in red and the master bug report is colored in orange, although both belong to the 3th topic (colored in purple). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

As can be seen in Fig. 12, the words "online_update", "option" etc. in Topic 3 refer to the software online update operation. Here, we consider each topic recovered by topic modeling as a technical concern (Xia et al., 2016) and use these topics as features to represent the top-100 bug reports in the list returned by the DL-based method. Then, we project the topic features of each bug report with high dimensions to 2-dimensional spaces using t-SNE. Finally, we plot the 2-D projected points in Fig. 13.

As can be seen in Fig. 13, both query bug report 89219 and its duplicates (i.e., 83441) belong to the same topic (i.e., Topic 3). In addition, we find that Topic 3 contains many other bug reports

Table 12
A pair of duplicate bug reports 89219 and 83441 from OpenOffice.

Field	Bug report 1 (query)	Bug report 2 (master)
Bug_id	89219	83441
Summary	missing "online update"	Win2K: Online Update not working
Description	Hi, Install OOo 3.0 beta on pc/windows 2000sp4. "online update" is missing below; "java" : tools − > options − > OOo. Remarks: - OOo 2.4 is run well on this machine; - not possible to run "check for updates" and "add new dictionaries" (proxy setup is ok); - same install file installed on a pc/windows xp sp2 : no problem.	- installed wntmsci11 build on Win2000 and Win98; - installed with selected Online Update Module; - after installation Online Update is not listed in Tools Options; - checking for updates in Help Menu does nothing.
Product	General	Installation
Component	www	code
Op_sys	Windows 2000	Windows, all
Version	OOo 3.0 Beta	OOo 2.3
Priority	P3	P2
Severity	Trivial	Trivial

that are not duplicates with the above two bug reports. This means that bug reports sharing the same topic may not duplicate of each other. In addition, we observed that a large number of bug reports (e.g., 70315, 71040, 72885, 79563, …) with the same topic as the query bug report are likely to be ranked higher than the master report (i.e., 83441). The full ranked list of historical bug reports according to their similarity to the query bug report (i.e., 89219) using the DL-based method can be found in the supplementary material.[4] From this case study, we can conclude that DL methods perform well below our expectations in capturing the semantics of bug reports, especially when distinguishing minor differences between bugs with the same topic.

Even though, DL can help improve the performance of IR-based methods, as shown in the experiments in Section 4.4. Still taking the query bug report 89219 on OpenOffice as an example, its master report (i.e., 83441) can be successfully retrieved at the top of the list returned by the combined method CombineIRDL, outperforming the ranking results produced by both IR-based and DL-based methods.

5.3. Time efficiency

In this section, we present an analysis of execution time for the DL approaches. For the classification scenario, each studied dataset is partitioned into 80% training and 20% testing. When performing experiments on the large-scale datasets shown in the left part of Table 5, the training and testing time of different DL methods are presented in Table 13. As can be seen from the table, BiLSTM+CNN+MLP adopts a combined network to learn bug features automatically from bug reports, which consumes much longer than those use a single network. In addition, we found that BERT-based models take the longest time to train since they use much more complex architectures and typically have more than 100 million parameters (Jiang et al., 2022). In addition, most existing DL-based methods do not show significant differences in prediction time. Furthermore, the prediction time can be amortized across all testing samples. Therefore, a few seconds varying is almost negligible for the entire testing dataset.

In addition to the time analysis of the DL-based methods, we also analyze the tuning and testing time of the IR-based method (i.e., REP) and the combined method (i.e., CombineIRDL). In our experiments, a total of 100 chunks of data are used for empirical study, and the evaluation periods of different chunks vary greatly because they are randomly selected from the entire lifecycle of the studied BTs. This results in a different number of historical bug reports being searched for each bug report in different chunks. Therefore, analyzing the time efficiency of IR-based methods for a single query bug report is not easy. Here, we present the total experimental times of various methods on the benchmark datasets, as shown in Table 14.

As can be seen from the Table 14, our experiments on the benchmark datasets in the ranking scenario take a total of 54,972 min (about 38 days) to compare the performance between REP and CombineIRDL, which is extremely time-consuming due to the vast search space of historical bug reports for each testing bug reports in 100 chunks of data. However, as an empirical study, we consider this expensive work valuable since it provides practitioners with advice on how to better apply DL in real-world applications.

5.4. Threats to validity

5.4.1. Threats to internal validity

In this study, we perform extensive experiments on the effects of different DNNs such as CNN, LSTM and BERT on duplicate bug report detection. Although we have considered some popular neural networks, there may be a few existing techniques that could potentially be used to detect duplicate bug reports but they are ignored in this study. In addition, we compare the performance of several state-of-the-art DL-based methods for duplicate bug report detection. Since no open implementations were available for these methods, we made a fair effort toward the best implementation of DL-based methods based on specifications in their original paper for ease of comparison. Therefore, there could be some minor deviations from the original implementation. To reduce the threats of re-implementation bias, we test our implementation on the datasets from the same projects and compare the results produced by our implementation to the original one. We find that our implementations yield nearly identical results to the original paper. The slight differences in results may be due to the fact that the dataset used in this study is randomly divided into training and testing set, which may not be exactly the same as in the previous studies.

For IR-based methods including REP and $BM25F_{ext}$, we use the original implementation provided by its authors Sun et al. (2011) to avoid re-implementation bias. For the proposed method CombineIRDL, we have double-checked our implementation and all the experimental results to reduce the likelihood of making mistakes.

5.4.2. Threats to external validity

The threat to external validity comes from the generalization of the obtained results. We evaluate the performance of different detection methods on datasets covering the entire lifecycle of three studied BTSs (about 2,190,787 bug reports in total). To the best of our knowledge, the size of dataset used in our experiments is much larger than previous studies. Furthermore, we

[4] https://sites.google.com/view/dbddl.

Table 13
Running time comparisons between different DL models on the large scale datasets in the classification scenario (in Seconds).

Time	OpenOffice				Eclipse				Mozilla			
	Siamese		Triplet		Siamese		Triplet		Siamese		Triplet	
	Training	Testing	Training	Testing	Training	Testing	Training	Testing	Training	Testing	Training	Testing
BiLSTM	743.4	18.39	1195.4	18.6	957.0	20.5	1389.1	20.9	1989.6	28.6	2147.9	26.6
CNN	707.4	19.1	743.4	18.6	934.2	21.8	947.4	20.6	1806.1	27.9	2143.2	30.3
DC-CNN	760.5	19.4	1194.9	18.3	974.4	20.8	1900.8	19.6	1998.9	28.8	4873.5	26.5
BiLSTM+CNN+MLP	1012.5	21.4	1437	19.4	1308.9	24.2	1857.0	21.5	3256.2	39.5	4515.0	28.1
BERT	6504.3	32.5	9178.8	31.1	8325.1	35.0	16 520.4	34.3	18 315.2	43.7	32 368.5	40.1

Table 14
Running time comparisons of REP and CombineIRDL on the benchmark datasets in the ranking scenario (in Minutes).

Project name	REP	CombineIRDL
Eclipse	6768	14,400
Mozilla	8640	20,448
OpenOffice	1428	3288

have demonstrated that the detection methods for recently reported bug reports are still quite effective despite the vast search space of historical bug reports. Nonetheless, we are not sure yet whether the models we considered would be applicable to other projects. To further mitigate the threat, we plan to do a more comprehensive evaluation of various methods by considering more projects in the future.

6. Related work

Many methods have been proposed for detecting duplicate bug reports, which mainly use information retrieval, machine learning and deep learning techniques.

6.1. Information retrieval based methods

Hiew (2006) proposed a practical approach to detect duplicate bug reports, in which the textual fields (i.e., *summary* and *description*) are converted into TF–IDF vectors and then the vectors of similar reports are clustered together as centroids. According to whether the similarity between the newly submitted bug report and each centroid is smaller or greater than a predetermined threshold, it will be labeled as UNIQUE or DUPLICATE. Runeson et al. (2007) used a vector space model to represent the textual information of bug reports, and then calculated the textual similarity between bug reports via three similarity measures including dice, cosine and jaccard. The higher the similarity value, the more likely the two bug reports are duplicates. Wang et al. (2008) proposed an approach by further adding execution information instead of using natural language information alone to detect duplicate bug reports. However, it is often difficult to obtain execution information for most bug reports. Sun et al. (2011) proposed a new duplicate bug report detection model called REP, which linearly combines the lexical similarity of textual fields and the similarity of categorical features to evaluate the global similarity between bug reports. Nguyen et al. (2012) proposed a new model DBTM that combines topic models with information retrieval to improve the accuracy of duplicate bug report detection, and it can address textual dissimilarity between bug reports to some extent. In similar efforts, Alipour et al. (2013), Lazar et al. (2014) and Aggarwal et al. (2017) proposed to combine IR-based techniques with topic models in different ways, which have also been shown to be effective for duplicate bug report detection in their experiments on a particular dataset covering a specific time period of studied BTSs.

6.2. Machine learning based methods

Jalbert and Weimer (2008) developed a classifier to detect duplicate bug reports. The classifier is trained by linear regression using surface features, textual similarity, and graph clustering. Sun et al. (2010) introduced an SVM-based discriminative model to identify duplicate bug reports. Their experiments prove that these methods outperform previous methods based on information retrieval. Gopalan and Krishna (2014) proposed a threshold-based clustering approach to detect duplicate bug reports. Feng et al. (2013) proposed leveraging user profiles and query feedback as features for a classifier to detect whether a pair of bug reports are duplicates. The user profile reflects the knowledge background of the submitter and the quality of submitted reports, while the query feedback can take into consideration the submitter's writing style. In addition, their approach considers three different classifiers, including Naive Bayes, Decision Tree and SVM, as candidate predictors for the detection task. Similar to the work of Feng et al. (2013), Banerjee et al. (2017) first calculated 24 document similarities between two bug reports via a combination of three simple techniques, which is then passed to a random forest classifier provided by the Weka toolkit to predict whether the two bug reports are duplicates of each other.

6.3. Deep learning based methods

Recently, DL has undoubtedly become the most popular technique with great success in many different application areas, such as natural language processing (Otter et al., 2020), computer vision (Voulodimos et al., 2018) and speech recognition (Nassif et al., 2019), etc. Compared with traditional machine learning methods, DL methods have two major advantages: (1) effectively and efficiently capturing highly complicated non-linear features (Li et al., 2017); (2) eliminating the need for hand-crafted feature extractors (LeCun et al., 1998). Such attractive characteristics of DL raise the interest of its applications in the field of software engineering. Using deep learning to automatically capture features from bug reports is an interesting and promising area of research. Performing an empirical study to evaluate whether DL methods offer similar advantages in real situations is an important contribution of this paper to the research field of duplicate bug report detection. Existing DL methods for duplicate bug report detection are as follows.

Deshmukh et al. (2017) introduced deep learning to detect duplicate bug reports for the first time. In their work, various neural networks are used to encode *summary*, *description* and categorical information respectively, and then the learned representations of bug reports are fed into a two-layer network to complete the binary classification task. Budhiraja et al. (2018a,b) proposed a DL-based approach (called DWEN) to detect duplicate bug reports by computing the similarity between two bug reports. Poddar et al. (2019) proposed a multi-task learning (MTL) architecture that works on both detecting duplicate bug reports and aggregating bug report's topics. In addition, considering the different importance of the same words in the two tasks, they designed

a two-step attention mechanism to focus on different text parts of bug reports for different tasks. He et al. (2020) proposed a novel model DC-CNN to represent a pair of bug reports in order to better capture semantic relationships between bug reports, which has shown effectiveness in the duplicate bug report detection task under the classification scenario.

6.4. Empirical study

Besides the above approaches, there are several empirical studies on duplicate bug report detection. For example, Rakha et al. (2017) empirically evaluated the performance of the popular IR-based methods REP and BM25F in different evaluation settings. Their empirical study shows that classical evaluation suffers from problems with over-estimation performance. This happens due to classical evaluation being performed on a subset of dataset spanning across a few years, which significantly limits the search space of historical bug reports, and thus noticeably decreases the difficulty of the task. To address this problem, Rakha et al. (2017) proposed to conduct performance evaluation in a realistic setup without ignoring any bug reports, which allows for a fair comparison of different methods. Inspired by their work, we follow the realistic evaluation setting to compare the performance of different approaches. In addition, to draw stabler conclusions, we adopt the same strategy as in Rakha et al. (2017) to randomly select 100 different chunks of data from each studied BTS to perform experiments, and further conduct some statistical analysis on the experimental results.

Although the evaluating setting is the same as in previous work (Rakha et al., 2017), the present study has several points of novelty. First, to the best of our knowledge, it is the first empirical study work to verify the effectiveness of different DL-based approaches on the entire datasets in both classification and ranking scenarios. Second, instead of using only lexical similarity to measure the relevance between textual fields of bug reports, we propose to combine multiple types of similarities to capture the semantic relatedness between bug reports in a more comprehensive way. Third, we complement existing benchmark datasets by collecting newly submitted bug reports, and then conduct an empirical study to compare the performance of different methods on the new datasets. Note that, the key idea of combining lexical and semantic similarities has been applied to other fields of software engineering, such as bug localization (Lam et al., 2015; Ye et al., 2016). However, it is the first work, to our knowledge, which uses various types of similarities to automatically detect duplicate bug reports and gives an overall assessment of this combined method in realistic evaluations.

7. Conclusions and future work

In this paper, we perform an empirical study to investigate how far the DL-based methods have really progressed in the task of duplicate bug report detection. To this end, we perform a series of experiments on the entire datasets of studied BTSs and make some interesting observations.

In our experiments, we first show that DL-based methods for duplicate bug report detection achieve outstanding performance when evaluated in the classification scenario. This result is consistent with similar findings from previous work. However, through empirical evaluations, we further show that DL-based methods exhibit lower performance compared to IR-based baselines when evaluated in the ranking scenario. Secondly, considering that neither lexical similarity nor semantic similarity can provide a comprehensive measure to get a more accurate textual similarity between bug reports, we propose to tackle this problem by combining the two types of similarities to characterize the relatedness between textual fields of bug reports, which has been shown to be effective for duplicate bug report detection in our study. More concretely, the proposed CombineIRDL achieves a significant performance boost with a range of 7.1% to 11.3% in terms of median MAP over the REP method on three benchmark datasets. Finally, we explore whether these models generalize beyond their benchmark datasets. Through extensive experiments on our newly collected datasets with recently submitted bug reports, we show that CombineIRDL still achieves the state-of-the-art performance on all datasets, outperforming the previous classic IR-based model.

Our future work includes: (1) exploring more advanced models to work with the duplicate bug report detection task, (2) exploring new integrated approaches to combine DL- and IR-based methods, and (3) conducting more experiments on other projects.

CRediT authorship contribution statement

Yuan Jiang: Conceptualization, Data curation, Methodology, Software, Writing – original draft, Writing – review & editing, Formal analysis. **Xiaohong Su:** Supervision, Project administration, Resources, Funding acquisition, Writing – original draft, Writing – review & editing. **Christoph Treude:** Investigation, Software, Writing – original draft, Visualization. **Chao Shang:** Data curation, Writing – original draft. **Tiantian Wang:** Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant Nos.62272132).

References

Aggarwal, K., Timbers, F., Rutgers, T., Hindle, A., Stroulia, E., Greiner, R., 2017. Detecting duplicate bug reports with software engineering domain knowledge. J. Softw.: Evol. Process 29 (3), e1821.

Akbar, S., Kak, A., 2019. SCOR: Source code retrieval with semantics and order. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 1–12.

Alduailij, M., Al-Duailej, M., 2015. Performance evaluation of information retrieval models in bug localization on the method level. In: 2015 International Conference on Collaboration Technologies and Systems. CTS, IEEE, pp. 305–313.

Alipour, A., Hindle, A., Stroulia, E., 2013. A contextual approach towards more accurate duplicate bug report detection. In: 2013 10th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 183–192.

Banerjee, S., Syed, Z., Helmick, J., Culp, M., Ryan, K., Cukic, B., 2017. Automated triaging of very large bug repositories. Inf. Softw. Technol. 89, 1–13.

Bengio, Y., Ducharme, R., Vincent, P., Janvin, C., 2003. A neural probabilistic language model. J. Mach. Learn. Res. 3, 1137–1155.

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., Dutta, K., Shrivastava, M., Reddy, R., 2018b. Towards word embeddings for improved duplicate bug report retrieval in software repositories. In: Proceedings of the 2018 ACM SIGIR International Conference on Theory of Information Retrieval. pp. 167–170.

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G., 2005. Learning to rank using gradient descent. In: Proceedings of the 22nd International Conference on Machine Learning. pp. 89–96.

Cramir, H., 1946. Mathematical Methods of Statistics, Vol. 500. Princeton U. Press, Princeton.

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. ICSME, IEEE, pp. 115–124.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Fan, F., Zhao, W.X., Wen, J.-R., Xu, G., Chang, E.Y., 2017. Mining collective knowledge: inferring functional labels from online review for business. Knowl. Inf. Syst. 53 (3), 723–747.

Feng, L., Song, L., Sha, C., Gong, X., 2013. Practical duplicate bug reports detection in a large web-based development community. In: Asia-Pacific Web Conference. Springer, pp. 709–720.

Gegick, M., Rotella, P., Xie, T., 2010. Identifying security bug reports via text mining: An industrial case study. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp. 11–20.

Gopalan, R.P., Krishna, A., 2014. Duplicate bug report detection using clustering. In: 2014 23rd Australian Software Engineering Conference. IEEE, pp. 104–109.

Guo, J., Cheng, J., Cleland-Huang, J., 2017. Semantically enhanced software traceability using deep learning techniques. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 3–14.

He, J., Xu, L., Yan, M., Xia, X., Lei, Y., 2020. Duplicate bug report detection using dual-channel convolutional neural networks. In: Proceedings of the 28th International Conference on Program Comprehension. pp. 117–127.

Hermawati, A., Mas, N., Hermawati, A., Mas, N., 2017. Overview of the Okapi projects. 59, 602–614. http://dx.doi.org/10.1108/EL-01-2017-0019, arXiv:/dx.doi.org/10.1108/BIJ-10-2012-0068.

Hiew, L., 2006. Assisted Detection of Duplicate Bug Reports (Ph.D. thesis). University of British Columbia.

Hindle, A., Alipour, A., Stroulia, E., 2016. A contextual approach towards more accurate duplicate bug report detection and ranking. Empir. Softw. Eng. 21 (2), 368–410.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Huo, X., Thung, F., Li, M., Lo, D., Shi, S.-T., 2019. Deep transfer bug localization. IEEE Trans. Softw. Eng. 47 (7), 1368–1380.

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. DSN, IEEE, pp. 52–61.

Jiang, Y., Lu, P., Su, X., Wang, T., 2020. LTRWES: A new framework for security bug report detection. Inf. Softw. Technol. 124, 106314.

Jiang, Y., Su, X., Treude, C., Wang, T., 2022. Hierarchical semantic-aware neural code representation. J. Syst. Softw. 191, 111355.

Joanes, D., Gill, C., 1998. Comparing measures of sample skewness and kurtosis. J. R. Stat. Soc. D 47 (1), 183–189.

Jones, K.S., Walker, S., Robertson, S.E., 2000a. A probabilistic model of information retrieval: development and comparative experiments: Part 1. Inf. Process. Manage. 36 (6), 709–808.

Jones, K.S., Walker, S., Robertson, S.E., 2000b. A probabilistic model of information retrieval: development and comparative experiments: Part 2. Inf. Process. Manage. 36 (6), 809–840.

Kaya, M., Bilge, H.Ş., 2019. Deep metric learning: A survey. Symmetry 11 (9), 1066.

Kim, Y., 2014. Convolutional neural networks for sentence classification. Eprint Arxiv.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kochhar, P.S., Xia, X., Lo, D., Li, S., 2016. Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 165–176.

Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N., 2015. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 476–481.

Lazar, A., Ritchey, S., Sharif, B., 2014. Improving the accuracy of duplicate bug report detection using textual similarity measures. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 308–311.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proc. IEEE 86 (11), 2278–2324.

Lee, S.-R., Heo, M.-J., Lee, C.-G., Kim, M., Jeong, G., 2017. Applying deep learning based automatic bug triager to industrial projects. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 926–931.

Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 318–328.

Lin, M.-J., Yang, C.-Z., 2014. An improved discriminative model for duplication detection on bug reports with cluster weighting. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. IEEE, pp. 117–122.

Lin, M.-J., Yang, C.-Z., Lee, C.-Y., Chen, C.-C., 2016. Enhancements for duplication detection in bug reports with manifold correlation features. J. Syst. Softw. 121, 223–233.

Liu, T.-Y., 2011. Learning to Rank for Information Retrieval. Springer Science & Business Media.

Lu, Z., Du, P., Nie, J.-Y., 2020. VGCN-BERT: augmenting BERT with graph embedding for text classification. In: European Conference on Information Retrieval. Springer, pp. 369–382.

Manning, C.D., Raghavan, P., Schütze, H., 2008. Introduction to Information Retrieval. Cambridge University Press, Cambridge, UK, URL: http://nlp.stanford.edu/IR-book/information-retrieval-book.html.

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

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013b. Distributed representations of words and phrases and their compositionality. arXiv preprint arXiv:1310.4546.

Nassif, A.B., Shahin, I., Attili, I., Azzeh, M., Shaalan, K., 2019. Speech recognition using deep neural networks: A systematic review. IEEE Access 7, 19143–19165.

Nguyen, A.T., Nguyen, T.T., Nguyen, T.N., 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.

Nguyen, T.P., Pham, C.C., Ha, S.V.-U., Jeon, J.W., 2018. Change detection by training a triplet network for motion feature extraction. IEEE Trans. Circuits Syst. Video Technol. 29 (2), 433–446.

Otter, D.W., Medina, J.R., Kalita, J.K., 2020. A survey of the usages of deep learning for natural language processing. IEEE Trans. Neural Netw. Learn. Syst. 32 (2), 604–624.

Poddar, L., Neves, L., Brendel, W., Marujo, L., Tulyakov, S., Karuturi, P., 2019. Train one get one free: Partially supervised neural network for bug report duplicate detection and clustering. arXiv preprint arXiv:1903.12431.

Rakha, M.S., Bezemer, C.-P., Hassan, A.E., 2017. Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. IEEE Trans. Softw. Eng. 44 (12), 1245–1268.

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.

Schroff, F., Kalenichenko, D., Philbin, J., 2015. Facenet: A unified embedding for face recognition and clustering. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 815–823.

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

Sun, C., Lo, D., Wang, X., Jiang, J., Khoo, S.-C., 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.

Taylor, M., Zaragoza, H., Craswell, N., Robertson, S., Burges, C., 2006. Optimisation methods for ranking functions with multiple parameters. In: Proceedings of the 15th ACM International Conference on Information and Knowledge Management. pp. 585–593.

Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E., 2018. Deep learning for computer vision: A brief review. Comput. Intell. Neurosci. 2018.

Wang, S., Lo, D., 2016. Amalgam+: Composing rich information sources for accurate bug localization. J. Softw.: Evol. Process 28 (10), 921–942.

Wang, J., Song, Y., Leung, T., Rosenberg, C., Wang, J., Philbin, J., Chen, B., Wu, Y., 2014. Learning fine-grained image similarity with deep ranking. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1386–1393.

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.

Wen, M., Wu, R., Cheung, S.-C., 2016. Locus: Locating bugs from software changes. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 262–273.

Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N., Wang, X., 2016. Improving automated bug triaging with specialized topic model. IEEE Trans. Softw. Eng. 43 (3), 272–297.

Xie, Q., Wen, Z., Zhu, J., Gao, C., Zheng, Z., 2018. Detecting duplicate bug reports with convolutional neural networks. In: 2018 25th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 416–425.

Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C., 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering. pp. 404–415.

Yin, W., Kann, K., Yu, M., Schütze, H., 2017. Comparative study of CNN and RNN for natural language processing. arXiv preprint arXiv:1702.01923.

Youm, K.C., Ahn, J., Lee, E., 2017. Improved bug localization based on code change histories and bug reports. Inf. Softw. Technol. 82, 177–192.

Yuan Jiang, is an assistant professor at the School of Computer Science and Technology, Harbin Institute of Technology. His main research interests are in the areas of mining software repository, software vulnerabilities detection, source code representation.
E-mail: jiangyuan@hit.edu.c.

Xiaohong Su, is a professor at the School of Computer Science and Technology, Harbin Institute of Technology. Her research interests include Intelligent software engineering, software vulnerability identification, code representation learning, bug triaging and localization, clone detection, and code search.
E-mail: sxh@hit.edu.cn.

Christoph Treude, is a Senior Lecturer in the School of Computing and Information Systems at the University of Melbourne, Australia. He received his Ph.D. in computer science from the University of Victoria, Canada in 2012. The goal of his research is to advance collaborative software engineering through empirical studies and the innovation of tools and processes that explicitly take the wide variety of artefacts available in a software repository into account. He currently serves on the editorial board of the Empirical Software Engineering journal and was general co-chair for the IEEE International Conference on Software Maintenance and Evolution 2020.
E-mail: christoph.treude@unimelb.edu.au.

Chao Shang, received his Master's degree in the School of Computer Science and Technology, Harbin Institute of Technology. His main research interests are in the areas of mining software repository, duplicate bug report detection and natural language processing.
E-mail: 19S103131@stu.hit.edu.cn.

Tiantian Wang, born in 1980. Received the Doctor's degree from Harbin Institute of Technology, Harbin, Heilongjiang, China, in 2009. Since 2013, she has been an Associate Professor in computer science department of Harbin Institute of Technology. Her current research interests are software engineering, program analysis and computer aided education.
E-mail: sweetwtt@126.com.