DupHunter: Detecting Duplicate Pull Requests in Fork-Based Development

He Jiang[®], *Member, IEEE*, Yulong Li[®], Shikai Guo[®], Xiaochen Li, Tao Zhang[®], *Senior Member, IEEE*, Hui Li[®], and Rong Chen[®], *Member, IEEE*

Abstract—The emergence of numerous fork-based development platforms facilitates the development of Open-Source Software (OSS) projects. Developers across the world can fork software projects and submit their Pull Requests (PRs) to the projects. However, as the number of forks increases, numerous duplicate PRs might be submitted. These duplicate PRs may cause extra code review workload and frustrate developers working on the projects. To detect duplicate PRs, many approaches have been proposed, which analyze the similarity of different elements in PRs. However, previous approaches still suffer from unsatisfied detection accuracy due to two challenges. That is, they ignore the syntactic structural information of text elements in PRs and lack the joint reasoning between different elements of two PRs. In this study, we propose an automated duplicate PRs detector named DupHunter (Duplicate PRs Hunter), which includes a graph embedding component and a duplicate PRs detection component to address the above challenges. The graph embedding component uses a feature graph to represent a PR. It encodes the syntactic structure and semantics of text elements (e.g., the title and the description), as well as the knowledge of non-text elements (e.g., the submission time), to address the syntactic structural information challenge. The duplicate PRs detection component tackles the joint reasoning challenge using a graph matching network, which enables the information exchange and matching across different elements of two feature graphs with an attention coefficient mechanism. Experiments on 26 open-source projects show that DupHunter achieves an average F1-score@1 value of 0.650, significantly outperforming the state-of-the-art approaches by 3.2% to 48.1%. DupHunter can accurately detect

Manuscript received 14 April 2022; revised 6 September 2022 and 23 December 2022; accepted 28 December 2022. Date of publication 23 March 2023; date of current version 18 April 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 62032004 and 61902050, in part by the Natural Science Foundation of Liaoning Province under Grant 2021-MS-136, in part by the Fundamental Research Funds for the Central Universities under Grants DUT22RC(3)028 and 3132019355, and in part by the Macao Science and Technology Development Fund under Grants 0047/2020/A1 and 0014/2022/A. Recommended for acceptance by Andy Zaidman. (*Corresponding author: He Jiang.*)

He Jiang is with the School of Software, Dalian University of Technology, Dalian 116024, China, and with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116024, China, and also with the DUT Artificial Intelligence Institute, Dalian 116024, China (e-mail: jianghe@dlut.edu.cn).

Yulong Li, Shikai Guo, Hui Li, and Rong Chen are with the School of Information Science and Technology, Dalian Maritime University, Dalian 116026, China (e-mail: liyulong@dlmu.edu.cn; shikai.guo@dlmu.edu.cn; li_hui@dlmu.edu.cn; rchen@dlmu.edu.cn).

Xiaochen Li is with the School of Software, Dalian University of Technology, Dalian 116024, China, and also with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116024, China (e-mail: xiaochen.li@dlut.edu.cn).

Tao Zhang is with the School of Computer Science and Engineering, Macau University of Science and Technology, Taipa, Macao 999078, China (e-mail: tazhang@must.edu.mo).

Digital Object Identifier 10.1109/TSE.2023.3235942

duplicate PRs, with an average *Precision@1* value of 0.922 and an average *Recall@1* value of 0.502.

Index Terms—Duplicate pull requests detection, fork-based development, open source, graph embeddings, text processing.

I. INTRODUCTION

T HE widespread use of fork-based development [1] facilitates the development and evolution of Open-Source Software (OSS) projects, which enables developers around the world to collaborate online [2], [3], [4], [5], [6], [7]. In the collaborative coding process, developers can clone any software repository to make their desired modifications [8], [9], [10], [11]. These modifications can be integrated into the software repository once they are accepted. Since fork-based development helps developers collaborate efficiently, the number of OSS projects using the fork-based mechanism is increasing. Many collaborative coding platforms (such as GitHub [12] and GitLab [13]) also start to support this mechanism, which in turn attracts more developers to make contributions to OSS projects on these platforms productively [14], [15], [16].

The manifestation of a contribution in fork-based development is called a Pull-Request (PR), which contains the *title*, the *description*, and the *patch content* modified by a developer. These PRs mainly aim to fix bugs or implement new features in the project. For example, the OSS project Rails1 has received more than 23,000 PRs during software development [17]. Although fork-based development is widely used [14], it also introduces new problems to the software development process. Since fork-based development is essentially an inconsistent distributed process, duplicate PRs can be submitted as the number of forks increases in projects [18], [19]. Gousios et al. [1] summarize the reasons of rejected PRs in 291 projects on GitHub. They report that 23% of the rejections are due to duplicate PRs. The large number of rejections caused by duplicate PRs can have the following negative effects on the projects.

- Duplicate PRs increase the workload of project maintainers to review code changes [1], [18], [20], [21]. Especially for some popular projects, maintainers have to manually review PRs from thousands of developers [22], [23]. Yu et al. [24] study PRs from 26 popular projects on GitHub. They find that an average of 2.5 reviewers participate in the discussion of a duplicate PR and 5.2 review comments are generated before a duplicate PR is identified.
- Duplicate PRs may lead contributors to be more frustrated, and get doubtful about the project maintainers if duplicate

0098-5589 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information. PRs are improperly managed [24]. Contributors can be frustrated and even conflict with project maintainers if contributors spend a lot of time and effort to discuss a PR with project maintainers (i.e., 5.2 review comments on average), which is later considered to be duplicate. Huang et al. [25] investigate 170 online software projects with open-end survey questions. They find that when conflicts are mismanaged, developers tend to depart from the projects. Also, negative emotions such as frustration and anger may cause contributors to leave.

Several approaches have been proposed to detect duplicate PRs [26], [27], [28]. These approaches either calculate the similarity between different elements of two PRs (e.g., the *title* or the *description* of PRs) or construct features (e.g., the number of overlapping in changed files) to build discriminative prediction models to analyze duplicate PRs. Although these approaches can provide a list of potentially duplicate PR pairs to developers, to accurately detect duplicate PRs, two challenges are still required to address.

Challenge 1. Ignoring Syntactic Structural Information: Previous works [26], [27], [28] consider the title and the description of PRs as text features to detect duplicate PRs. They represent the text features as a series of word tokens to calculate the similarity of two PRs. Such approaches only focus on the frequency of words but overlook the syntactic structural information of text, namely the long-distance and non-consecutive word interactions. Since the title and description written by different developers vary significantly, the frequency of words cannot fully capture the underlying relationships of the syntactic structure of text features.

Challenge 2. Lacking joint reasoning between PRs: Previous works separately measure the similarity of each element (e.g., *title*) between two PRs [26], [27], [28]. They then train a classifier with these separate similarity values to predict duplicate PR pairs. These approaches lack the joint reasoning on two PRs, namely the ability to measure the similarity between different elements of two PRs. Since PRs are written by different developers, it is common that the description or other elements of a PR are incomplete. In this case, the similarity of such elements between two duplicate PRs is low, thus misleading the classifier trained by the existing approaches. Hence, it is important to take different PR elements as a whole and fully consider the information exchange and similarity across PR elements.

To solve these challenges, we propose an automated duplicate PRs detector named *DupHunter* (**Dup**licate PRs *Hunter*). DupHunter includes two main components, namely the graph embedding component and the duplicate PRs detection component. In the graph embedding component, we summarize the text features (e.g., the title and the description) and non-text features (e.g., the time interval of two PR submissions) that characterize the content of a PR. We use a graph structure to capture these features from PRs. For the text features, we use a sliding window co-occurrence mechanism to construct a text feature graph, where graph embedding and word embedding are used to reflect both the structure and the syntax of the text. For the non-text features, the value of each feature is considered as a separate node, which is then aggregated into the text feature graph to form a final feature graph to solve *Challenge 1*. Based on the feature graphs of two PRs, the duplicate PRs detection component applies a graph matching network to detect duplicate PRs. We calculate the cross-graph attention coefficients between two graphs to consider the joint reasoning of two PRs, allowing us to solve *Challenge 2*. This component then maps each feature graph into an embedding vector. We calculate the similarity of two PRs in the vector space and recommend a list of duplicate PRs to project maintainers and developers based on the similarity of PRs.

We conduct comprehensive experiments on a public dataset [24], containing 26 projects on GitHub with 2,323 pairs of duplicate PRs, to evaluate the effectiveness of DupHunter. We compare DupHunter against three state-of-the-art duplicate PRs detection approaches [26], [27], [28]. Experimental results show that DupHunter achieves an average *F1-score@1* value of 0.650, outperforming the comparative approaches by 3.2 pp to 48.1 pp (with pp = percentage points). DupHunter can accurately detect duplicate PRs, with an average *Precision@1* value of 0.922 and an average *Recall@1* value of 0.502. We also analyze the factors affecting the effectiveness of DupHunter. We find *title, description*, and the time interval of two PR submissions are important for duplicate PRs detection.

The main contributions of this work are as follows.

- We propose a novel approach named DupHunter for detecting duplicate PRs in fork-based development. DupHunter utilizes a graph embedding component and a duplicate PRs detection component to address the challenges in duplicate PRs detection.
- Extensive experiments are conducted to evaluate the effectiveness of DupHunter. DupHunter can accurately detect duplicate PRs, significantly outperforming the baselines, in terms of multiple evaluation metrics.
- We release DupHunter as a replication package for improving the efficiency of fork-based development [29]. Developers can take DupHunter as a tool to reduce their time and effort, which are wasted on analyzing duplicate PRs in practice.

The remainder of this paper is organized as follows. Our motivation is discussed in Section II. The main components of DupHunter are introduced in Section III. Experimental setups and experimental results are presented in Sections IV and V, respectively. We discuss practical implications in Section VI. Related works are shown in Section VII. Finally, Section VIII concludes this work and introduces future work.

II BACKGROUND AND MOTIVATION

In this section, we first introduce the fork-based development process used by OSS platforms such as GitHub. Then we use two examples to justify our motivation.

A. Fork-Based Development

As the latest distributed development paradigm [1], forkbased development is a lightweight model that allows developers to easily collaborate with or without explicit coordination. Developers can start development from an existing code repository



Fig. 1. Overview of forked-based development.

by cloning it to their local repository. They can independently make any necessary modification to the local repository and submit the modification to the original repository through the PR mechanism [30]. Fig. 1 shows the main steps of fork-based development [19]:

- a) *Fork:* the developer forks a repository of a project to get a forked repository that contains all files and commit history of the original repository.
- b) *Clone:* the developer clones the forked repository as a local repository on a local computer.
- c) *Edit:* modifications are made in the local repository to fix bugs or add new functionalities in the code.
- d) *Sync:* the developer synchronizes the latest original repository to the local repository to fix any conflict.
- e) *Push:* the local changes are pushed to the forked repository by the developer.
- f) *Submit:* the developer creates PRs in the original repository to submit his/her changes.
- g) *Evaluation:* project maintainers review the PRs and merge the changes to the original repository if the PRs are accepted.

The above process separates the participants of a project into two teams: the external developers and the internal project maintainers [19]. Fork-based development first requires external developers to fork and clone the original repository of a project. External developers can independently modify the cloned local repository in parallel and submit the changes as PRs. The internal project maintainers are responsible for evaluating and integrating the PRs into the original repository.

B. Motivating Examples

Although fork-based development is widely used, it also introduces new problems to the software development process. Since external developers work independently, the number of forked repositories could increase significantly. The large number of forked repositories makes it costly to keep tracking a particular fork [31], thus causing many duplicate PRs. Gousios et al. [1] analyze the reasons of rejected PRs in 290 projects on GitHub and find that 23% of all the rejections are caused by duplicate



Fig. 2. Duplicate PR#35745¹ and PR#32790² in kubernetes/kubernetes.

PRs. However, duplicate PRs are difficult to detect due to two challenges.

Fig. 2 shows a motivating example that demonstrates Chal*lenge 1* for duplicate PRs detection in previous works, i.e., ignoring syntactic structural information. Previous works take the vector representation of features with the one-hot approach [26], [27], [28]. They only focus on the frequency of words in text, but lack the analysis of the long-distance and non-consecutive word interactions. Hence, these approaches cannot fully capture the underlying relationships of the syntactic structure of text [32]. As shown in Fig. 2, the keywords in the title are 'add', 'home', 'environment', and 'windows', but these words are not present sequentially. Therefore, the long-distance relationships of discontinuous words and the context of words can not be effectively learned, thus leading to the inaccurate calculation of the similarity of the two titles. Moreover, the length of the text in the description is usually longer than that in the title, which makes this challenge more serious.

Fig. 3 explains *Challenge 2* for duplicate PRs detection, i.e., lacking joint reasoning between PRs. Although the two PRs in Fig. 3 are duplicate, their titles do not have the same words, thus leading to the poor performance of calculating the similarity between the two titles. However, the title of PR#28978 shown in Fig. 3 has similar semantics with the description of PR#28996, and vice versa. Therefore, compared to calculating the similarity of each element of two PRs separately, it is critical to consider and represent a PR as a whole and conduct joint reasoning on different elements (e.g., the title, the description, the submission time) of two PRs to infer duplicate PRs.

Although traditional similarity calculation approaches such as TF-IDF (term frequency – inverted document frequency) can capture the occurrence of words regardless of their order or distance, it is not enough to detect duplicate PRs by only using TF-IDF. First, in a PR, we can have semantically related words.

2922

¹[Online]. Available: https://github.com/kubernetes/kubernetes/pull/35745

²[Online]. Available: https://github.com/kubernetes/kubernetes/pull/32790

} ⊷ Me	rged k8s-githul	b-robot merged 1	commit into kuberne	tes:master from foxish:	patch-1	
D) C	onversation 4	-O- Commits	1 🗊 Checks	0 🛨 Files changed	1 🖸 on 16 Ju	II 2016
	foxish commer	nted on 15 Jul 2016	6		Mem	hber
	HostPath volun provisioner to questions like t	nes are not autom the controller ma	atically provisioned u mager. This was not n	nless we pass inenable nentioned in the docs prev	-hostpath- riously, leading to	- Description
Inf	o about er	nabling th	e hostpath	provisioner #2	28996	- Title
Info	o about er	habling th	e hostpath	provisioner #2	28996]- Title
Infe ৯ Me দ্ব্য Co	o about er rged k8s-githut onversation 11	nabling th p-robot merged 1 -O- Commits	commit into kuberne 1 E Checks	provisioner #2 tes:master from zefciu: 0	28996 provisioning-docs 1 @ on 20 Jul]- Title
Infe ⊱Me ₽) Ca	o about er rged k8s-githut onversation 11 zefciu commen	nabling th p-robot merged 1 Commits ted on 15 Jul 2016	commit into kuberne 1 E Checks	provisioner #2 tes:master from zefciu: 0 🗈 Files changed	28996 provisioning-docs 1 @ on 20 Jul]- Title

Fig. 3. Duplicate PR#28978³ and PR#28996⁴ in kubernetes/kubernetes.

For example, words 'support' and 'option' have similar semantics in Fig. 2, which cannot be captured by TF-IDF. Second, TF-IDF typically calculates similarity by matching on single words, which overlooks the importance of word sequences or groups. Third, joint reasoning cannot be done by the typical TF-IDF approach.

There, we propose DupHunter for effectively detecting duplicate PRs with the graph embedding component and the duplicate PRs detection component to address these challenges. The graph embedding component uses a sliding window co-occurrence mechanism to construct a feature graph for each PR. This graph reflects both the structure and the syntax of the text to solve Challenge 1. For example, using a sliding window, the longdistance relationship between the words "add" and "windows" in Fig. 2 can be identified through the context word "home". The duplicate PRs detection component applies a graph matching network to connect all the elements in two PRs together. In the network, a cross-graph attention coefficient is applied, which enables the matching of different PR elements to solve Challenge 2. For example, the title of PR#28978 and the description of PR#28996 can be matched through the common words in the text (such as "instruction" and "provision").

Multiple stakeholders can benefit from DupHunter. On the one hand, DupHunter automatically detects duplicate PRs, which can decrease the workload of project maintainers to review redundant code changes. On the other hand, DupHunter can be used to notify developers when potentially duplicate PRs occur in other forks, and encourage developers to collaborate as early as possible. Specifically, DupHunter can send warnings when duplicate PRs are detected, which could assist project maintainers and developers in the fork-based development process.

III. THE DUPHUNTER MODEL

In this section, we first present an overview of the framework of DupHunter in Section III-A. Then, we detail the components included in DupHunter in Sections III-C and III-D.

³[Online]. Available: https://github.com/kubernetes/kubernetes/pull/28978

⁴[Online]. Available: https://github.com/kubernetes/kubernetes/pull/28996

The framework of DupHunter is presented in Fig. 4. DupHunter consists of two components, i.e., the graph embedding component and the duplicate PRs detection component. DupHunter takes two PRs as input. It constructs two categories of features from the PRs, including text features (from title and description) and non-text features (based on the patch content, changed files, etc.). To conduct a comprehensive analysis of the text features, the graph embedding component is used to encode the text features into a text feature graph, which captures the lexical, syntactic, and keywords information of the text (Fig. 4(a)) to address the first challenge, i.e., ignoring syntactic structural information. For non-text features, we compute the similarity of two PRs in terms of each feature. We use the similarity value of each non-text feature as a separate node and aggregate it with the text feature graph in order to make full use of all the features. The output of the graph embedding component is a feature graph consisting of both text and non-text features of a PR. In the duplicate PRs detection component, DupHunter introduces the cross-graph calculation attention coefficient mechanism to enable information exchange and joint reasoning between the feature graphs of two PRs to address the second challenge, i.e., lacking joint reasoning between PRs (Fig. 4(b)). This component encodes a feature graph as a feature vector. We finally detect duplicate PRs by computing the euclidean similarity of the feature vectors of two PRs.

B. Preprocessing

We apply a preprocessing step to preprocess PRs before analyzing duplicate PRs with DupHunter. We use GitHub API⁵ to obtain the information of PRs to be analyzed. For each PR, we use the 'word_tokenize' function in the nltk toolkit⁶ to divide the original sentence in the title and description into words, and separate a word by a list of predefined separators (e.g., 'cluster/centos' becomes 'cluster' and 'centos'). We then use regular expressions to filter out punctuation, and convert all words into lowercase. Next, we remove stop words (e.g., 'to', 'the', 'on') based on an English stop words list.⁷ Finally, Porter stemming is used to convert words into their root forms (e.g., 'files' becomes 'file'). Since a PR is usually used to modify and improve source code, we use the git diff command to obtain the modified code snippets (i.e., the patch content) in a PR. We preprocess the patch content with the preprocessing step used for the title and description. Based on the patch content, we can get the changed lines and files of a PR.

C. Graph Embedding Component

This component characterizes a PR with both text features and non-text features. Specifically, text features of a PR describe the goal and the summary of the code modification by developers, while non-text features show the activity and operations

A. Overview

⁵[Online]. Available: https://docs.github.com/en/rest

⁶[Online]. Available: https://www.nltk.org/

⁷[Online]. Available: https://www.ranks.nl/stopwords



Fig. 4. The framework of Dup-Hunter.

5 6

7

8

9

10

FEATURES AND THEIR PROCESSING METHODS									
Feature	Processing method	Туре							
Title Description	Title Graph Description Graph	Text Featur							
Time interval Reference Ratio of overlapping in changed files Number of overlapping in changed files	Number -1,0,1 Similarity Number	Non toyt							

		TAE	BLE I	
FEATURES	AND 7	Their	PROCESSING	METHODS

conducted by developers when making the code modification. Table I presents the features and their processing methods. We capture these features from each PR. For text features (i.e., *title* and *description*), we process them with graph embedding to identify the syntactic structural information of the text. For non-text features, we mainly calculate their similarity between two PRs.

Patch content similarity on all changed files

Ratio of overlapping in changed lines

Number of overlapping in changed lines

Patch content similarity on overlapping changed files

1) Text-Feature-Based Graph: We encode each text feature into a text feature graph that can represent lexical and syntactic information of the text. The text feature graph is denoted as $G = \{V, E\}$, where V is the words in the text and E represents the edges connecting the words. We use natural language graph embedding to obtain the initial weights of edges E. For each word in V, we use word embedding to represent it as a vector for further analysis.

Natural Language Graph Embedding. Graph embedding encodes texts with a graph structure. The graph structure can preserve the syntactic structural information of sentences by considering the word co-occurrence. To capture the global word co-occurrence, we apply a sliding window [33] on the text of a PR for processing text features. For example, Fig. 5 shows the title of a PR⁸ from the kubernetes/kubernetes repository. For this title "*temporarily point to older bootstrap script*", let's assume that the size of the sliding window is set to three.



res

Features

Fig. 5. An example of sliding window.

Similarity

Similarity

Similarity

Similarity

Therefore each time the sliding window captures three words in this sentence. Starting from the beginning of the sentence, the first three words in the sliding window are "temporarily", "point", and "to". We process the words in each sliding window as follows. We consider each word as a node. We count the number of co-occurrences between node i and node j in the current sliding window W_n . The initial weight x_{ij} of the edge between i and j is the total number of co-occurrences of the two words in all the sliding windows. Here we use co-occurrence

⁸[Online]. Available: https://github.com/kubernetes/kubernetes/pull/661

to analyze syntactic structural information instead of grammars, because the natural language used by developers to write PRs is relatively arbitrary and messy. Many incorrect analyses of the grammatical structure of sentences can be produced by a grammar parser (e.g., Stanford NLP Parser).

We construct a text-feature graph with the above process for two reasons. On the one hand, since the edge weights of the graph are computed based on the sliding windows, they can capture the local information (in a window) of the text features. On the other hand, during the construction of the text-feature graph, the relationship of words can be propagated and aggregated as the window slides. It also helps capture the global syntactic structure of text features.

Word Embedding. We use the word embedding scheme to distribute an initial value for each word (i.e., nodes in the graph). We first obtain the root form of each word with Porter stemming [26] (e.g., "*works*" becomes "*work*"). We then apply a word embedding dictionary trained using Glove⁹ to obtain the embedding vector for each word. Next, We use the default setting of Glove, where the length of the embedding vector is 300. For words that are not in the dictionary, we randomly assign an initialized vector, where the value of each dimension is in the range between -0.01 and 0.01 [34].

2) Non-Text-Feature-Based Graph: We follow previous studies [27], [28] to calculate the values of non-text features. These non-text features are proven to be effective in detecting duplicate PRs. We use the same set of features as previous studies to show the effectiveness of the core components of DupHunter, i.e., the analysis of syntactic structural information and joint reasoning.

Time Interval (TIME): We use the time interval of two PR submissions as a feature. Two PRs are more likely to be duplicate when their created time is close to each other.

Reference (REF): Each PR can link to the issues that it addresses. If two PRs link to the same issue, they are more likely to be duplicate. We define three values for the "Reference" feature [27]: (a) the value of "Reference" is 1 if two PRs link to the same issue; (b) the value is -1 if two PRs link to different issues; (c) otherwise, the value is 0.

Ratio of Overlapping in Changed Files (ROCF): Duplicate PRs are more likely to change the same files. For this feature, we first put the files changed by two PRs into two sets A and B. We use the Jaccard similarity coefficient $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$ to calculate the ratio of overlapping between the changed files by the two PRs [27].

Number of Overlapping in Changed Files (NOCF): We count the number of files changed by both PRs as a feature.

Patch Content Similarity on All Changed Files (PC): We define "patch content" as the code snippets that are different between two source code versions in one PR. We extract the patch content by comparing the original source code files and the modified source code files. According to the study by Wang et al. [28], we use the Term-Frequency Inverse Document Frequency (TF-IDF) to calculate the score of tokens in the patch content and embed all scores into a vector to represent the patch



Fig. 6. Feature graph of a PR.

content of a PR. We consider the cosine similarity between the patch content vectors of two PRs as a feature.

Patch Content Similarity on Overlapping Changed Files (PCOCF): Since two PRs may change the same source code files, we extract the code difference only in the changed files that overlap in the two PRs and calculate the similarity.

Ratio of Overlapping in Changed Lines (ROCL): Duplicate PRs are more likely to change the same lines of code. We calculate the ratio of lines of code that are changed in both PRs with the Jaccard similarity coefficient presented in Feature#5, where A and B in the formulae represent the lines of code changed by the PRs.

Number of Overlapping in Changed Lines (NOCL): We count the number of lines of code changed in both PRs as a feature.

For features ROCF & NOCF, PC & PCOCF, and ROCL & NOCL, they measure the same PR elements in different ways (e.g., ratio versus number, the complete set of files versus overlapping files). These feature pairs are constructed because Ren et al. [27], [27] found that there are 28.5% of PR pairs where one PR is five times larger than another PR in terms of the number of changed files. When this is the case, we could have a small ratio of overlapping in changed files. Hence, the ratio or similarity on the complete set of files could be small (i.e., features ROCF, PC, and ROCL). Features NOCF, PCOCF, and NOCL are designed to solve this problem by analyzing the number or similarity of overlapping files only. A detailed analysis on the importance of these features is presented in Section V-C.

After calculating the non-text features, we expand each feature value to 300 dimensions by adding 299 zeros to make it align with the length of the word embedding vectors. As shown in Fig. 6, we take each non-text feature as a separate node and connect the eight non-text feature nodes in sequence with edges. The weights of these edges are initialized as x_{ij} (e.g., x_{89} is 1 is our work). Finally, we create a feature graph by adding a root node to connect text feature graphs and the non-text feature



Fig. 7. The framework of duplicate PRs detection component.

graph. Since the input of DupHunter is a pair of PRs for analysis, we only append the non-text feature graph to one of the PR pairs. For the other PR, we append it with a "baseline" non-text feature graph, which also has eight nodes connected in sequence. Each "baseline" node vector has 300 dimensions, where the value of the first dimension is 1 and other dimensions are set to zero. In this way, we enforce DupHunter to learn the difference between two PRs regarding non-text features by comparing the non-text feature graph with the "baseline" graph.

3) Duplicate PRs Detection Component

In this stage, our goal is to obtain the similarity value between feature graphs constructed from two PRs. As shown in Fig. 7, we use a graph matching network [34] to connect two feature graphs. This network enables the information exchange between two feature graphs by calculating the attention coefficient of any two nodes in different feature graphs. The graph matching network adjusts the vector representations of two feature graphs to make them more easily distinguished when two PRs are not duplicate.

A graph matching network has three parts, including encoder, propagation layers, and aggregator.

Encoder maps the initial values of nodes and edges in a feature graph to initial vectors using multilayer perceptron (MLP) [34]:

$$h_i^{(0)} = MLP_{node}(x_i), \,\forall i \in V,\tag{1}$$

$$e_{ij} = MLP_{edge}(x_{ij}), \,\forall (i,j) \in E,$$
(2)

where x_i represents the initial value of node *i* in the feature graph, $h_i^{(0)}$ is the node vector after mapping x_i with *MLP*, x_{ij} is the initial value of the edge between two adjacent nodes *i* and *j*, and e_{ij} represents the edge vector after mapping x_{ij} with *MLP*.

Propagation layers map the old node vectors $\{h_i^{(t)}\}_{\forall i \in V}$ (that are composed of both text features and non-text features) into new node vectors $\{h_i^{(t+1)}\}_{\forall i \in V}$ to update nodes state. Thus, the representation of each node vector can accumulate knowledge from its local neighborhood by multiple layers of propagation.

First, the graph matching network calculates the attention coefficient $m_{j\rightarrow i}$ of neighboring nodes in the same feature graph. This coefficient reflects the importance of the relationship between text features and non-text features

$$m_{j \to i} = f_{message}(h_i^{(t)}, h_j^{(t)}, e_{ij}), \, \forall (i, j) \in E,$$
 (3)

where $f_{message}$ is the multilayer perceptron applied to the concatenated input.

Second, the graph matching network calculates the attention coefficients $\mu_{j' \rightarrow i}$ of nodes between two feature graphs. This step considers the joint reasoning between two PRs

$$\mu_{j' \to i} = f_{match}(h_i^{(t)}, h_{j'}^{(t)})(h_i^{(t)} - h_{j'}^{(t)}),$$

$$\forall i \in V_1, j' \in V_2 \text{ or } \forall i \in V_2, j' \in V_1,$$
(4)

where $f_{match}(h_i^{(t)}, h_{j'}^{(t)})$ is a function that conveys the information across feature graphs, which can be calculated by follow:

$$f_{match}(h_i^{(t)}, h_{j'}^{(t)}) = \frac{exp(s_h(h_i^{(t)}, h_{j'}^{(t)}))}{\sum_{j'} exp(s_h(h_i^{(t)}, h_{j'}^{(t)}))},$$
(5)

where s_h is the similarity of two PRs in the vector space.

Then, we calculate the cumulative matching value between the two PRs by substituting formula (5) into formula (4), as follows:

$$\sum_{j'} \mu_{j' \to i} = \sum_{j'} f_{match}(h_i^{(t)}, h_{j'}^{(t)})(h_i^{(t)} - h_{j'}^{(t)})$$
$$= h_i^{(t)} - \sum_{j'} f_{match}(h_i^{(t)}, h_{j'}^{(t)})h_{j'}^{(t)}, \qquad (6)$$

where $\sum_{j'} \mu_{j' \to i}$ intuitively measures the total difference between node *i* and each of its neighbor node *j'* in another feature graph.

In this manner, the vector of a node, its attention coefficients with neighboring nodes in the same feature graph, and its attention coefficients with nodes in another feature graph can be used

#	Repository	# of Dup	PR pairs non-Dup	Ratio of multi-Dup	Language	Domain
1	symfony/symfony	216	4188	6.4%	PHP	Web
2	kubernetes/kubernetes	213	4109	2.4%	Go	Network
3	twbs/bootstrap	127	4240	13.7%	CSS	Web
4	rust-lang/rust	107	4234	3.8%	Rust	Language
5	nodejs/node	104	4178	6.3%	JavaScript	Language
6	symfony/symfony-docs	100	4154	6.4%	PHP	Tool
7	scikit-learn/scikit-learn	68	4141	2.2%	Python	Machine Learning
8	zendframework/zendframework	53	4116	6.5%	PHP	Web
9	servo/servo	52	4106	0.0%	Rust	Web
10	pandas-dev/pandas	49	4232	2.5%	Python	Library
11	saltstack/salt	47	4175	0.0%	Python	Tool
12	mozilla-b2g/gaia	38	4127	0.0%	JavaScript	Operating System
13	rails/rails	199	3702	4.8%	Ruby	Web
14	joomla/joomla-cms	152	3710	6.5%	PHP	Web
15	angular/angular.js	112	985	9.6%	JavaScript	Web
16	ceph/ceph	104	3778	4.8%	C++ ^	Operating System
17	ansible/ansible	103	3780	6.8%	Python	Tool
18	facebook/react	74	3822	7.3%	JavaScript	Tool
19	elastic/elasticsearch	62	3800	4.0%	Java	Network
20	docker/docker	61	3729	3.7%	Go	Network
21	cocos2d/cocos2d-x	57	3760	2.4%	C++	Game
22	django/django	55	3701	4.0%	Python	Web
23	hashicorp/terraform	52	3702	0.0%	Go	Network
24	emberjs/ember.js	46	3783	0.0%	JavaScript	Web
25	JuliaLang/julia	42	3901	3.5%	Julia	Language
26	dotnet/corefx	30	3847	0.0%	C#	Library

TABLE II PROJECT REPOSITORIES AND DUPLICATE PR PAIRS

together for updating the state of the node. We update the node vector with the following formula:

$$h_{i}^{(t+1)} = f_{node} \left(h_{i}^{(t)}, \sum_{j} m_{j \to i}, \sum_{j'} \mu_{j' \to i} \right),$$

$$\forall i, j \in V_{1}, j' \in V_{2} \text{ or } \forall i, j \in V_{2}, j' \in V_{1},$$
(7)

where f_{node} stands for a multilayer perceptron.

Aggregator takes the node vector sets as input to generate a graph-level representation vector h_G . We use the following propagation module [35]:

$$h_G = MLP_G\left(\sum_{i \in V} \sigma(MLP_{gate}(h_i^{(T)})) \odot MLP(h_i^{(T)})\right).$$
(8)

The formula (8) uses a weighted sum with gated vectors MLP_{gate} to perform aggregation across nodes. MLP_G represents the set of all MLP_{gate} . This weighted sum can filter irrelevant information of PRs. If two PRs are similar, their graph-level representation vectors tend to be more similar.

We obtain a graph-level representation vector for each of the feature graphs (i.e., h_{G1} and h_{G2}) and calculate the euclidean similarity between the two vectors to obtain the similarity between two PRs. Finally, DupHunter recommends a list of duplicate PRs to project maintainers and developers based on the similarity values to improve the development efficiency.

IV. EXPERIMENTAL SETTING

A. Dataset

We conduct our experiments on a public dataset from the replication package¹⁰ published by the work of Ren et al. [27].

This dataset contains 2323 duplicate PRs from 26 open-source repositories on GitHub, involving 12 programming languages and various application domains (e.g., web, machine learning, and operating system). In this dataset, all the duplicate PR pairs have been verified in the previous study by manually reading the title, description, and comments of PRs [24]. Following the existing study [24], 1174 pairs of duplicate PRs from repositories #1–#12 in Table II are selected for training and 1149 pairs from repositories #13-#26 are used for testing. In this dataset, there are in total 100000 non-duplicate PR pairs, including 50000 PR pairs for training (in repositories #1-#12) and 50000 PR pairs for testing (in repositories #13-#26)¹¹. These PR pairs are created by randomly paring two merged PRs in a repository, since two merged PRs are usually non-duplicate [27]. Columns 3 and 4 in Table II show the number of PR pairs in each repository in the dataset.

In machine learning, to better use of available data, another way for data partition is to use 80%–90% of data for training, and the remaining data for testing. We do not use this setting to make the setting be consistent with that of existing studies in duplicate PRs detection [27], [28]. Besides, when using 80%–90% of data for training, it assumes that we can get a lot of labeled duplicate PR pairs in different repositories. In the experiments, we do not make this assumption. When training a model with 50% of data (which is a more difficult training scenario than training with 90% of data), it can save some human cost on labeling.

In the column 'ratio of multi-Dup', we counted in each repository the ratio of multiple duplicate PRs, where more than two PRs are duplicate with each other. For the 26 repositories in our dataset, 20 repositories have multiple duplicate PRs. The ratio of multiple duplicate PRs varies from 2.2% (for the

¹¹The number of non-duplicate PR pairs in the repository angular.js is smaller than that of other repositories. We did not modify the dataset by adding additional PR pairs, since the dataset is widely used for duplicate PRs detection.

¹⁰[Online]. Available: https://github.com/luyaor/INTRUDE add

repository scikit-learn/scikit-learn) to 13.7% (for the repository twbs/bootstrap).

B. Experiment Process

We use a similar strategy as the study of Ren et al. [27] to evaluate the effectiveness of our approach in detecting duplicate PRs. Specifically, in our experiment, we analyze the ability of DupHunter in distinguishing duplicate and non-duplicate PR pairs. The basic idea is that the ability of DupHunter to detect duplicate PRs is high if it can rank all duplicate PR pairs ahead of non-duplicate PR pairs. We feed every PR pair in each repository in the testing set to DupHunter. DupHunter sorts all PR pairs based on their similarity values and ranks potentially duplicate PR pairs ahead. We check whether duplicate PR pairs are at the top of the ranking list.

We model the duplicate PRs detection problem in a "crossproject" setting, i.e., training and testing using different repositories, for two reasons. First, there are millions of repositories on open-source software platforms such as GitHub. It is unrealistic and costly to train a model for each repository. It is also costly for developers to manually label a subset of duplicate PR pairs in each project for training a model. The cross-project setting can alleviate these problems. Second, in open-source software platforms, the number of duplicate PRs in each repository can vary significantly. For some new projects, the corresponding repositories may have few duplicate PR pairs initially. In this case, a model trained in a "cross-project" setting can work for these new repositories.

C. Evaluation Metrics

We evaluate DupHunter with three metrics that are used in the related work [28].

Precision@k: Precision is the proportion of duplicate PRs pairs in the ranking list

$$P@k = \frac{N_k}{T_k},\tag{9}$$

where T_k is the number of PR pairs in the top-k% of the ranking list and N_k refers to the number of actual duplicate PR pairs in the top-k% of the ranking list.

Recall@k: Recall is the proportion of duplicate PR pairs that are correctly classified as duplicates among all duplicate PR pairs in a repository

$$R@k = \frac{N_k}{N_r},\tag{10}$$

where N_r refers to the actual number of duplicate PR pairs in each repository.

F1-score@k: The *F1-score* is the harmonic mean of precision and recall. It evaluates whether an increase in *Precision* (or *Recall*) outweighs a reduction in *Recall* (or *Precision*)

$$F@k = \frac{2 \times P@k \times R@k}{P@k + R@k}.$$
(11)

MAP@k: We use MAP@k to measure the position of duplicate PRs in the top-k% of the ranking list. In our context, MAP@k

is defined as follows:

$$MAP@k = \sum_{t=1}^{T_k} \frac{AvgP(t)}{T_k},$$
(12)

where T_k is the number of PR pairs in the top-k% of the ranking list, AvgP(t) is the average precision for the first t PR pairs in the top-k% of list.

In this study, we set k = 1, 2, 3, 4, 5. This setting is consistent with the setting used by the baseline [28]. As a way to measure a recommender system, we can observe the trends and tradeoff of different evaluation metrics by varying k. In addition, as the statistics in Table II, one PR could have multiple duplicates in a repository. By increasing k, it could be beneficial to project maintainers to identify all duplicate PR pairs for better understanding the overall status of PRs in a project.

D. Training Details

The detailed setting for training DupHunter is as follows. We set the size of the sliding window to nine according to the influence analysis of different sliding window sizes in Section V-B. We use a 300-dimensional vector obtained from glove.6B.300 to represent each node in the feature graph as a vector, which is the suggested setting in the previous study [32]. The initial edge weight is obtained by sliding window. Similar to Li et al. [34], the batch size for training DupHunter is set to 32 and the Adam optimizer with an initial learning rate of 0.001 is used.

V. EXPERIMENTAL RESULTS

In this section, four experiments are conducted to evaluate the effectiveness of DupHunter. Our evaluation aims to answer the following Research Questions (RQs).

RQ1: Is DupHunter effective in detecting duplicate PRs compared to the state-of-the-art approaches?

RQ2: What is the influence of the size of the sliding window on DupHunter?

RQ3: What features are important for DupHunter to detect duplicate *PRs*?

RQ4: What is the influence of the size of the training set on DupHunter?

RQ1 is used to evaluate the capability of DupHunter on duplicate PRs detection. RQ2 and RQ3 are employed to evaluate the effect of different parameters on DupHunter. RQ4 is used to evaluate the robustness of DupHunter under different scales of the training data.

A. Answer to RQ1: Comparisons to Baselines

Motivation: DupHunter is the first approach that uses neural networks to detect duplicate PRs. In this RQ, we analyze the accuracy of DupHunter in duplicate PRs detection and compare DupHunter with the state-of-the-art approaches.

Methodology: We compare DupHunter with three categories of approaches. The first category is the algorithms proposed by Li et al. [26], Ren et al. [27] (dubbed as Ren), and Wang et al. [28] (dubbed as Wang). All of them are dedicated duplicate PRs detection approaches. Li et al. [26] proposed three approaches,

TABLE III EFFECTIVENESS OF DUPHUNTER AND BASELINES

	k	Li-T	Li-D	Li-TD	Ren	Wang	ADA	BiMPM	Senten- ceBert	DupH- unter
	1	0.566	0.276	0.493	0.834	0.881	0.791	0.825	0.883	0.922
	2	0.453	0.224	0.395	0.596	0.635	0.649	0.643	0.679	0.723
P@k	3	0.371	0.188	0.327	0.448	0.474	0.540	0.522	0.542	0.572
	4	0.319	0.160	0.280	0.366	0.382	0.452	0.435	0.454	0.471
	5	0.281	0.140	0.244	0.307	0.319	0.394	0.372	0.395	0.407
	1	0.252	0.121	0.211	0.423	0.465	0.419	0.448	0.475	0.502
	2	0.380	0.189	0.316	0.531	0.590	0.641	0.631	0.666	0.716
R@k	3	0.448	0.236	0.384	0.555	0.613	0.750	0.724	0.743	0.787
	4	0.502	0.274	0.439	0.572	0.627	0.801	0.768	0.791	0.824
	5	0.538	0.300	0.479	0.579	0.634	0.838	0.797	0.832	0.856
	1	0.349	0.169	0.295	0.561	0.609	0.548	0.581	0.618	0.650
	2	0.413	0.205	0.351	0.562	0.611	0.645	0.637	0.672	0.719
F@k	3	0.406	0.209	0.354	0.495	0.534	0.628	0.606	0.627	0.662
	4	0.390	0.202	0.342	0.446	0.474	0.578	0.555	0.577	0.600
	5	0.369	0.191	0.323	0.401	0.425	0.536	0.508	0.535	0.552
	1	0.472	0.196	0.407	0.827	0.878	0.730	0.772	0.857	0.905
	2	0.357	0.144	0.306	0.590	0.631	0.573	0.583	0.643	0.701
MAP@k	3	0.284	0.113	0.242	0.444	0.471	0.466	0.462	0.506	0.549
	4	0.238	0.090	0.197	0.363	0.380	0.386	0.380	0.418	0.450
	5	0.206	0.075	0.166	0.304	0.318	0.333	0.323	0.360	0.386

which use TF-IDF to detect duplicate PRs by 1) title similarity, 2) description similarity, and 3) the sum of the two similarities. We name these approaches as Li-T, Li-D, and Li-TD, respectively. We reproduce the approach by Ren et al. [27] with the publicly available source code. For approaches by Li et al. [26] and Wang et al. [28], we re-implement them according to their papers.

The second category of algorithms is ADA [36] and BiMPM [37]. ADA is a cross-domain question-question similarity, which trains neural networks for analyzing duplicate questions in Question & Answer forums such as Stack Exchange. BiMPM is a bilateral multi-perspective matching approach to analyze the similarity of sentences, which achieves better accuracy than many typical deep neural networks. We compare them to investigate whether text similarity analysis approaches for other tasks can be directly used for duplicate PRs detection. The inputs of the two approaches are the title and description of two PRs. The two approaches are reproducible, since their source code are publicly available.

The third category of algorithm is SentenceBert [38]. SentenceBert is aim to reduce the computational overhead with the help of siamese networks. SentenceBert inputs two sentences into two bert models and obtains the sentence representation vector of each sentence. The vectors can be used for semantic similarity calculation. We compare it to investigate whether graph representation performs better than the sequence representation.

For a fair comparison, we use the same training set to train these approaches and test them on the testing set created in Section IV-A. We evaluate the accuracy of each approach on each repository in the testing set with P@k, R@k, F@k, and MAP@k. The overall effectiveness of an approach is computed by averaging an evaluation metric on different repositories.

Results: Experimental results on overall effectiveness are presented in Table III. DupHunter achieves the best results on all

the metrics. P@k is used to evaluate the proportion of detected duplicate PRs in the ranking list. DupHunter achieves a P@lvalue of 0.922. It means DupHunter can accurately identify duplicate PRs. R@k is used to evaluate the proportion of detected duplicate PRs among all duplicate PR pairs. For this metric, DupHunter can identify more than half of duplicate PR pairs in the top-1% of the ranking list. Most of duplicate PR pairs can be found when k = 5 (with a R@5 value of 0.856). As to the trade-off of *precision* and *recall*, F@l of DupHunter is 0.650 at k = 1. Regarding the position of duplicate PR pairs in the ranking list, DupHunter ranks duplicate PR pairs higher than the baselines. DupHunter outperforms the best baselines Wang and SentenceBert by 2.7 pp to 4.8 pp for MAP@l.

For the baselines, Li, Ren, and Wang are state-of-the-art duplicate PRs detection approaches. All the three approaches by Li et al. achieve poor results. It's not surprising because Li et al.'s approaches only consider the similarity of *title* and *description*. To improve Li et al.'s approaches, Ren et al. [27] consider 7 more features (i.e., Feature#4 – Feature#10 in Table I) to train detection models. The accuracy of Ren et al.' approach far exceeds that by Li et al. which means these features are helpful to detect duplicate PRs. For example, Ren outperforms Li-T by 21.3 pp in terms of F@1. After that, Wang et al. [28] further consider the time interval as a new feature (i.e., Feature#3 in Table I), which achieves the best results in terms of the four evaluation metrics. However, Wang et al. still simply calculate the similarity of text features to compare PR pairs.

For the baselines related to text similarity analysis ADA and BiMPM, DupHunter outperforms the two baselines by 9.7 pp to 13.1 pp, 5.4 pp to 7.2 pp, 6.9 pp to 10.2 pp, and 13.3 pp. to 17.5 pp in terms of P@1, R@1, F@1, and MAP@1, respectively. When we extend the ranking list from k=1 to k=5, we can also observe an obvious difference between DupHunter and these two approaches for different sizes of ranking list.

For the baseline SentenceBert, DupHunter outperforms it by 3.9 pp, 2.7 pp, 3.2 pp, 4.8 pp in terms of P@1, R@1, F@1, and MAP@1, respectively. The result could also indicate that the graph structure is better than the text sequence to represent the relations of different features including text and non-text features [32].

Although DupHunter uses the same features as the baselines, we extract the syntactic structural information of text features with feature graphs and complete joint reasoning of all features with a graph matching network. Therefore, DupHunter captures more useful information from these features to improve the accuracy of detecting duplicate PRs.

To present the fine-grained differences between DupHunter and baselines, we show the evaluation results on each repository in Table IV. A value in bold is the best result of an evaluation metric. In general, DupHunter achieves the best results for 11 out of 14 repositories in the testing set in terms of P@1, R@1, and F@1. We note that some baselines can achieve the same best results as DupHunter in a subset of repositories. For example, DupHunter and Wang have achieved the same best results in 6 repositories in terms of P@1, R@1, and F@1. For more than half of repositories, the P@1 value of DupHunter (for

TABLE IV EFFECTIVENESS OF DUPHUNTER AND BASELINES ON EACH REPOSITORY

	P@1									R@1								
	Li-T	Li-D	Li-TD	Ren	Wang	ADA	BiMPM	Senten- ceBert	DupH- unter	Li-T	Li-D	Li-TD	Ren	Wang	ADA	BiMPM	Senten- ceBert	DupH- unter
rails	0.868	0.895	0.974	1.000	1.000	0.947	1.000	0.974	1.000	0.166	0.171	0.186	0.191	0.191	0.181	0.191	0.186	0.191
joomla-cms	0.816	0.711	0.895	0.974	0.947	0.763	0.763	1.000	0.974	0.204	0.178	0.224	0.243	0.237	0.191	0.191	0.250	0.243
angular.js	1.000	0.000	0.900	1.000	1.000	1.000	1.000	1.000	1.000	0.089	0.000	0.080	0.089	0.089	0.089	0.089	0.089	0.089
ceph	0.895	0.605	0.632	1.000	1.000	1.000	0.974	1.000	1.000	0.327	0.221	0.231	0.365	0.365	0.365	0.356	0.365	0.365
ansible	0.763	0.026	0.632	1.000	1.000	1.000	0.763	1.000	1.000	0.282	0.010	0.233	0.369	0.369	0.369	0.282	0.369	0.369
react	0.342	0.368	0.447	0.974	0.974	0.763	0.789	0.921	0.974	0.176	0.189	0.230	0.500	0.500	0.392	0.405	0.473	0.500
elasticsearch	0.658	0.526	0.658	0.684	0.790	0.711	0.816	0.868	0.842	0.403	0.323	0.403	0.419	0.484	0.435	0.500	0.532	0.516
docker	0.460	0.027	0.297	0.892	1.000	0.811	0.892	0.838	1.000	0.279	0.016	0.180	0.541	0.607	0.492	0.541	0.508	0.607
cocos2d-x	0.474	0.026	0.316	0.842	0.947	0.763	0.895	0.974	0.974	0.316	0.018	0.211	0.561	0.632	0.509	0.596	0.649	0.649
django	0.622	0.027	0.108	0.973	1.000	0.730	0.946	0.919	0.973	0.418	0.018	0.073	0.655	0.673	0.491	0.636	0.618	0.655
terraform	0.297	0.162	0.189	0.919	0.973	0.811	0.757	0.865	1.000	0.212	0.115	0.135	0.654	0.692	0.577	0.538	0.615	0.712
ember.js	0.368	0.263	0.395	0.790	0.842	0.684	0.737	0.737	0.868	0.304	0.217	0.326	0.652	0.696	0.565	0.609	0.609	0.717
julia	0.308	0.205	0.436	0.308	0.333	0.513	0.615	0.667	0.692	0.286	0.191	0.405	0.286	0.310	0.476	0.571	0.619	0.643
corefx	0.053	0.026	0.026	0.316	0.526	0.579	0.605	0.605	0.605	0.067	0.033	0.033	0.400	0.667	0.733	0.767	0.767	0.767
					F@	91								MA.	P@1			
	Li-T	Li-D	Li-TD	Ren	F@ Wang	ADA	BiMPM	Senten- ceBert	DupH- unter	Li-T	Li-D	Li-TD	Ren	MA. Wang	P@1 ADA	BiMPM	Senten- ceBert	DupH- unter
rails	Li-T 0.278	Li-D 0.287	Li-TD 0.312	Ren 0.321	F@ Wang 0.321	01 ADA 0.304	BiMPM 0.321	Senten- ceBert 0.312	DupH- unter 0.321	Li-T	Li-D 0.852	Li-TD 0.961	Ren 1.000	MA. Wang 1.000	P@1 ADA 0.940	BiMPM 1.000	Senten- ceBert 0.972	DupH- unter 1.000
rails joomla-cms	Li-T 0.278 0.326	Li-D 0.287 0.284	Li-TD 0.312 0.358	Ren 0.321 0.389	F@ Wang 0.321 0.379	ADA 0.304 0.305	BiMPM 0.321 0.305	Senten- ceBert 0.312 0.400	DupH- unter 0.321 0.389	Li-T 0.793 0.751	Li-D 0.852 0.610	Li-TD 0.961 0.861	Ren 1.000 0.969	MA. Wang 1.000 0.937	P@1 ADA 0.940 0.682	BiMPM 1.000 0.701	Senten- ceBert 0.972 1.000	DupH- unter 1.000 0.965
rails joomla-cms angular.js	Li-T 0.278 0.326 0.164	Li-D 0.287 0.284 0.000	Li-TD 0.312 0.358 0.148	Ren 0.321 0.389 0.164	F@ Wang 0.321 0.379 0.164	ADA 0.304 0.305 0.000	BiMPM 0.321 0.305 0.000	Senten- ceBert 0.312 0.400 0.164	DupH- unter 0.321 0.389 0.164	Li-T 0.793 0.751 1.000	Li-D 0.852 0.610 0.000	Li-TD 0.961 0.861 0.900	Ren 1.000 0.969 1.000	MA. Wang 1.000 0.937 1.000	P@1 ADA 0.940 0.682 1.000	BiMPM 1.000 0.701 1.000	Senten- ceBert 0.972 1.000 1.000	DupH- unter 1.000 0.965 1.000
rails joomla-cms angular.js ceph	Li-T 0.278 0.326 0.164 0.479	Li-D 0.287 0.284 0.000 0.324	Li-TD 0.312 0.358 0.148 0.338	Ren 0.321 0.389 0.164 0.535	F@ Wang 0.321 0.379 0.164 0.535	ADA 0.304 0.305 0.000 0.535	BiMPM 0.321 0.305 0.000 0.521	Senten- ceBert 0.312 0.400 0.164 0.535	DupH- unter 0.321 0.389 0.164 0.535	Li-T 0.793 0.751 1.000 0.872	Li-D 0.852 0.610 0.000 0.545	Li-TD 0.961 0.861 0.900 0.623	Ren 1.000 0.969 1.000 1.000	MA. Wang 1.000 0.937 1.000 1.000	P@1 ADA 0.940 0.682 1.000 1.000	BiMPM 1.000 0.701 1.000 0.940	Senten- ceBert 0.972 1.000 1.000 1.000	DupH- unter 0.965 1.000 1.000
rails joomla-cms angular.js ceph ansible	Li-T 0.278 0.326 0.164 0.479 0.411	Li-D 0.287 0.284 0.000 0.324 0.014	Li-TD 0.312 0.358 0.148 0.338 0.340	Ren 0.321 0.389 0.164 0.535 0.539	F@ Wang 0.321 0.379 0.164 0.535 0.539	ADA 0.304 0.305 0.000 0.535 0.539	BiMPM 0.321 0.305 0.000 0.521 0.411	Senten- ceBert 0.312 0.400 0.164 0.535 0.539	DupH- unter 0.321 0.389 0.164 0.535 0.539	Li-T 0.793 0.751 1.000 0.872 0.638	Li-D 0.852 0.610 0.000 0.545 0.001	Li-TD 0.961 0.861 0.900 0.623 0.514	Ren 1.000 0.969 1.000 1.000 1.000	MA. Wang 1.000 0.937 1.000 1.000 1.000	P@1 ADA 0.940 0.682 1.000 1.000 1.000	BiMPM 1.000 0.701 1.000 0.940 0.675	Senten- ceBert 0.972 1.000 1.000 1.000 1.000	DupH- unter 0.965 1.000 1.000 1.000
rails joomla-cms angular.js ceph ansible react	Li-T 0.278 0.326 0.164 0.479 0.411 0.232	Li-D 0.287 0.284 0.000 0.324 0.014 0.250	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304	Ren 0.321 0.389 0.164 0.535 0.539 0.661	F@ Wang 0.321 0.379 0.164 0.535 0.539 0.661	ADA 0.304 0.305 0.000 0.535 0.539 0.518	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536	Senten- ceBert 0.312 0.400 0.164 0.535 0.539 0.625	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661	Li-T 0.793 0.751 1.000 0.872 0.638 0.178	Li-D 0.852 0.610 0.000 0.545 0.001 0.203	Li-TD 0.961 0.900 0.623 0.514 0.300	Ren 1.000 0.969 1.000 1.000 0.958	MA. Wang 1.000 0.937 1.000 1.000 1.000 0.964	P@1 ADA 0.940 0.682 1.000 1.000 0.725	BiMPM 0.701 1.000 0.940 0.675 0.717	Senten- ceBert 0.972 1.000 1.000 1.000 0.884	DupH- unter 1.000 0.965 1.000 1.000 0.960
rails joomla-cms angular.js ceph ansible react elasticsearch	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.400	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.304	Ren 0.321 0.389 0.164 0.535 0.539 0.661 0.520	F@ Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600	ADA 0.304 0.305 0.000 0.535 0.539 0.518 0.540	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536 0.620	Senten- ceBert 0.312 0.400 0.164 0.535 0.539 0.625 0.660	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.523	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496	Ren 1.000 0.969 1.000 1.000 0.958 0.684	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790	P@1 ADA 0.940 0.682 1.000 1.000 0.725 0.564	BiMPM 1.000 0.701 1.000 0.940 0.675 0.717 0.772	Senten- ceBert 1.000 1.000 1.000 0.884 0.853	DupH- unter 1.000 0.965 1.000 1.000 0.960 0.827
rails joomla-cms angular.js ceph ansible react elasticsearch docker	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500 0.347	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.400 0.020	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.304 0.500 0.225	Ren 0.321 0.389 0.164 0.535 0.539 0.661 0.520 0.674	F@ Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600 0.755	ADA 0.304 0.305 0.000 0.535 0.539 0.518 0.540 0.612	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536 0.620 0.673	Senten- ceBert 0.312 0.400 0.164 0.535 0.539 0.625 0.660 0.633	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640 0.755	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.523 0.338	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314 0.001	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496 0.186	Ren 1.000 0.969 1.000 1.000 0.958 0.684 0.892	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790 1.000	P@1 ADA 0.940 0.682 1.000 1.000 0.725 0.564 0.710	BiMPM 1.000 0.701 1.000 0.940 0.675 0.717 0.772 0.789	Senten- ceBert 1.000 1.000 1.000 0.884 0.853 0.816	DupH- unter 1.000 0.965 1.000 1.000 0.960 0.827 1.000
rails joomla-cms angular.js ceph ansible react elasticsearch docker cocos2d-x	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500 0.347 0.379	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.400 0.020 0.021	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.500 0.225 0.253	Ren 0.321 0.389 0.164 0.535 0.539 0.661 0.520 0.674 0.674	Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600 0.755 0.758	ADA 0.304 0.305 0.000 0.535 0.539 0.518 0.540 0.612 0.611	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536 0.620 0.673 0.716	Senten- ceBert 0.312 0.400 0.335 0.539 0.625 0.660 0.633 0.779	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640 0.755 0.779	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.523 0.338 0.357	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314 0.001	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496 0.186 0.164	Ren 1.000 0.969 1.000 1.000 0.958 0.684 0.892 0.842	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790 1.000 0.947	P@1 ADA 0.940 0.682 1.000 1.000 0.725 0.564 0.710 0.710	BiMPM 1.000 0.701 1.000 0.940 0.675 0.717 0.772 0.789 0.872	Senten- ceBert 0.972 1.000 1.000 1.000 0.884 0.853 0.816 0.967	DupH- unter 1.000 0.965 1.000 1.000 0.960 0.827 1.000 0.972
rails joomla-cms angular.js ceph ansible react elasticsearch docker cocos2d-x django	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500 0.347 0.379 0.500	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.400 0.020 0.021 0.022	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.500 0.225 0.253 0.087	Ren 0.321 0.389 0.164 0.535 0.539 0.661 0.520 0.674 0.674 0.674	F@ Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600 0.755 0.758 0.804	ADA 0.304 0.305 0.000 0.535 0.539 0.518 0.540 0.612 0.611 0.587	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536 0.620 0.673 0.716 0.761	Senten- ceBert 0.312 0.400 0.164 0.535 0.539 0.625 0.663 0.633 0.779 0.739	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640 0.755 0.779 0.783	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.523 0.338 0.357 0.568	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314 0.001 0.001 0.027	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496 0.186 0.164 0.091	Ren 1.000 0.969 1.000 1.000 0.958 0.684 0.892 0.842 0.842 0.970	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790 1.000 0.947 1.000	P@1 ADA 0.940 0.682 1.000 1.000 0.725 0.564 0.710 0.710 0.631	BiMPM 1.000 0.701 1.000 0.940 0.675 0.717 0.772 0.789 0.872 0.893	Senten- ceBert 0.972 1.000 1.000 1.000 0.884 0.853 0.816 0.967 0.887	DupH- unter 1.000 0.965 1.000 1.000 0.960 0.827 1.000 0.972 0.951
rails joomla-cms angular.js ceph ansible react elasticsearch docker cocos2d-x django terraform	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500 0.347 0.379 0.500 0.247	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.400 0.020 0.021 0.022 0.135	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.500 0.225 0.253 0.253 0.087 0.157	Ren 0.321 0.389 0.164 0.535 0.539 0.661 0.520 0.674 0.674 0.674 0.783 0.764	F@ Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600 0.755 0.758 0.804 0.809	ADA 0.304 0.305 0.000 0.535 0.539 0.518 0.540 0.612 0.611 0.587 0.674	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536 0.620 0.673 0.716 0.761 0.761	Senten- ceBert 0.312 0.400 0.164 0.535 0.539 0.625 0.660 0.633 0.779 0.739 0.739	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640 0.755 0.779 0.783 0.783 0.783	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.523 0.338 0.357 0.568 0.155	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314 0.001 0.001 0.027 0.023	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496 0.186 0.164 0.091 0.073	Ren 1.000 0.969 1.000 1.000 0.958 0.684 0.892 0.842 0.970 0.906	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790 1.000 0.947 1.000 0.973	P@1 ADA 0.940 0.682 1.000 0.725 0.564 0.710 0.710 0.631 0.749	BiMPM 1.000 0.701 1.000 0.940 0.675 0.717 0.772 0.789 0.873 0.893 0.730	Senten- ceBert 0.972 1.000 1.000 0.884 0.853 0.816 0.967 0.887 0.820	DupH- unter 1.000 0.965 1.000 1.000 0.960 0.827 1.000 0.972 0.951 1.000
rails joomla-cms angular.js ceph ansible react elasticsearch docker cocos2d-x django terraform ember.js	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500 0.347 0.379 0.500 0.247 0.333	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.400 0.020 0.021 0.022 0.135 0.238	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.500 0.225 0.253 0.087 0.157 0.357	Ren 0.321 0.389 0.164 0.539 0.661 0.520 0.674 0.674 0.783 0.764 0.714	F€ Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600 0.755 0.758 0.804 0.809 0.762	ADA 0.304 0.305 0.000 0.535 0.518 0.518 0.540 0.612 0.611 0.587 0.674 0.619	BiMPM 0.321 0.305 0.000 0.521 0.411 0.536 0.620 0.673 0.716 0.629 0.667	Senten- ceBert 0.312 0.400 0.535 0.539 0.625 0.660 0.633 0.739 0.779 0.719 0.719	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640 0.755 0.779 0.783 0.831 0.786	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.523 0.338 0.357 0.568 0.155 0.245	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314 0.001 0.001 0.027 0.023 0.117	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496 0.186 0.186 0.091 0.073 0.283	Ren 1.000 0.969 1.000 1.000 0.958 0.684 0.892 0.842 0.970 0.906 0.741	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790 1.000 0.947 1.000 0.973 0.836	P@1 ADA 0.940 0.682 1.000 1.000 0.725 0.564 0.710 0.710 0.631 0.749 0.636	BiMPM 1.000 0.701 1.000 0.940 0.675 0.717 0.772 0.789 0.872 0.893 0.730 0.556	Senten- ceBert 1.000 1.000 1.000 0.884 0.853 0.816 0.967 0.820 0.867	DupH- unter 1.000 0.965 1.000 1.000 0.960 0.827 1.000 0.951 1.000 0.855
rails joomla-cms angular.js ceph ansible react elasticsearch docker cocos2d-x django terraform ember.js julia	Li-T 0.278 0.326 0.164 0.479 0.411 0.232 0.500 0.347 0.379 0.500 0.247 0.333 0.296	Li-D 0.287 0.284 0.000 0.324 0.014 0.250 0.020 0.020 0.021 0.022 0.135 0.238 0.198	Li-TD 0.312 0.358 0.148 0.338 0.340 0.304 0.304 0.225 0.253 0.087 0.157 0.357 0.420	Ren 0.321 0.389 0.164 0.535 0.539 0.661 0.520 0.674 0.764 0.764 0.764 0.714 0.296	F€ Wang 0.321 0.379 0.164 0.535 0.539 0.661 0.600 0.755 0.758 0.804 0.809 0.762 0.321	ADA 0.304 0.305 0.000 0.535 0.538 0.518 0.540 0.612 0.611 0.587 0.674 0.619 0.494	BiMPM 0.321 0.305 0.000 0.521 0.411 0.526 0.620 0.673 0.716 0.620 0.673 0.761 0.629 0.667	Senten- ceBert 0.312 0.400 0.164 0.535 0.639 0.633 0.633 0.779 0.739 0.739 0.739 0.667 0.667	DupH- unter 0.321 0.389 0.164 0.535 0.539 0.661 0.640 0.755 0.779 0.783 0.831 0.786 0.786 0.667	Li-T 0.793 0.751 1.000 0.872 0.638 0.178 0.338 0.357 0.568 0.155 0.245 0.159	Li-D 0.852 0.610 0.000 0.545 0.001 0.203 0.314 0.001 0.027 0.023 0.117 0.049	Li-TD 0.961 0.861 0.900 0.623 0.514 0.300 0.496 0.186 0.186 0.073 0.283 0.240	Ren 1.000 0.969 1.000 1.000 0.958 0.684 0.892 0.842 0.970 0.906 0.9741 0.308	MA. Wang 1.000 0.937 1.000 1.000 0.964 0.790 1.000 0.947 1.000 0.947 1.000 0.973 0.836 0.333	P@1 ADA 0.940 0.682 1.000 1.000 0.725 0.564 0.710 0.710 0.631 0.749 0.636 0.405	BiMPM 0.701 1.000 0.675 0.717 0.772 0.789 0.872 0.893 0.730 0.556	Senten- ceBert 1.000 1.000 1.000 0.884 0.853 0.816 0.967 0.887 0.887 0.887 0.687 0.687	DupH- unter 1.000 0.965 1.000 1.000 0.827 1.000 0.972 0.951 1.000 0.855 0.582

TABLE V STATISTICAL TEST (*p*-VALUES) FOR DUPHUNTER AND BASELINES

	Li-T	Li-D	Li-TD	Ren	Wang	ADA	BiMPM	SentenceBert
P@1	0.0015	0.0001	0.0001	0.0180	0.0398	0.0033	0.0033	0.0498
R@1	0.0015	0.0001	0.0001	0.0180	0.0357	0.0033	0.0033	0.0382
F@1	0.0015	0.0001	0.0001	0.0180	0.0298	0.0022	0.0015	0.0367
MAP@1	0.0015	0.0001	0.0001	0.0218	0.1097	0.0033	0.0037	0.0682

ten repositories) is higher than 0.973 and the R@1 value (for seven repositories) is higher than 0.607. Ren and Wang are better than the other baselines, which achieve the best results for 6 and 7 repositories in terms of F@1, respectively. For the F@1 metric, DupHunter and Wang achieve the same results in 6 repositories. Wang outperforms DupHunter in 1 repository (i.e., django/django). For 4 repositories, DupHunter slightly outperforms Wang by 1%-2.4%. For 3 repositories, DupHunter outperforms Wang by a relatively higher margin from 4% to 34.6%. Regarding MAP@1, DupHunter is higher than the best baseline Wang in 7 repositories, and on two repositories MAP@1 of Wang is higher than DupHunter. Although these repositories have different numbers of duplicate PR pairs, DupHunter generally has better ability to rank duplicate PR pairs higher than non-duplicate PR pairs compared with the baselines. Experimental results for setting k as 2 to 5 are available in our replication package, of which DupHunter also gets to best results for the majority of repositories.

We conduct statistical test between DupHunter and baselines according to their effectiveness on each repository in Table IV. The null hypothesis is that there is no difference between DupHunter and a baseline in detecting duplicate PRs on different repositories in terms of a metric (i.e., P@k, R@k, F@k, and MAP@k). We use the Wilcoxon test, since it is a nonparametric statistical test without the assumption of the distribution of the data. The results of p-values are shown in Table V. A value in bold means p > 0.05 (i.e., null hypothesis cannot be rejected). The *p*-values show that there is a significant difference between DupHunter and baselines in detecting duplicate PRs on different repositories in most cases, which means DupHunter can better find duplicate PRs for software repositories compared to baselines. An exception is that there is no difference in terms of MAP@1 between Wang, SentenceBert, and DupHunter. With other k values, ADA and SentenceBert achieve similar results with DupHunter when k = 4 or 5. However, in other cases, DupHunter is preferred.



Fig. 8. A false negative example of DupHunter in the repository elastic/elasticsearch. (a) PR#6124. (b) PR#2326.

regression 3.6.3: Fix xtd buttons close for tinyMCE #12372	Fix issue with TinyMCE editor #13354 I Merged zero-24 merged 1 commit into jocalisistaging from jocalesestationspatch-10 [] on 6 Jan 2017 Q Conversation (a) <> Commits (1) E) Checks (0) E) Files changed (1)
Conversation (a) Conversation (b) Conversation (conversation) Conv	(joomdonation commented on 23 Dec 2016 • edited by joomla-cms-bot • Committee Pull Request for Issue #. Summary of Changes This tiny PR fix a small issue with ImyMCE editor. The javascript code defined in tinymce.minjs use code defined in media/system ///comparis, so we need to ull Htms_Cbehavior.core1; in TimyMCE editor plugin before tinymce.minjs is loaded Testing Instructions 1. Configure your site to use Timy MCE editor .0. Open the file administrator/components/com_content/content.php. replace it with simple PHP code etco 3/actory:rgstdiator()->stapilay('description', ', '108', '108', '109', '6'); . Access to Content-> Articles. Instead of seeing a HTML editor. you will only see a large textanes. If you look at the console of web browser, you will see this javascript error Uncaught ReferenceTrorr. Joomla is not defined at tinymce-init.min.jst
(a)	(b)

Fig. 9. A false positive example of DupHunter in the repository joomla/joomla-cms. (a) PR#12372. (b) PR#13354.

These results have to be interpreted with the usage scenarios of DupHunter. After submitting a PR, developers need to efficiently identify duplicate PRs. A high precision is more important in this scenario, since the ranking list can always contain duplicate PR pairs. When k is increased, more false negatives are included, leading to lower precision. However, as a tradeoff, more duplicate PR pairs can be identified (i.e., higher recall). We assume that the recall can be interested by project maintainers, who may have the requirement to understand the overall status of a project (e.g., all duplicate PRs). Despite the tradeoff, DupHunter outperforms the baselines with different settings.

Error Analysis: Despite the high accuracy of DupHunter, we in this part analyzed two cases of which DupHunter does not perform well. The first case is a false negative that DupHunter assigns a low similarity for two duplicate PRs. The second case is a false positive that DupHunter predicts a high similarity for two non-duplicate PRs.

For the first case, we use $PR\#6124^{12}$ and $PR\#2326^{13}$ in the repository elastic/elasticsearch as an example (in

Fig. 8). The titles of the two PRs are very short. The majority of information is from the description. However, PR#6124 explains the PR with natural language, while PR@2326 presents an example. In the example, there are many example-specific terms, such as 'name', 'Doc1', 'file', 'A', and 'B'. Since DupHunter conducts prediction based on the semantics and syntactic structure of text and source code, these example-specific terms may increase the difficulty to analyze the semantics and structure of PRs. Hence, DupHunter did not predict the two PRs as duplicate. One possible solution to solve this problem is to add a pre-processing step in DupHunter to remove some sentences related to examples or test cases before prediction.

The second case is PR#12372¹⁴ and PR#13354¹⁵ in the joomla/joomla-cms repository. The two PRs in Fig. 9 solve different bugs in the same component (i.e., TinyMCE). However, the semantics of the titles are similar, which contain words 'fix' and 'tinyMCE'. Words 'button' and 'editor' are also semantically similar. In addition, the two PRs use the same template to write the description, including summary of changes, testing

¹²[Online]. Available: https://github.com/elastic/elasticsearch/pull/6124

¹³[Online]. Available: https://github.com/elastic/elasticsearch/pull/2326

¹⁴[Online]. Available: https://github.com/joomla/joomla-cms/pull/12372

¹⁵[Online]. Available: https://github.com/joomla/joomla-cms/pull/13354

instructions, and documentation changes required. Therefore, after pre-processing, DupHunter is misled into taking them as duplicate. This case shows that DupHunter may wrongly identify some duplicate PRs when the PRs are in the same component or implement the same functionality. In such cases, a high threshold to decide duplicate PRs can be needed.

Conclusion: DupHunter is effective in duplicate PRs detection, which significantly outperforms the baselines.

B. Answer to RQ2: Influence of the Size of Sliding Window

Motivation: As mentioned in Section III-C, DupHunter uses a sliding window to extract the syntactic structural information from the *title* and *description* of PRs. The size of the sliding window affects the edge weights when constructing the text feature graphs, which may further affect the input of the graph matching network. In this RQ we analyze the influence of the size of the sliding window on duplicate PRs detection and try to identify the optimal sliding window size for DupHunter.

Methodology: We evaluate the effectiveness of DupHunter by varying the size of the sliding window to 3, 5, 7, 9, 11. We do not evaluate DupHunter with a larger sliding window size for two reasons. On the one hand, as the size of the sliding window increases, more words in a window are considered as co-occurred. This significantly increases the complexity of feature graphs and the time for constructing the graph matching network. On the other hand, according to our preliminary analysis, 84.17% of sentences in the title of a PR are less than 10 words. A sliding window size of 11 can capture the majority of information in a sentence.

Results: The results of P@k, R@k, F@k, and MAP@k for different sliding window sizes are presented in Fig. 10. Experimental results show that a larger sliding window size slightly improves the effectiveness of DupHunter. For example, the P@1 value is 0.828 when the size of the sliding window is 3. This value increases to 0.916 when the size of the sliding window is 11. The reason is that a larger sliding window may take into account the long-distance dependency of words which helps extract more syntactic structural information in PRs. However, when the size of the sliding window is larger than 9, the results of all evaluation metrics become stable. The differences of DupHunter with sliding window sizes of 9 and 11 are less than 1 pp in terms of P@k, R@k, F@k, and MAP@k. Since Fig. 10 shows that DupHunter with a sliding window size of 9 outperforms the majority of other settings in terms of the four metrics, we set the sliding window size as 9 in our experiments.

Conclusion: Within a reasonable range of sliding window sizes, DupHunter gets the best results when the sliding window size is set to 9.

C. Answer to RQ3: Importance of Features

Motivation: In our experiment, DupHunter constructs feature graphs with the information from text features and non-text features to capture the syntactic structural information of PRs. In this RQ, we discuss the influence of these features on detecting duplicate PRs in DupHunter.



Fig. 10. Results on different sliding window sizes. (a) P@k of DupHunter. (b) R@k of DupHunter. (c) F@k of DupHunter. (d) MAP@k of DupHunter.

Methodology: To eliminate the influence of a feature, we delete each feature separately. For example, when we delete the *title* feature from DupHunter, we only train DupHunter with the *description* feature and all non-text features. This variant model is named as DupHunter $_{rmTitle}$, which can be used to analyze the effect of the text feature on DupHunter. Following this approach, we create ten variants of DupHunter, each of which is trained without using one of the features in Table I. In addition, to analyze the importance of all non-text features, we also create a variant called DupHunter $_{rmNontext}$, which is only trained with the text features. Although we try to remove all text features to create another variant, the trained model cannot converge because the remaining non-text features are meaningless for representing PRs. We use P@k, R@k, F@k, and MAP@k to evaluate the effectiveness of these variants.

Results: The experimental results are presented in Tables VI and VII. A value in bold is the lowest value for the variants. We label the lowest value since it means that the corresponding feature has more influence on the accuracy of DupHunter.

Table VI shows the influence of text features. None of the two variants perform better than the original DupHunter, which approves the importance of capturing the syntactic structural information from text features. Both title and description of PRs are important to detect duplicate PRs, since the evaluation metrics drop a lot after removing any of title and description (especially for precision at k = 1). For example, when k = 1, P@1 and MAP@1 drop 6.5 pp and 11.3 pp respectively, when removing the description of PRs. When we increase the ranking list size k to include more PR pairs, the title becomes more important. P@5, R@5, F@5, and MAP@5 values for

TABLE VI THE INFLUENCE OF TEXT FEATURES ON DUPHUNTER

	P@k					R@k					F@k					MAP@k				
DupHunter	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
rmTitle rmDes	0.869 0.857	0.684 0.697	0.534 0.562	0.439 0.469	0.377 0.407	0.467 0.462	0.668 0.686	0.732 0.768	0.764 0.815	0.793 0.849	0.608 0.601	0.676 0.692	0.617 0.649	0.558 0.595	0.511 0.551	0.839 0.792	0.647 0.635	0.500 0.510	0.409 0.423	0.349 0.367
DupHunter	0.922	0.723	0.572	0.471	0.407	0.502	0.716	0.787	0.824	0.856	0.650	0.719	0.662	0.600	0.552	0.905	0.701	0.549	0.450	0.386

TABLE VII THE INFLUENCE OF NON-TEXT FEATURES ON DUPHUNTER

	P@k						R@k				F@k				MAP@k					
DupHunter	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
rmTime	0.873	0.686	0.540	0.449	0.384	0.460	0.658	0.720	0.762	0.782	0.602	0.672	0.617	0.565	0.515	0.852	0.659	0.514	0.425	0.362
rmREF	0.855	0.695	0.562	0.461	0.396	0.459	0.688	0.771	0.803	0.827	0.598	0.691	0.650	0.586	0.536	0.784	0.629	0.506	0.414	0.356
rmROCF	0.837	0.690	0.557	0.465	0.400	0.455	0.688	0.771	0.810	0.836	0.589	0.689	0.647	0.591	0.541	0.767	0.620	0.497	0.414	0.357
rmNOCF	0.906	0.713	0.564	0.465	0.401	0.485	0.692	0.759	0.800	0.829	0.632	0.702	0.647	0.588	0.541	0.893	0.694	0.545	0.446	0.382
rmPC	0.912	0.707	0.561	0.463	0.395	0.492	0.692	0.763	0.802	0.824	0.639	0.699	0.647	0.587	0.534	0.889	0.681	0.537	0.440	0.374
rmPCOCF	0.893	0.699	0.556	0.464	0.397	0.479	0.691	0.764	0.807	0.830	0.624	0.695	0.644	0.589	0.537	0.865	0.664	0.523	0.433	0.369
rmROCL	0.891	0.701	0.554	0.458	0.392	0.480	0.688	0.753	0.788	0.809	0.624	0.695	0.638	0.579	0.528	0.852	0.662	0.521	0.429	0.368
rmNOCL	0.858	0.694	0.556	0.461	0.398	0.454	0.676	0.760	0.803	0.832	0.593	0.685	0.642	0.586	0.538	0.807	0.640	0.508	0.418	0.359
rmNontext	0.307	0.233	0.196	0.169	0.154	0.127	0.180	0.227	0.253	0.292	0.180	0.203	0.210	0.203	0.202	0.214	0.147	0.113	0.093	0.079
DupHunter	0.922	0.723	0.572	0.471	0.407	0.502	0.716	0.787	0.824	0.856	0.650	0.719	0.662	0.600	0.552	0.905	0.701	0.549	0.450	0.386

DupHunter $_{rm Title}$ are 0.377, 0.793, 0.511, and 0.349, respectively, which drop more than only removing the description. We analyze the above results as follows. Both the title and description are important, since the *title* of a PR is often a highly refined and summarized sentence, while the *description* provides more details to accurately analyze PRs. However, as explained in Fig. 8, the *description* of a PR can also contain messy texts (e.g., running examples, test cases, and code analysis), which can be useless for analyzing the syntactic structural information.

Table VII shows the influence of non-text features. We find that removing any non-text feature can affect the effectiveness of DupHunter. Non-text features TIME, REF, ROCF, and NOCL have more influence on the effectiveness of DupHunter, since the accuracy of DupHunter drops a lot on certain evaluation metrics after removing one of these features. REF and ROCF improve the position (i.e., MAP@k) of duplicate PRs, while TIME and NOCL have more influence on precision and recall. For the eight non-text features, NOCF and PC have less influence on DupHunter compared with other features, since the changes of evaluation metrics are small after removing the two features. Moreover, when we remove all non-text features (i.e., DupHunter _{rmNontext}), DupHunter has the lowest accuracy to detect duplicate PRs, which only obtains 0.307, 0.127, 0.180, and 0.214 in terms of P@1, R@1, F@1, and MAP@1, respectively. The above results show that we cannot rely on a single feature to detect duplicate PRs accurately. It is important to consider a PR as a whole, and conduct joint reasoning on all elements of two PRs for duplicate PRs detection.

Conclusion: The syntactic structural information of text features is helpful to detect duplicate PRs. Features *ti*-*tle*, *description*, *TIME*, *REF*, *ROCF*, and *NOCL* are more important.

D. Answer to RQ4: Influence of the Size of Training Set

Motivation: As mentioned in Section III-D, we use the graph matching network to train DupHunter. However, when training a neural network, the size of the training set usually affects the effectiveness of the model. In this RQ, we analyze the influence of the size of the training set on DupHunter.

Methodology: To build training sets with different sizes, we randomly select a subset of 20% (i.e., 10000) to 100% (i.e., 50000) PR pairs with a step of 20% (i.e., 10000) from the 50000 non-duplicate PR pairs in the original training set. We then add the 1174 duplicate PR pairs to each selected subset of non-duplicate PR pairs to form new training sets. For example, when we select 20% non-duplicate PR pairs, the new training set contains $500000 \times 20\% + 1174 = 11174$ PR pairs. We run DupHunter on the new training sets with different sizes and use P@k, R@k, F@k, and MAP@k to evaluate the effectiveness of the trained models.

In this RQ, we only change the number of non-duplicate PR pairs for the following reason. Since we test DupHunter in a cross-project setting, we expect it is not a challenge to get the number of duplicate PR pairs required for training (e.g., 1174 in this study) from different projects. In this case, the number of non-duplicate PR pairs is more important, since one can easily generate millions of non-duplicate PR pairs by paring two merged PRs.

Results: The experimental results are presented in Fig. 11. Clearly, the effectiveness of DupHunter improves as the size of the training set increases. When the size of the training set is small (e.g., with 11,174 PR pairs), DupHunter performs poorly. The P@1, R@1, F@1, and MAP@1 values of DupHunter are all less than 0.300. The reason is that a small training set can lead to incomplete learning of the neural network used by DupHunter. However, the effectiveness of DupHunter



Fig. 11. Results on different training set sizes. (a) P@k of DupHunter. (b) R@k of DupHunter. (c) F@k of DupHunter. (d) MAP@k of DupHunter.

significantly improves, when the size of the training set increases from 11174 to 21174. With 21174 PR pairs in the training set, the effectiveness of DupHunter increases to 0.867, 0.459, 0.600, and 0.838 in terms of the four metrics. This improvement is mainly caused by the increase of precision of DupHunter. As more PR pairs are added to the training set, DupHunter can better learn to distinguish duplicate PR pairs from non-duplicate ones. Meanwhile, the differences of the distribution between duplicate and non-duplicate PR pairs also become smaller for the training and testing sets. The evaluation metrics become stable as we continue to increase the size of the training set. For example, the difference of F@1 is less than 1% when training DupHunter with 31174 and 51174 PR pairs. It means when the size of the training set reaches a certain scale, the accuracy of DupHunter will keep relatively stable. According to the trends in Fig. 11, DupHunter can obtain high accuracy with a training set containing 21174 PR pairs. Since DupHunter can be used in the cross-project setting, and non-duplicate PR pairs are relatively easy to generate, the size of the training set is not a burden for training DupHunter.

Conclusion: The effectiveness of DupHunter improves when the size of the training set increases. When the size of the training

TABLE VIII TIME FOR DUPLICATE PR DETECTION OF EACH APPROACH

#	Pre-processing	Train	Test	Inference
Li-T	$21\mathrm{m}$	N/A	$0.2\mathrm{s}$	$< 0.1 \mathrm{s}$
Li-D	$21\mathrm{m}$	N/A	$0.2\mathrm{s}$	$< 0.1 \mathrm{s}$
Li-TD	$21\mathrm{m}$	N/A	$0.2\mathrm{s}$	$< 0.1 \mathrm{s}$
Ren	$25\mathrm{m}$	$5.5\mathrm{s}$	$1.4\mathrm{s}$	$60.8\mathrm{s}$
Wang	$32\mathrm{m}$	$6.1\mathrm{s}$	$1.4\mathrm{s}$	$74.0\mathrm{s}$
ADĂ	$32\mathrm{m}$	$6 \mathrm{h}$	$573\mathrm{s}$	$114\mathrm{s}$
BiMPM	$32\mathrm{m}$	$5\mathrm{h}$	$510\mathrm{s}$	$110.3\mathrm{s}$
SentenceBert	$32\mathrm{m}$	$5\mathrm{h}$	$986\mathrm{s}$	$144.2\mathrm{s}$
DupHunter	$39\mathrm{m}$	$6\mathrm{h}$	$742\mathrm{s}$	$143\mathrm{s}$

set reaches a certain number, the effectiveness of DupHunter is basically stable.

VI. DISCUSSION

A. Complexity Overhead

DupHunter has the complexity overhead for achieving accurate detection results. When using DupHunter in practice, we need to compare a submitted PR with all PRs or a subset of recent PRs in a repository, which can take several minutes. DupHunter has two phases, i.e., the training phase and the prediction phase. In our context, the prediction time complexity is more important. In the prediction phase, since attention weights are required to compute for every pair of nodes across two graphs, DupHunter has a computation cost of $O(|V_1||V_2|)$, where $|V_1|$ and $|V_2|$ represent the number of nodes in each graph. This time complexity is higher than non neural-network-based baselines (such as Ren and Wang), which cost is $O(N * \log(n))$, where N represents the number of weaker classifiers for Adaboost and n represents the number of features.

Table VIII shows the time spent in different stages for DupHunter and all baselines, including the preprocessing, training, testing, and inference time. The pre-processing time is to process all the raw data (i.e., PRs) in the dataset. It includes the time to transform PR pairs into the format that can be fed into each approach for duplicate PR detection. For the training time, the three baselines Li-T, Li-D, Li-TD do not need training, since they directly compute the similarity of PR pairs. For the baselines Ren and Wang, we use the default parameters set in Ren's replication package to compute the training time. For ADA, BiMPM, SentenceBert, and Dup-Hunter, all these models need to train iteratively. We train these models until the minimum loss defined in each model unchanged. It takes around 20 epochs. For testing time, we counted the time required to process all PR pairs in the test set, i.e., from feeding these PR pairs into a model to returning the ranking list.

We define the inference time as the time to infer duplicate PRs for a newly submitted PR. It measures how long an approach can return a list of potentially duplicate PRs for a new PR. Given a new PR, we need to first preprocess this PR. For baselines Li-T, Li-D, and Li-TD, they can directly compute the similarity between this PR and all the preprocessed PRs in a repository to return the ranking list. For the other approaches, they need to take the new PR and each PR in the repository as a pair to compute some features or networks. These models take the features or networks as inputs to compute the ranking list. The inference time is the average time to process one new PR. Since there are on average 3800 PRs in each testing repositories (i.e., repositories #13-#16), the inference time can show the time to return a duplicate PR list for a newly submitted PR in a repository in this scale.

For the preprocessing time, all approaches take less than 1 hour. The training time and testing time for Li-T, Li-D, Li-TD, Ren, and Wang are short, which take at most 6.1 s. All neural network based approaches are slow. They spend 5–6 hours on training and 510–986 seconds on testing. Regarding the inference time, all approaches (excluding approaches of Li's family) take at least 1 minutes to return a ranking list for a submitted PR.

In our experiment, DupHunter took 143 s to process all PR pairs in a repository for a newly submitted PR. However, this overhead can be acceptable in our application scenario. Traditionally, after submitting a PR, developers need to wait for hours or days to receive the feedback from project maintainers. With the help of DupHunter, developers can get some accurate feedback in a few minutes to further decide whether or not to confirm the submission. In addition, the recommendation list by DupHunter is helpful for project maintainers, since as a supplementary of project maintainers' experience, the list can narrow down the number of historical PRs to be analyzed. For these reasons, more accurate solutions could be preferred by different stakeholders.

Overhead Versus Benefit: To interpret the additional benefit improved by DupHunter, we analyze the number of PR pairs developers have to read to find all duplicate PR pairs in the test set, since all approaches rank potentially duplicate PR pairs in the test set of each repository. We assume that developers check PR pairs in the list from top to bottom [39]. If in our test set, an approach can rank all duplicate PRs to the top, developers require the minimum effort to find all duplicate PRs. We define the effort as follows: given a ranked list, the effort to find a duplicate PR pair equals with the position of this PR pair in the list. If a duplicate PR pair is not present in the list, the effort is the size of the list (as developers have to check the entire list). The effort to find all duplicate PR pairs is the sum of effort to find each duplicate PR pair in the test set. We define the additional benefit gained by DupHunter as $effort_{baseline} - effort_{DupHunter}$; it shows the number of PR pairs that developers can save time to check by using DupHunter.

As shown in Table IX, each cell represents the benefit obtained after replacing the baseline with DupHunter. When k=1, DupHunter can help developers read 147 to 5672 fewer PR pairs. When k=5, DupHunter can reduce the reading effort by at least 6424 PR pairs in the test set. This benefit is gained because DupHunter can rank duplicate PR pairs in the test set in a higher position compared to the baselines.

Regarding the overhead (i.e., the efficiency in our context), as shown in Table VIII, all approaches allow their models to conduct daily updates, empowered with the information derived from new PRs. Daily updates are a common setting for machine learning tools [40]. For the test time, neural network based

TABLE IX Additional Benefit Obtained by Using DupHunter

Baselines	k=1	<i>k</i> =2	<i>k</i> =3	k=4	<i>k</i> =5
Li-T	3085	12676	24251	36640	48965
Li-D	5672	21146	40192	60442	81229
Li-TD	3730	15015	28371	42438	56350
Ren	466	4102	10617	18896	28639
Wang	147	2361	7264	14024	22471
ADĂ	175	2211	4321	9642	15092
BiMPM	241	2734	6976	12344	18640
SentenceBert	207	1593	3148	4972	6424
DupHunter	/	/	/	/	/

TABLE X	
THE INFLUENCE OF DISTANCE MEASUREMENT]	FUNCTIONS

	P@1	R@1	F@1	MAP@1
Cosine Hamming Euclidean	0.922 0.866 0.922	0.502 0.466 0.502	0.650 0.606 0.650	0.905 0.801 0.905

approaches are slow, which take $510 \pm 0.742 \pm 0.0123$ (i.e., several minutes) for testing. For the inference time, only Li's approaches can suggest a list of duplicate PRs instantaneously given a newly submitted PR. However, the accuracy of Li's approaches is poor. All other approaches take more than 1 minutes. That means the tools will work in an offline manner. DupHunter takes 143 ± 0.0123 (i.e., between 2 and 3minutes) to return a list, which is around one minute's (i.e., 69 ± 0.0123 slow compared to the best baseline Wang. The overhead is comparable.

We have randomly selected 100 duplicate PRs to compute the time interval to label a PR as duplicate by developers manually. We find that the average time interval is 14 days with an average of 7 comments generated. Hence, it would be acceptable to help developers make decision with an overhead of the current inference time. DupHunter can be used for practical cases.

Although these benefit and overhead are analyzed, an empirical study could be needed to analyze whether this accuracy improvement benefits teams' performance in real-world settings (compared to the baselines and considering the overhead). We leave it as future work.

B. The Choice of Distance Functions

In the duplicate PRs detection component, we detect duplicate PRs by computing the euclidean similarity of the feature vectors of two PRs. To analyze the impact of distance measurement functions, we replace euclidean similarity with hamming similarity and cosine similarity. As shown in Table X, euclidean similarity and cosine similarity have the same performance,¹⁶ while hamming similarity is less effective. Hamming similarity calculates the number of different elements between two vectors. This method only measures whether the value of a dimension is the same or not, which cannot reflect the fine-grained 'distance' of each feature. In contrast, cosine similarity and euclidean similarity show better performance by calculating the cosine value of the angle and the linear distance between two vectors,

¹⁶For other k values, results are also similar.

	P@1	R@1	F@1	MAP@1
Oversampling	0.763	0.146	0.245	0.695
Undersampling	0.327	0.140	0.196	0.209
Loss weighting	0.740	0.373	0.496	0.694
DupHunter	0.922	0.502	0.650	0.905

respectively. Hence, other distance measurement functions such as cosine similarity can also be applied.

C. The Influence of Balancing Techniques

The dataset in the experiments is extremely unbalanced. We tested three classical imbalanced learning techniques, i.e., oversampling, undersampling, and loss weighting, on duplicate PRs detection. For oversampling, we replicated duplicate PR pairs in the training set to the same number as that of non-duplicate PR pairs in each repository. For undersampling, we randomly selected a subset of non-duplicate PR pairs based on the number of duplicate PR pairs in each repository. For loss weighting, we assign a higher weight to the loss function of DupHunter for the minority class (i.e., duplicate PR pairs). We implement loss weighting with the DSC (dice coefficient) loss weighting method [41]. We re-train DupHunter with the data after balancing.

As shown in Table XI, these balancing techniques do not improve the accuracy of DupHunter. We analyze the reasons as follows. First, oversampling and loss weighting attempt to increase the weight of positive instances (i.e., the minority class). However, methods such as replicating positive instances does not add new knowledge of duplicate PR pairs, which could also amplify the impact of the noise in positive instances. Second, undersampling discards a large amount of data, causing the overfitting issue. Since it is still an open question to select the suitable techniques for improving deep neural networks on significantly imbalanced training data [42], [43], we leave it as future work to improve DupHunter with advanced balancing techniques.

D. Practical Implication

This subsection discusses the practical implications of DupHunter for different stakeholders, including developers, project maintainers, and researchers.

Developers. Since fork-based development is a distributed development process, duplicate PRs are submitted by developers from all over the world. When developers submit a PR, after the analysis by DupHunter, a list of potential duplicate PRs can be provided to developers. By perusing this list, developers can deeply analyze the PR to be submitted in advance to avoid submitting a duplicate PR to the repository. In addition, since DupHunter provides a list of similar (i.e., potentially duplicate) PRs, another practical implication to developers is that developers can efficiently identify, trace, and investigate the status of a set of PRs focusing on similar issues. We remark

that DupHunter cannot avoid developers developing duplicate PRs; however, compared with the time cost and effort caused by subsequent discussions and rejections, DupHunter is still helpful to developers in the fork-based development.

Project Maintainers. Project maintainers may have to tackle with thousands of PRs submitted by developers. They need to manually analyze whether a submitted PR is duplicate with any PR in the repository according to their experience. This is a time-consuming and laborious process. With DupHunter, a list of potentially duplicate PRs is recommended to assist this process. Although the true duplicate PRs may not be always ranked as the first one(s), the list can narrow down the number of historical PRs to be analyzed. Therefore, DupHunter, as a supplementary of project maintainers' experience, is an effective tool for them to detect duplicate PRs.

Researchers. DupHunter analyzes software artifacts according to syntactic structure information and joint reasoning, which provides an innovation for solving the similarity measurement of other software artifacts (besides PRs). In addition, DupHunter detects duplicate PRs by calculating their similarity. The similarity of PRs can be used as inputs for other software repository mining tasks, such as identifying developers with similar interest and clustering PRs for repository statistical analysis.

E. Threats to Validity

1) Internal Threats: The main threat to internal validity is the correctness of the reproduced models (Li-T, Li-D, Li-TD [26], Ren [27], and Wang [28]). To alleviate this threat, we have double-checked our implementation of Li's approach and Wang's approach with reference to their papers. Additionally, we directly use the source code published by Ren et al. [27] to implement their approach.

2) External Threats: The first external threat is that we only conduct experiments on the repositories from GitHub, which may not represent all fork-based PRs. However, GitHub is one of the most commonly used OSS platforms. We evaluated DupHunter with 26 projects from different domains and using different programming languages in GitHub to alleviate this threat.

The second threat is that we only evaluate the ability of different approaches in distinguishing labeled duplicate PR pairs from a large collection of non-duplicate PR pairs, because of the huge cost of manually checking whether or not a given PR is duplicate with all the existing PRs in a repository. However, we believe our evaluation can reflect the effectiveness of an approach in helping project maintainers and developers efficiently detect duplicate PRs.

Third, our dataset is constructed by adding 100000 nonduplicate PR pairs into training and testing sets. The number of non-duplicate PR pairs can affect the effectiveness of DupHunter as evaluated in Section V-D. Since non-duplicate PR pairs are easy to create automatically, we can obtain enough non-duplicate PR pairs for training. Another threat is that non-duplicate PR pairs in our dataset are randomly created from the merged PRs without manual check. There may be potential duplicate PR pairs. Since two merged PRs are usually non-duplicate [27], we believe the ratio of duplicate PR pairs is small compared with the large number of non-duplicate PR pairs.

In the future, we plan to collect additional datasets from different open-source platforms and evaluate DupHunter in a real developing scenario.

3) Construct Threats: DupHunter extracts syntactic structural information from the *title* and *description* of a PR. We use the approaches proposed in the existing study [27] to process non-text features. In future work, we will design a model that also extracts the syntactic structural information from non-text features (e.g., the patch content). However, our current experiments demonstrate the importance of syntactic structural information and joint reasoning in detecting duplicate PRs.

VII. RELATED WORK

In this section, we discuss duplicate detection for different software artifacts, including duplicate PRs detection, duplicate questions detection in Stack Overflow, and duplicate bug reports detection.

A. Duplicate PRs Detection

There are mainly three approaches for duplicate PRs detection. Li et al. [26] calculate the similarity between titles and descriptions of PRs using TF-IDF to detect duplicate PRs. They examine the effectiveness of the similarity of *titles*, the similarity of *descriptions*, and the similarity of both *titles* and *descriptions* to indicate duplicate PRs.

Ren et al. [27] study additional features for detecting duplicate PRs. They manually check 45 pairs of duplicate PRs to identify seven new features (i.e., Feature#4–Feature#10 in Table I) for measuring the similarity of two duplicate PRs. Combining with the similarity of titles and descriptions, the values of the new features are used as input to train a classifier to predict whether a pair of PRs is duplicate or not. In a follow-up study, Wang et al. [28] explore the influence of the time interval (i.e., Feature#3 in Table I) on detecting duplicate PRs.

Different from existing studies, we construct feature graphs to adopt the syntactic structural information of text features and use a graph matching network to represent PRs. Our approach can generate vectors from feature graphs by considering the joint reasoning between PRs to further improve the accuracy of duplicate PRs detection.

B. Duplicate Questions Detection in Stack Overflow

Many duplicate questions are submitted to Stack Overflow daily. However, multiple duplicate questions cannot be detected simultaneously in Stack Overflow [44]. Muhammad et al. [45] investigate the reasons for submitting duplicate questions. They find that the most common reason is the lack of searching for existing questions and answers before submission. Zhang et al. [44] propose an initial solution for duplicate questions detection in Stack Overflow by measuring the similarity of different factors of a question (such as the title and the description). However, the effectiveness of this approach is limited since semantic information of questions could be lost. To further improve the detection accuracy, Mizobuchi et al. [46] use word embedding to overcome the word ambiguity when comparing the similarity of questions. Zhang et al. [47] compare the effectiveness of continuous word vectors, topic model, and frequent phrase pairs to capture the semantic similarity between questions. In recent studies, Wang et al. [48], [49] construct three deep learning networks, including WV-CNN, WV-RNN, and WV-LSTM, to detect duplicate questions, which are respectively based on the structure of convolutional neural networks, recurrent neural networks, and long short-term memory. Zhou et al. [50] add the attention mechanism to the deep neural networks.

DupHunter is different from these deep learning based approaches from two aspects. On the one hand, these approaches mainly analyze the title and the description of questions. However, a PR has lots of non-text information which drives us to conduct joint reasoning on PRs. On the other hand, we use a cross-graph calculation attention coefficient mechanism to capture the semantic similarity between PRs, which is not applied in the existing studies.

C. Duplicate Bug Reports Detection

Bug reports allow users and developers to provide timely feedback on bugs encountered when using software [51], [52]. Similar to fork-based development, different users and developers may submit duplicate bug reports.

To detect duplicate bug reports, various types of automated approaches have been proposed [53], [54], [55]. Most of the existing approaches use information retrieval to analyze duplicate bug reports. They compare the text information [51], [56] and the execution information [57], [58] in bug reports to detect duplicates. Besides, Ashima et al. [59] propose a model that uses a convolutional neural network to extract relevant features form bug reports. These features are then used to determine duplicate bug reports. Thiago et al. [60] proposed SiameseQAT which combines context and semantic learning and corpus-level topic extraction to improve the detection accuracy. In addition, Nathan et al. [61] combine visual information and textual information to help developers find duplicate video-based bug reports. Zhang et al. [62] summarize the related work on duplicate bug reports detection.

For bug reports, the main content is the *title* and the *description*. Approaches for duplicate bug reports detection may not work well in detecting duplicate PRs, because developers often rarely write particularly detailed descriptions in PRs, but upload the modified source code. In addition, duplicate bug report detection approaches usually construct features specified to bug reports, such as the differences of products, components, versions, priority of two bug reports. These features cannot be used for duplicate PRs detection. Hence, for the task of duplicate PRs detection, it is important to comprehensively consider the syntactic structural information and the joint reasoning of

different elements of PRs, which have not been considered in duplicate bug reports detection approaches.

VIII. CONCLUSIONS AND FUTURE WORK

Most open-source software projects are maintained based on an uncoordinated process called the fork mechanism, leading to many duplicate PRs. To help project maintainers and developers reduce the workload of analyzing duplicate PRs, in this paper, we propose DupHunter for duplicate PRs detection. DupHunter contains a graph embedding component and a duplicate PRs detection component, which are designed to respectively address the challenges of existing approaches, i.e., ignoring syntactic structural information and lacking joint reasoning between PRs. Experimental results on 26 open-source projects show that both components improve the effectiveness of DupHunter in detecting duplicate PRs. DupHunter outperforms the state-of-the-art approaches in terms of multiple evaluation metrics. It achieves a Precision@1 value of 0.922. Such results show the potential capability of DupHunter to reduce the development cost caused by duplicate PRs in the fork-based development process.

In the future, we plan to construct additional features and extract semantic structural information from them to build a more effective model. We also plan to use datasets from different open-source communities to further evaluate DupHunter.

REFERENCES

- G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 345–355.
- [2] K. Crowston, K. Wei, Q. Li, and J. Howison, "Core and periphery in free/libre and open source software team communications," in *Proc. IEEE* 39th Hawaii Int. Conf. Syst. Sci., 2006, pp. 118a–118a.
- [3] I. Steinmacher, T. U. Conte, C. Treude, and M. A. Gerosa, "Overcoming open source project entry barriers with a portal for newcomers," in *Proc.* 38th Int. Conf. Softw. Eng., 2016, pp. 273–284.
- [4] K.-J. Stol and B. Fitzgerald, "Two's company, three's a crowd: A case study of crowdsourcing software development," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 187–198.
- [5] C. Gutwin, R. Penner, and K. A. Schneider, "Group awareness in distributed software development," in *Proc. ACM Conf. Comput. Supported Cooperative Work*, 2004, pp. 72–81.
- [6] B. Lin, G. Robles, and A. Serebrenik, "Developer turnover in global, industrial open source projects: Insights from applying survival analysis," in *Proc. IEEE 12th Int. Conf. Glob. Softw. Eng.*, 2017, pp. 66–75.
- [7] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.
- [8] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proc. IEEE 17th Eur. Conf. Softw. Maintenance Reeng.*, 2013, pp. 25–34.
- [9] J. Bitzer and P. J. H. Schrder, "The impact of entry and competition by open source software on innovation activity," *Econ. Open Source Softw. Develop.*, vol. 6, pp. 219–246, 2006.
- [10] N. A. Ernst, S. M. Easterbrook, and J. Mylopoulos, "Code forking in opensource software: A requirements perspective," 2010, arXiv:1004.2889.
- [11] G. R. Vetter, "Open source licensing and scattering opportunism in software standards," Rochester, NY, USA: Social Science Electronic Publishing, 2008.
- [12] GitHub, 2008. Accessed: Jun. 4, 2022. [Online]. Available: https://github. com/
- [13] GitLab, 2011. Accessed: Jun. 4, 2022. [Online]. Available: https://gitlab. com/

- [14] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 805–816.
- [15] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. IEEE/ACM 26th Int. Conf. Automated Softw. Eng.*, 2011, pp. 253–262.
- [16] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang, "Determinants of pull-based development in the context of continuous integration," *Sci. China Inf. Sci.*, vol. 59, no. 8, pp. 080104:1–080104:14, 2016.
- [17] Rails, 2015. Accessed: Jun. 4, 2022. [Online]. Available: https://github. com/rails/rails/pulls/
- [18] S. Zhou, B. Vasilescu, and C. Kästner, "What the fork: A study of inefficient and efficient forking practices in social coding," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 350–361.
- [19] Z. Li et al., "Redundancy, context, and preference: An empirical study of duplicate pull requests in OSS projects," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1309–1335, Apr. 2022.
- [20] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open source software projects," in *Proc.* 40th Int. Conf. Softw. Eng., 2018, pp. 256–266.
- [21] G. Gousios, M.-A. D. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proc. IEEE 38th Int. Conf. Softw. Eng.*, 2016, pp. 285–296.
- [22] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-I. Matsumoto, "Who should review my code? A file location-based code-reviewer recommendation approach for modern code review," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reeng.*, 2015, pp. 141–150.
- [23] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?," *Inf. Softw. Technol.*, vol. 74, pp. 204–218, 2016.
- [24] Y. Yu, Z. Li, G. Yin, T. Wang, and H. Wang, "A dataset of duplicate pullrequests in GitHub," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 22–25.
- [25] W. Huang, T. Lu, H. Zhu, G. Li, and N. Gu, "Effectiveness of conflict management strategies in peer review process of online collaboration projects," in *Proc. 19th ACM Conf. Comput. Supported Cooperative Work Social Comput.*, 2016, pp. 715–726.
- [26] Z. Li, G. Yin, Y. Yu, T. Wang, and H. Wang, "Detecting duplicate pullrequests in GitHub," in *Proc. 9th Asia-Pacific Symp. Internetware*, 2017, pp. 20:1–20:6.
- [27] L. Ren, S. Zhou, C. Kästner, and A. Wasowski, "Identifying redundancies in fork-based development," in *Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reeng.*, 2019, pp. 230–241.
- [28] Q. Wang, B. Xu, X. Xia, T. Wang, and S. Li, "Duplicate pull request detection: When time matters," in *Proc. 11th Asia-Pacific Symp. Internetware*, 2019, pp. 8:1–8:10.
- [29] Code and datasets, 2022. [Online]. Available: https://github.com/ ILikeCode0/Dup-Hunter
- [30] L. A. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *Proc. ACM Conf. Comput. Supported Cooperative Work*, 2012, pp. 1277–1286.
- [31] S. Zhou, S. Stanciulescu, O. Leßenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 105–116.
- [32] Y. Zhang, X. Yu, Z. Cui, S. Wu, Z. Wen, and L. Wang, "Every document owns its structure: Inductive text classification via graph neural networks," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 334–339.
- [33] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in Proc. 33rd AAAI Conf. Artif. Intell., 2019, pp. 7370–7377.
- [34] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proc.* 36th Int. Conf. Mach. Learn., 2019, pp. 3835–3845.
- [35] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proc. 4th Int. Conf. Learn. Representations* (*Poster*), 2016.
- [36] D. Shah, T. Lei, A. Moschitti, S. Romeo, and P. Nakov, "Adversarial domain adaptation for duplicate question detection," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2018, pp. 1056–1063.
- [37] Z. Wang, W. Hamza, and R. Florian, "Bilateral multi-perspective matching for natural language sentences," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 4144–4150.

- [38] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Proc. Conf. Empir. Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process.*, 2019, pp. 3982–3992.
- [39] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
- [40] H. Belgacem, X. Li, D. Bianculli, and L. Briand, "A machine learning approach for automated filling of categorical fields in data entry forms," *ACM Trans. Softw. Eng. Methodol.*, to be published, doi: 10.1145/3533021.
- [41] X. Li, X. Sun, Y. Meng, J. Liang, F. Wu, and J. Li, "Dice loss for dataimbalanced NLP tasks," in *Proc. 58th Annu. Meeting Assoc. Computat. Linguistics (ACL)*, 2020, pp. 465–476.
- [42] Q. Dong, S. Gong, and X. Zhu, "Imbalanced deep learning by minority class incremental rectification," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 6, pp. 1367–1381, Jun. 2019.
- [43] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," J. Big Data, vol. 6, no. 1, pp. 1–54, 2019.
- [44] Y. Zhang, D. Lo, X. Xia, and J. Sun, "Multi-factor duplicate question detection in stack overflow," *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 981–997, 2015.
- [45] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Mining duplicate questions in stack overflow," in *Proc. IEEE 13th Work. Conf. Mining Softw. Repositories*, 2016, pp. 402–412.
- [46] Y. Mizobuchi and K. Takayama, "Two improvements to detect duplicates in stack overflow," in *Proc. IEEE 24th Int. Conf. Softw. Anal. Evol. Reeng.*, 2017, pp. 563–564.
- [47] W. E. Zhang, Q. Z. Sheng, J. H. Lau, and E. Abebe, "Detecting duplicate posts in programming QA communities via latent semantics and association rules," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 1221–1229.
- [48] L. Wang, L. Zhang, and J. Jiang, "Duplicate question detection with deep learning in stack overflow," *IEEE Access*, vol. 8, pp. 25964–25975, 2020.
- [49] L. Wang, L. Zhang, and J. Jiang, "Detecting duplicate questions in stack overflow via deep learning approaches," in *Proc. IEEE 26th Asia-Pacific Softw. Eng. Conf.*, 2019, pp. 506–513.
- [50] Q. Zhou, X. Liu, and Q. Wang, "Interpretable duplicate question detection models based on attention mechanism," *Inf. Sci.*, vol. 543, pp. 259–272, 2021.
- [51] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. IEEE 29th Int. Conf. Softw. Eng.*, 2007, pp. 499–510.
- [52] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 461–470.
- [53] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 308–311.
- [54] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. 32th Int. Conf. Softw. Eng.*, 2010, pp. 45–54.
- [55] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. IEEE 27th Int. Conf. Automated Softw. Eng.*, 2012, pp. 70–79.
- [56] L. Hiew, "Assisted detection of duplicate bug reports," M.S. thesis, Univ. Brit. Columbia, Vancouver, Canada, 2006.
- [57] A. Podgurski et al., "Automated support for classifying software failure reports," in Proc. IEEE 25th Int. Conf. Softw. Eng., 2003, pp. 465–477.
- [58] Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei, "JDF: Detecting duplicate bug reports in jazz," in *Proc. IEEE 32nd Int. Conf. Softw. Eng.*, 2010, pp. 315–316.
- [59] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K.-S. Kwak, "Duplicate bug report detection and classification system based on deep learning technique," *IEEE Access*, vol. 8, pp. 200749–200763, 2020.
- [60] T. M. Rocha and A. L. da Costa Carvalho, "SiameseQAT: A semantic context-based duplicate bug report detection using replicated cluster information," *IEEE Access*, vol. 9, pp. 44610–44630, 2021.
- [61] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "It takes two to TANGO: Combining visual and textual information for detecting duplicate video-based bug reports," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 957–969.
- [62] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Sci. China Inf. Sci.*, vol. 58, no. 2, pp. 1–24, 2015.



He Jiang (Member, IEEE) received the Ph.D. degree in computer science from the University of Science and Technology of China, Hefei, China. He is currently a professor with the Dalian University of Technology, Dalian, China. He is also a member of the ACM and the CCF (China Computer Federation). He is one of the ten supervisors for the Outstanding Doctoral Dissertation of the CCF in 2014. His current research interests include intelligent software engineering, software testing with focus on system software, and search-based software engineering (SBSE).

His work has been published at premier venues like ICSE, FSE, and ASE, as well as in major IEEE Transactions like the *IEEE Transactions on Software Engineering, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Cybernetics, and IEEE Transactions on Services Computing.*



Yulong Li received the bachelor's degrees from Dalian maritime University, in 2022. He is currently working toward the master degree with the College of Intelligence and Computing, TianJin University, China. His current research interests include intelligent software engineering, machine learning, and software testing.



Shikai Guo received the B.S. degree in computer science and technology from Liaoning University, Shenyang, China, in 2011, and the Ph.D. degree in computer applications technology from Dalian Maritime University, Dalian, China, in 2018. He is currently an Associate Professor with the College of Information Science and Technology, Dalian Maritime University, Dalian, China. His current research interests include intelligent software engineering, software testing, and program analysis techniques.



Xiaochen Li received the Ph.D. degree in software engineering from the Dalian University of Technology, China, in 2019, under supervision with Prof. He Jiang. He is an Associate Professor with the School of Software, Dalian University of Technology. He was a research associate with the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, headed by Prof. Lionel Briand. His current research interests include intelligent software engineering and software testing. His work has been published at premier venues like the IEEE TRANS-

ACTIONS ON SOFTWARE ENGINEERING, ACM Transactions on Software Engineering and Methodology, and ICSE. For more information, please visit https://xiaochen-li.github.io.



Tao Zhang (Senior Member, IEEE) received the B.S. degree in automation and the M.Eng. degree in software engineering from Northeastern University, China, and the Ph.D. degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an Associate Professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin

Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 70 high-quality papers at renowned software engineering and security journals and concerences such as the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON INFOR-MATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON RELIABILITY, ICSE, etc. His current research interests include AI for software engineering and mobile software security. He is a senior member of the ACM.



Rong Chen (Member, IEEE) received the M.S. and Ph.D. degrees in computer software and theory from Jilin University, Changchun, China, in 1997 and 2000, respectively. He is currently a professor with the College of Information Science and Technology, Dalian Maritime University, Dalian, China, and was previously with Sun Yat-sen University, Guangzhou, China. His research interests include soft, ware diagnosis, collective intelligence, activity recognition, and Internet and mobile computing. He is a member of the ACM.



Hui Li received the Ph.D. degree in computer architecture from Northeastern University, Shenyang, China, in 2013. He is an Associate Professor with the School of Information Science and Technology, Dalian Maritime University. His current research interests include intelligent software engineering, mining software repositories, and recommendation system.