An Empirical Study On Duplicate Bug Report Identification Using Siamese Cross-Encoder Network

Parvez Mahbub
Faculty of Computer Science
Dalhousie University
Halifax, Canada
parvezmrobin@dal.ca

Ohiduzzaman Shuvo
Faculty of Computer Science
Dalhousie University
Halifax, Canada
oh599627@dal.ca

Usmi Mukherjee
Faculty of Computer Science
Dalhousie University
Halifax, Canada
usmi.mukherjee@dal.ca

Sigma Jahan
Faculty of Computer Science
Dalhousie University
Halifax, Canada
sigma.jahan@dal.ca

***Abstract*—Resolving the issue of duplicate bug reports is a massive blockage for any software company since it wastes a significant amount of time and resources. Hence automation of duplicate bug report detection has been significant research interest in software engineering and natural language for decades. Several algorithms from various fields such as Natural Language Processing, Information Retrieval, and Machine Learning have been examined to address this problem. However, the outcomes have not proven very reliable or effective in natural world settings in most situations. Furthermore, with the revolution of deep learning, researchers tend to use fairly complex models without strong reasoning. These complex models might improve the performance but severely impacts software's portability, availability, and explainability. This project uses a simplified version of a former duplicate bug report identification system based on the siamese network, recurrent neural network (LSTM), and convolutional neural network (CNN). We show that our simpler model has the same performance in most cases (0.91 f_1-score).**

***Index Terms*—bug report, duplicate bug report, software engineering, natural language processing, machine learning, siamese network, recurrent neural network, long short-term memory, convolutional neural network**

I. Introduction

Software maintenance relies heavily on bug reports. Bug reports are one of the significant artifacts which are reported during the development or maintenance phase of the software. Previously, it has been reported that almost two-thirds of the whole effort is accounted for in the maintenance phase of the software development process [1]. Most of the large software projects maintain the bug reports by hosting a bug tracking system like Bugzilla. However, the submitted reports do not always fulfill the expectation of the tester. Consequently, duplicate bug reports are widespread in the bug report maintenance system. Duplicate bug reports occur when multiple users file bug reports for the same concern, and it is a massive issue in any software company. Cavalcanti et al. [2] reported in their study, the number of duplicate bug reports can be almost half of the total bug reports. Therefore, detecting duplicate bug reports has always been a major concern in software development. Besides, the detection of duplicate bugs may also help the software developer gather complementing information about the bug, increasing the overall productivity.

According to the study of Gupta et al., [3], detecting duplicate bug reports is more costly than reporting a new bug. Hence, manual examination of duplicate bug reports is not practical since it would require excessive time and expense. As a result, developing an automated approach to tackle the challenge of duplicate bug reports has become a rapidly growing concern to address. Unsupervised feature learning and deep learning based neural networks [4] have grown popular in recent years. Contemporary neural network models have been introduced for various tasks, which include but are not limited to image processing, recommendation systems, network security, speech recognition, and translation. Despite several acknowledged methodologies from several domains being adopted to automate the process of duplicate bug report detection, the performance of the existing approaches has not yet achieved that position to implement in the industry. Bug reports may contain structured information (stack trace) and unstructured information (natural language texts). There is a linguistic ambiguity and variability problem in natural language comprehension, making it difficult to detect similar reports [5]. On the other hand, structured information like stack trace cannot parse like natural language as they can contain code snippets from the source code. Although Cavalcanti et al. [6] explored the impact of various factors such as project lifetime, staff size, and the number of bug reports on identifying duplicate bug reports, the impact of structured information has not been explored yet.

Deshmukh et al., [1] recently proposed an automated approach by combining Siamese Convolutional Neural Networks (CNN) and Long Short Term Memory (LSTM) for accurate detection and retrieval of duplicate and similar bugs. They used structured and unstructured information separately from the bug report in their proposed model, and their model showed a significant performance in duplicate bug detection. However, the impact of the structured information in identifying duplicate bug reports has not been analyzed in their study. Hence, understanding the impact of structured information on duplicate bug report detection is a significant research problem to explore that could provide helpful insight to improve the existing techniques of duplicate bug report detection.

This paper is written as a part of the course – "CSCI6515 - Machine learning for Big Data".

With a view to understanding the impact of structured information on duplicate bug report detection, our study aims to answer two research questions.

- **RQ$_1$**: To what extent Siamese Cross-Encoder Network perform in detecting duplicate bug reports?
- **RQ$_2$**: Does handling structured information of bug reports separately has any impact on duplicate bug detection?

To identify the duplicate bug from a bug report database, our proposed approach employs Siamese Neural Networks [7] along with Bidirectional Long Short Term Memory (Bi-LSTM) being inspired by Deshmukh et al. [1]. Experimenting using three substantial published datasets [8], our proposed model obtained roughly 90% accuracy, precision, recall and F$_1$-score for both Eclipse and NetBeans dataset, which also aligns with the previous findings of Deshmukh et al. [1]. Furthermore, our study also suggests that handling structured information of bug reports does not always impact duplicate bug detection. However, if the bug repository is maintained accordingly, structured information can improve duplicate bug report identification.

The following is a representation of the framework of the paper. Section II presents contemporary researches on detecting duplicate bug reports. In section III, we explain the background study that requires for our proposed approach. The proposed methodology for duplicate bug report detection has been described in section IV. Section V contains our experimental design, and the results have been reported in section VI. The study concludes with section VII, which outlines some potential future research on the topic of duplicate bug report identification.

II. Related Works

Duplicate bug report detection has been a significant research interest for decades, and the study is being conducted in various approaches. The present research trend may be divided into two groups to simplify it. The first one is based on ranking and similarity scores with the specified query bug report, and an unsupervised algorithm generates a cluster of highest related bug reports [9]. Using this unsupervised method based on similarities, several duplicate bug reports may be readily found for a single bug report. Supervised schemes, the second group, essentially generates a binary classifier. Before submitting a bug report for triage, the classifier predicts if two bug reports are similar or not. This is how it avoids creating a new duplicate bug report in the bug tracking repository.

In the unsupervised approach, Runeson et al. [10] utilized a simple natural language processing approach called bag of words (BOW) to tally the frequency of words in a text when researching efficient techniques to detect duplicate bug reports. The similarity between bug reports is determined by applying cosine, Jaccard, and dice similarity metrics to calculate the similarity between the feature vectors. The topmost similar bug reports were retrieved based on the similarity scores. The proposed natural language-based approach, on the other hand, is unduly simplistic and lacks precision. Another study [11] improved on this research by using TF-IDF instead of BOW to quantify the similarity of two vectors while also taking execution information into account as an additional input. The results were marginally better but still insufficient for practical application.

Similarly, a ranking function, BM25 [12], was implemented in a study for duplicate bug report detection for the weighting strategy for duplicate bug retrieval. Nevertheless, another research [13] demonstrated that BM25F, an improvement of BM25, is preferred for weighting words in diverse domains. They also focused on taking domain-specific categorical and textual features into account, which only improved efficiency by 3.8% to 10.8% for each project. In a study article [14], Latent Dirichlet allocation (LDA) is developed for analyzing contextual information of bug reports using information retrieval as the preventative strategy, expanding the work of adopting BM25F.

Furthermore, another research [15] has provided a method involving information retrieval and machine learning for dealing with duplicate bug reports, focusing on both with and without a shared vocabulary. When the vocabulary was shared, they employed the vector space model (TF-IDF) and clustering approach (K-Means) and calculated similarity using Euclidean distance. Using vector space model with TF-IDF, they found the dissimilar duplicate bug reports by considering the angles. Next, after creating the word co-occurrence model, they employed the ranking function BM25 and language modelling approaches (Jelinek-Mercer smoothing and Dirichlet smoothing) to compute the similarity. This approach needs an adequately labelled dataset as it uses machine learning.

Moving on to supervised techniques, they advocated in the study [16] to leverage the Support Vector Machine (SVM) approach to learn the REP parameters. To solve the concerns of an unbalanced dataset, they limited the number of non-duplicates for training purposes. A somewhat different technique [17] divides the bug report collection into two categories depending on duplicate and non-duplicate reports before extracting features and translating them into feature vectors. Both sections are then trained using Support Vector Machine (SVM) to create a discriminative model that predicts two input bug reports' chance of duplicating [17]. However, there appears to be minimal compared with other existing models in this study to validate the proposed framework. The research [18] used supervised learning approaches to find the semantic similarity between two reports after finding feature sets in vector form. They implemented models like K-nearest neighbour, linear SVM, restricted Boltzmann Function (RBF), SVM, decision tree, random forest, and Naive Bayes for binary classification. They then compared their performances based on the metric evaluation recall.

A recent paper [19] introduced SiameseQAT, a unique tactic that incorporates the attention mechanism BERT for both semantic and contextual learning-based textual embedding. They also proposed a new loss function for the replicated duplicate cluster called Quintet Loss. For addressing duplicate bug reports with structured and unstructured information, their proposed retrieval and classification model obtained 85%

recall@25 for retrieval and 84% as the area under the receiver operating characteristics (AUROC) score for classification. However, their proposed scheme needs to be validated across domains and with more data to determine the model's generalization ability as a separate project depiction of duplicate reports.

III. Background

In this section, we cover some important concepts that we use in our work. We start by describing two important numeric representations of textual data, namely – vectorization and embedding. We then describe three architectures based on artificial neural networks, namely – recurrent neural network (RNN), convolutional neural network (CNN), and siamese network.

A. Vectorization

Vectorization is the process of transforming text data into a numerical representation which is a machine-understandable format of textual representation [20]. It is a common approach of converting input data from its raw format into vectors of real numbers that deep learning models support. It is used as a feature extraction technique to get distinct features out of the text [21]. While vectorization can represent words as numbers, it cannot capture the semantic meaning of the words. Furthermore, vectorized texts are often expressed as one-hot vectors where the word-indices contain 1s, and all the remaining indices are 0s. This representation is too sparse and is not a good use of computer memory.

B. Embedding

Word embedding captures both the semantic and syntactic information of words from a corpus [22]. Vector space representations of words have captured fine-grained semantic and syntactic regularities using vector arithmetic. GloVe is one such pre-trained model for obtaining vector representation of words [23]. The words in the vocabulary are assigned represented as feature vectors, and the probability distribution of word sequences is expressed in terms of these. The word feature vectors and parameters of the probability function are learned together by training a suitable feed-forward neural network [1].

C. Recurrent Neural Network

The Recurrent Neural Network (RNN) is a neural sequence model that achieves *state-of-the-art* performance on essential tasks that include language modelling [24]. It allows previous outputs to be used as inputs while having hidden states. It mainly uses the information present in the text from its numerical representation that is used as input to the neural network, and the computed output is then used to compute the output of the next word. They are known as recurrent since the same computation is performed for every sequence element using the output from the previous computation. These are also called Vanilla RNNs that have a couple of drawbacks. They have difficulty in learning long-term dependencies in the sequence via gradient descent training and have both vanishing and exploding gradient problems [25]. Long short term memory (LSTM), a variant of RNN, is shown to be effective in capturing dependencies and easier to train compared to vanilla RNNs [26]. Another extension of LSTM is the Bidirectional LSTMs. They improve the model's performance on sequence classification problems by training two instead of one LSTMs on the input sequence. Apart from the first sequence, it also takes the reversed sequence as input incorporating both future and past context, known as bidirectional LSTM (Bi-LSTM). This bidirectionality gives the network better context and results [27].

D. Convoluational Neural Network

Convolutional Neural Networks have recently found prevalence in tackling NLP tasks like text classification problems. We use Convolutional Neural Network (CNN) for handling such long texts like bug descriptions since LSTM can only capture the dependencies of small sequences [1]. CNNs are multilayered artificial neural networks that detect complex features in textual data. It consists of two main layers called convolution and pooling that act as feature extractors. The convolution is the process through which important features are extracted from the text. The outputs generated are concatenated, and max-pooling is applied. In our case, for text, we have one-dimensional input, which is the concatenation of the vector representation of the words. The convolution begins from one end of the text, keeping a window size of a certain length and computes the output. The same operation is repeated with different parameters [28]. The encoding that is generated finally is used for the prediction by the softmax layer that creates a mapping of semantically similar words [29].

E. Siamese Network

A Siamese Neural Network is a type of neural network architecture containing two or more subnetworks with the same configuration and parameters. It is generally used to find similarities between the inputs by comparing their vector representations [30]. It learns representations that embody the dissimilarities and selects desired data through explicit information about the similarity between the pairs of texts. The network learns an invariant and selective representation with the help of similarity and dissimilarity information. Along with the convolutional layer, these networks are used to find similarities and learn semantic entailment. In our case, we use a combination of Siamese LSTM and CNN to distinguish similar from non-similar bugs.

IV. Duplicate Bug Report Identification Using Siamese Cross-Encoder Network

We implement our duplicate bug report identification problem as classification. Our model expects a pair of bug reports and outputs whether the bug reports in this pair are duplicates or not. We train the model using triplets of bug reports where the first two elements of the triplet are two bug reports, and the

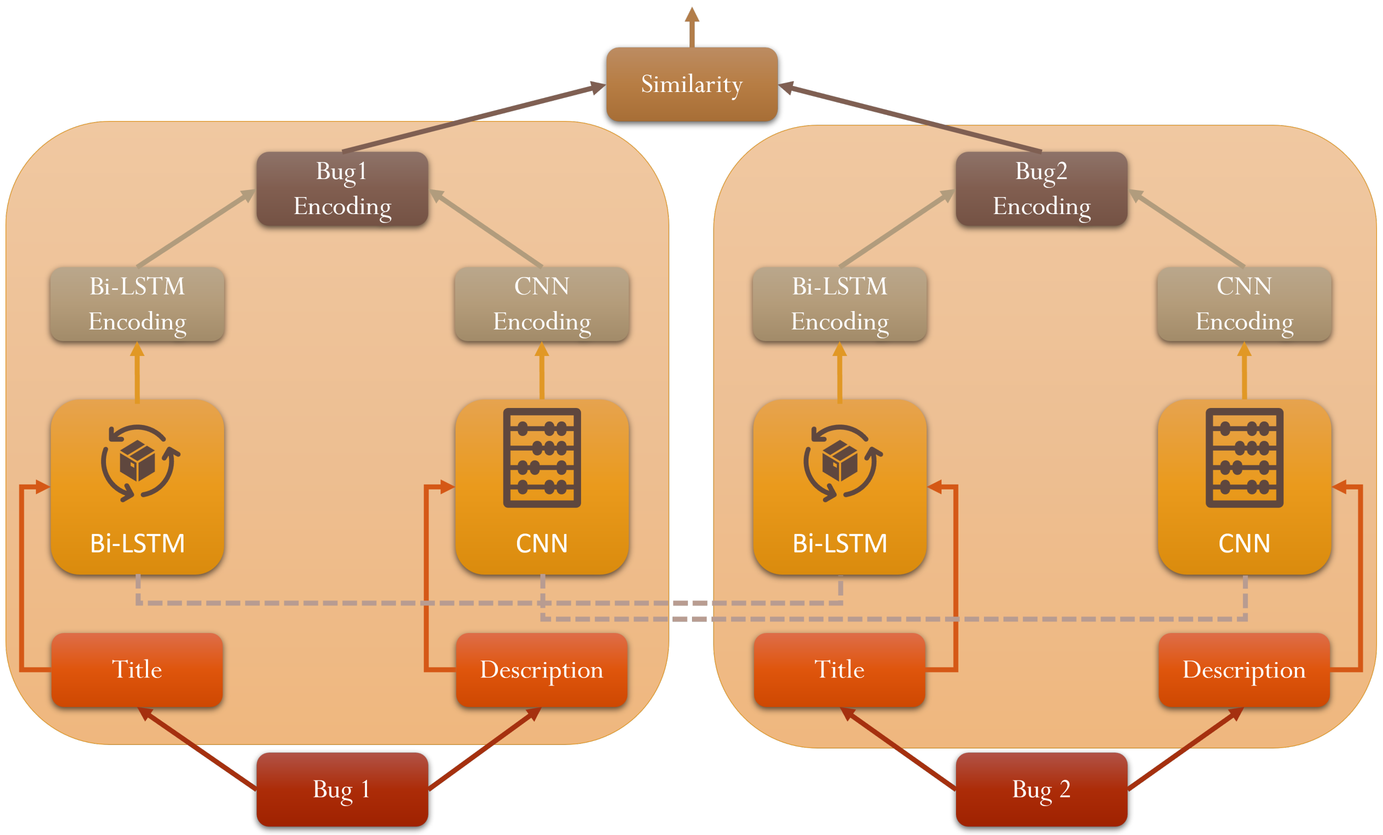

Fig. 1. The Complete Architecture of The Siamese Cross-Encoder

third element is the label defining whether they are duplicates or not. Over the training period, the model discriminates between similar and dissimilar pairs. This strategy is known as cross encoding.

Figure 1 shows the high-level architecture of our siamese cross-encoder network. First of all, we vectorize the title and description of a bug report. Then, we pass these vectorized texts to corresponding embedding layers to generate GloVe embedding for both title and description. Then, we pass the title-embedding to a bidirectional LSTM (Bi-LSTM) to generate a single encoding for all the title words. Simultaneously, we pass the description-embedding to a CNN network to generate another encoding representing the full description. Unlike the work of Deshmukh et al. [1], our description contains the structured information as well, and the CNN learns to properly focus and extract the necessary information from there as well. Once we have the encoding for both title and description, we generate a new encoding by concatenating these two. If E_t be the title encoding, and E_d be the description encoding, then this layer gives us the bug encoding defined as $E_B \longleftarrow E_t.E_d$. We repeat the same operation for the other bug report. The gray dotted lines in figure 1 denote that the Bi-LSTM and CNN used in encoding two bug reports share the same weights and parameters, essentially following a siamese architecture [31].

Once we have the encoding for both bug reports, we compute the cosine similarity between the pair of bug reports. Cosine similarity is a common approach to compute the similarity between numerical vectors. It is defined as

$$cosine(x,y) = \frac{x.y}{|x||y|} \tag{1}$$

Then, this cosine similarity is passed to a sigmoid function to compute the final prediction. The sigmoid function normalizes the input value to range (0, 1), ideal for predicting probability. The sigmoid function is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2}$$

We train the model using this method to place the bug in latent feature space so that similar bugs will have a high similarity score and dissimilar bugs will have a low similarity score. The distinguishing features which facilitate it are learned for the encodings from the supervised training using the binary cross entropy loss which is defined as

$$CE(y) = \frac{-1}{N}\sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1-y_i)\cdot\log(1-p(y_i)) \tag{3}$$

During the training phase, the model tries to minimize the cross-entropy loss using the Adam optimizer [32].

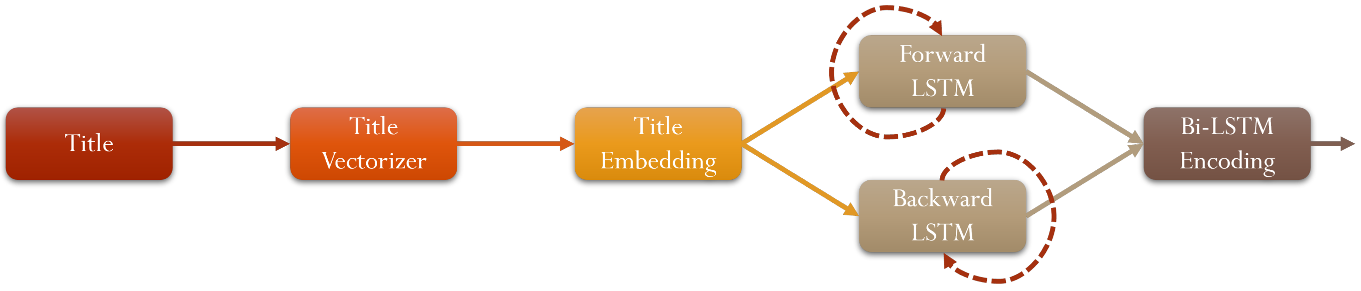

Fig. 2. The LSTM Network For Encoding Titles

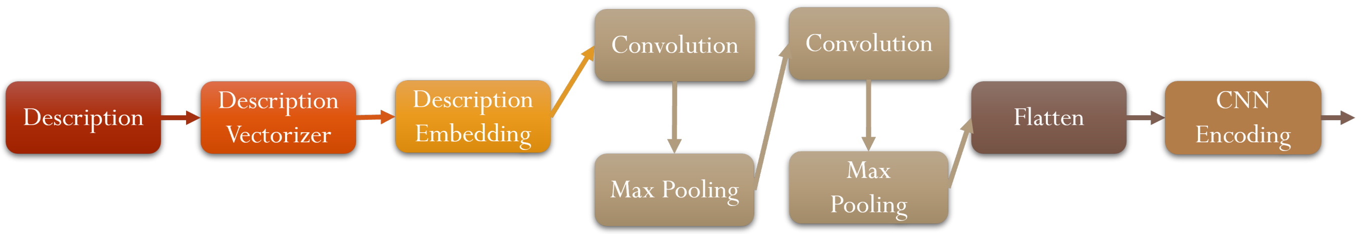

Fig. 3. The CNN Network For Encoding Descriptions

V. Experimental Design

This section describes the data collection and preprocessing of our dataset that we use to evaluate our proposed model (section IV). We implement our proposed network using TensorFlow [33] – an open-source library for implementing deep neural networks. We conduct all experiments in a Lenovo computer with a Core-i7 CPU (6 cores) and 16 GB primary memory. Due to the memory limitation of our system, we pass the data to our model in batches of size 2048.

A. Dataset

We use three large datasets of duplicate bug pairs created by Lazar et al. [34]. These datasets include bug reports with duplicate annotation from three leading open source projects, namely – 'Eclipse', 'NetBeans', and 'Open Office'. Eclipse and NetBeans are integrated development environments (IDE) for the Java programming language. Open Office is an open-source productivity software similar to Microsoft Office.

Table I presents a sample bug from these datasets. From the table, we see the datasets have a number for attributes describing the bug reports, among which "bug id" "sort desc", "description", "dup id", and "resolution" are the essential attributes in our work. The datasets also have a triplet version in the form $(b_1, b_2, sign)$ where b_1 and b_2 are two bug id, and $sign$ is either 1 or -1, denoting whether they are duplicate or not. This triplet version facilitates us to implement the duplicate bug report identification problem in a classification manner where a bug report pair comes as input and the model tries to predict whether they are duplicate or not. Table II contains the number of bug reports and triplets for each dataset.

TABLE I
Sample Bug Report

"bug id"	"2521"
"product"	"Writer"
""description"	"Opening a file from the file history that has been moved or deleted causes an error dialogue to pop up saying it doesn't exist etc. Upon clicking OK the dialogue comes up again. This continues indefinitely meaning the application has to be terminated."
"bug severity"	"trivial"
"dup id"	"2268"
"short desc"	"Opening a recent doc that has since been moved / deleted causes unfulfilled loop"
"priority"	"P3"
"version"	"641"
"component"	"ui"
"delta ts"	"2003-09-08 16:56:16 +0000"
"bug status"	"CLOSED"
"creation ts"	"2001-12-12 17:00:00 +0000"
"resolution"	"DUPLICATE"

TABLE II
Number Of Bug Reports And Triplets In The Datasets

Dataset	**#Bug Reports**	**#Triplets**
Eclipse	361,006	271,098
NetBeans	216,715	238,584
Open Office	98,070	152,872

B. Data Preprocessing

We create a new dataset from the bug reports and triplets during the preprocessing phase. This new dataset contains title and description from both b_1 and b_2 and corresponding sign. The dataset contains a significant amount of non-ASCII characters. We replace each non-ASCII character with the corresponding ASCII version if they have one, or just stripped them out using Unidecode python module [35]. For instance, $\hat{a}$ has a corresponding ASCII character a, but σ does not have any. Therefore, we replace $\hat{a}$ with a and stripe out σ. After that, we keep only the rows where both titles have at least ten characters, and both descriptions have at least 50 characters.

At this stage, we analyze the statistical distribution of the datasets to facilitate subsequent processing. We see that title length has a mean of 55 characters and 3^{rd}-quartile at 68 characters. However, the maximum title length is 255, definitely an outlier. Similarly, for description, the mean length is roughly 1460 characters, 3^{rd}-quartile at roughly 940 characters, and a maximum of 373,075 characters. Therefore, we perform outlier analysis for both title and description. We find that roughly 2% titles and 14% descriptions have outlier-lengths.

Following that, we make train, validation, and test splits for each dataset. Since we deal with big data, we do not use the traditional 2:1 holdout approach. Instead, we keep 20,000 instances for both test and validation and use the rest for training. Table III contains the train, validation, and test split sizes for each dataset.

TABLE III
NUMBER ON INSTANCES IN TRAIN, VALIDATION, AND TEST SPLITS

Dataset	**#Train**	**#Validation**	**#Test**
Eclipse	216,649	20,000	20,000
NetBeans	187,824	20,000	20,000
Open Office	105,480	20,000	20,000

Afterwards, we compute word-level tokens from the titles and the descriptions of the training data. We keep top-most 20,000 tokens in both title-vocabulary description-vocabulary. Then we replace each token in the titles and description with the corresponding word-index in the vocabulary. Based on the aforementioned analysis, we limit the titles to 100 characters and descriptions to 1000 characters which roughly corresponds to 21 words and 200 words. As mentioned in section III-A, this operation is known as vectorization. Then we pass the vectorized datasets to the siamese network to train and evaluate.

C. Network Implementation

Figure 2 and 3 illustrate the building blocks of LSTM and CNN components of our proposed siamese network. We use the best parameter combination to evaluate our model from the previous studies. As we use the GloVe embedding, our embedding dimension is 300. In the Bi-LSTM, both of the LSTMs have sequence lengths of 100. For CNN, we use 32 filters in each convolutional layer. The filters are 3×3 matrix for both convolutional layers. The first max-pooling layer has a 4×4 size pooling matrix and stride of 4. The second max-pooling layer has a 2×2 size pooling matrix and stride of 2.

D. Evaluation Metrics

As described in section V, we implement our siamese cross-encoder based duplicate bug report identification system as a classification problem. To evaluate the system, we use five different metrics. Among them, four are numerical, namely – accuracy, precision, recall, and F_1-score. The last one is a graphical metric, which is receiver operating characteristics. We describe each of them below.

1) Accuracy: Accuracy score is the ratio between the number of correctly identified test instances and the total number of test instances. Mathematically, accuracy score is defined as

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4}$$

where TP is the number of correctly identified true instance (i.e. duplicate pair), TN is the number of correctly identified false instance (i.e. non-duplicate pair), FP is the number of instances incorrectly identified as true, and FN is the number of instances incorrectly identified as false.

2) Precision: Precision is the measurement of if the model identify an instance is true, how precisely it can identify it. Mathematically,

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

In the case of our problem, precision measures the probability of two bug reports being duplicate when our siamese network identifies them as duplicates.

3) Recall: Recall is the measurement of how much true instances the model can identify from all the true instances. Mathematically,

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

In the case of our problem, recall measures if two bug reports are duplicates of each other, what is the probability of the siamese network identifying them.

4) F_1-Score: it is the harmonic mean of precision and recall. Mathematically,

$$F_1 = \left(\frac{precision^{-1} + recall^{-1}}{2}\right)^{-1} \tag{7}$$

The reason for using the harmonic mean over arithmetic mean is that the former is less sensitive to extreme values. If a dataset suffers from class imbalance, generally F_1-score is more reliable than the accuracy score. For instance, a test set has 90 false and ten true instances. If a model predicts everything as false, it will have an accuracy score of 0.90, not representing how the model performs. However, in such a scenario, the F_1-score will be 0.66, which represents the model's performance more precisely. As all three of our datasets show a little class imbalance (roughly 60:40), F_1-score helps to gain more trust in the model.

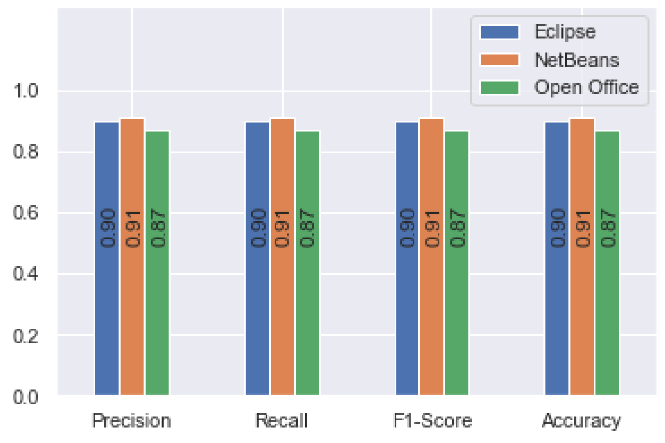

Fig. 4. Precision, Recall, F_1-score, and Accuracy for three datasets

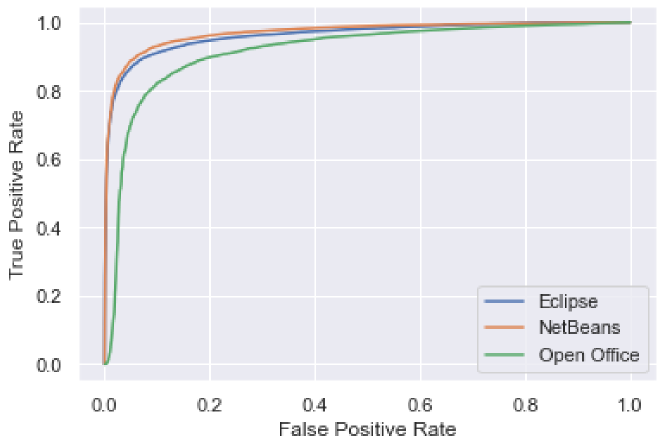

Fig. 5. Receiver Operating Characteristics Curve For The Datasets

5) Receiver Operating Characteristics: Receiver Operating Characteristics (ROC) is a graphical representation of how well the model can separate true instances from false instances. Essentially, it plots the true-positive rate (TPR) and false-positive rate (FPR) for all instances. True-positive rate and false-positive rate are defined as

$$TPR = \frac{TP}{TP + FN} \tag{8}$$

$$FPR = \frac{FP}{FP + TN} \tag{9}$$

From equation 6 and 8, we can see that true positive rate and recall are essentially the same metric.

Suppose a receiver operating characteristics curve lies along the counter-diagonal. In that case, the model cannot differentiate true instances from false ones, and it is essentially making random guesses. The higher the curve stays from the counter-diagonal, the better the model can differentiate.

ROC curve can be expressed as a single metric value as well known as area under the receiver operating characteristics (AUROC). It is computed by integrating the ROC curve. If the ROC curve lies on the counter-diagonal, then the corresponding AUROC will be 0.5, which means the model cannot differentiate true instances from false instances. A value below 0.5 means the model essentially identifies true instances as false and vice versa. The higher the AUROC value is, the better the model.

VI. Result

In this section, we describe how we answer our two research questions based on the outcome of our experiment.

A. Answering RQ_1

Figure 4 shows the precision, recall, f_1-score, and accuracy for the three different datasets. From there, we see that our siamese cross-encoder network performs quite consistently among datasets. Moreover, it is not biased toward positive or negative instances, and as a result, all metrics exhibit the same score.

Figure 5 shows the receiver operating characteristics curve for the datasets. The curves show that the network can differentiate true and false instances adequately. The AUROC scores for ‘Eclipse’, ‘NetBeans’, and ‘Open Office’ datasets are 0.96, 0.96, and 0.92, respectively. These scores also prove that our siamese network has an exquisite capability of separating true instances from false instances.

Figure 4 and 5 also show that our model can predict the ‘NetBeans’ the best and ‘Open Office’ dataset relatively lower. To understand if the differences among the metrics for different datasets are statistically significant, we perform two-sided t-tests. For the three datasets, we take the prediction of instances being duplicate. Then we compute unpaired t-test on the predicted values.

TABLE IV
P-values For Unpaired T-test

Dataset 1	**Dataset 2**	***p-value***
Eclipse	Open Office	1.95e-21
Eclipse	NetBeans	5.69e-4
Open Office	NetBeans	5.83e-39

We hypothesize that any dataset pair has identical expected values. We set the significance level to 0.001. Table IV shows the p-values from the t-test of all dataset pairs. However, from the table, we see that for no pair, we have a significant p-value to accept the null hypotheses. Therefore, we reject the null hypotheses and accept the alternative hypotheses that – there are significant differences in the performances among datasets.

Therefore, to answer RQ_1, a siamese cross-encoder network is an excellent choice for identifying duplicate bug reports. It has an exquisite capability to distinguish duplicate pairs from non-duplicate pairs with adequate accuracy.

B. Answering RQ$_2$

The primary difference between our siamese network and the siamese network by Deshmukh et al. [1] is that our network does not handle structured information separately. Instead, we pass the structured information along with titles and descriptions to the interconnected network. Therefore, we can surmise that the difference between the performances of the two networks is the utility of structured information in identifying duplicate bugs.

TABLE V
ACCURACIES FOR DATASETS IN OUR NETWORK AND DESHMUKH'S NETWORK

Dataset	Our Network	Deshmukh's Network
Eclipse	0.90	0.90
NetBeans	0.91	0.91
Open Office	0.87	0.94

Table V shows the accuracies for different datasets in our siamese network and Deshmukh's siamese network. From the table we see for 'Eclipse' and 'NetBeans' datasets, there is no difference in accuracies. However, in the case of 'Open Office', there is a significant difference between accuracies. As 'Open Office' is the worst performing dataset in our network and the best performing dataset in Deshmukh's network, handling structured data separately plays a major role in this dataset.

Therefore, to answer RQ$_2$, the impact of handling structured data separately depends on the bug repository itself. In most cases, it does not have a significant impact. However, if the repository is maintained accordingly (e.g. 'Open Office'), it can play a major role in identifying duplicate bug reports.

VII. CONCLUSION AND FUTURE WORK

Duplicate Bug Report Detection is a time-consuming and laborious job in software development, but it is also a vital job that must be addressed regularly. As a result, automating the duplication detection method for usage in the software business is essential. Given that, we proposed a deep learning-based approach that automates the detection of duplicate bug reports. Specifically, we employed a Bi-LSTM and CNN based approach that has been evaluated on three publicly available datasets. Our proposed model obtained an accuracy of more or equal to 90% for the datasets except for the *Open Office*, which supports the previous finding of Deshmukh et al. [1]. Furthermore, we have also explored the impact of structured information on duplicate bug report detection. In general, the structured information in bug reports does not significantly influence detecting duplicate bugs. Nevertheless, maintaining the bug repository accordingly, the structured information can play a significant role in duplicate bug detection.

Although our model obtained a exquisite result for *Eclipse* and *NetBeans*, the result for *Open Office* still has room for improvement. Therefore, in the future, we aim to investigate further how we can achieve similar performance for *Open Office* as well. Furthermore, the comparison between our study and Deshmukh et al. [1] showed that structured information could contribute to identifying duplicate bug reports if the repository is maintained properly. Thus, we will investigate the differences between how the bug repository for *Open Office* is maintained with respect to the other subject systems. We believe that from this investigation, we can come up with influential suggestions for the software industry on how they can more effectively maintain their bug repositories.

REFERENCES

[1] J. Deshmukh, K. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 115–124.

[2] Y. C. Cavalcanti, E. S. de Almeida, C. E. A. da Cunha, D. Lucrédio, and S. R. de Lemos Meira, "An initial study on the bug report duplication problem," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 264–267.

[3] S. Gupta and S. K. Gupta, "A systematic study of duplicate bug report detection."

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] L. Kang, "Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB," 2017. [Online]. Available: http://www.diva-portal.se/smash/get/diva2:1153748/FULLTEXT02.pdf

[6] Y. C. Cavalcanti, P. A. d. M. S. Neto, D. Lucrédio, T. Vale, E. S. de Almeida, and S. R. de Lemos Meira, "The bug report duplication problem: an exploratory study," *Software Quality Journal*, vol. 21, no. 1, pp. 39–66, 2013.

[7] Q. Guo, W. Feng, C. Zhou, R. Huang, L. Wan, and S. Wang, "Learning dynamic siamese network for visual object tracking," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[8] A. Lazar, S. Ritchey, and B. Sharif, "Generating duplicate bug datasets," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 392–395.

[9] S. Gupta and S. K. Gupta, "A systematic study of duplicate bug report detection," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 12, no. 1, 2021.

[10] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 499–510.

[11] 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 *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 461–470.

[12] C.-Z. Yang, H.-H. Du, S.-S. Wu, and X. Chen, "Duplication detection for software bug reports based on bm25 term weighting," in *2012 Conference on Technologies and Applications of Artificial Intelligence*. IEEE, 2012, pp. 33–38.

[13] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, "Detecting duplicate bug reports with software engineering domain knowledge," *Journal of Software: Evolution and Process*, vol. 29, no. 3, p. e1821, 2017.

[14] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 183–192.

[15] R. P. Gopalan and A. Krishna, "Duplicate bug report detection using clustering," in *2014 23rd Australian Software Engineering Conference*. IEEE, 2014, pp. 104–109.

[16] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *2012 16th European conference on software maintenance and reengineering*. IEEE, 2012, pp. 385–390.

[17] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 45–54.

[18] N. Klein, C. S. Corley, and N. A. Kraft, "New features for duplicate bug detection," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 324–327. [Online]. Available: https://doi.org/10.1145/2597073.2597090

[19] T. M. Rocha and A. L. D. C. Carvalho, "Siameseqat: A semantic context-based duplicate bug report detection using replicated cluster information," *IEEE Access*, vol. 9, pp. 44 610–44 630, 2021.

[20] H. T. Sueno, B. D. Gerardo, and R. P. Medina, "Converting text to numerical representation using modified bayesian vectorization technique for multi-class classification," *International Journal*, vol. 9, no. 4, 2020.

[21] A. Jha, "Vectorization techniques in nlp [guide] - neptune.ai," 2021. [Online]. Available: https://neptune.ai/blog/vectorization-techniques-in-nlp-guide

[22] S. Lai, K. Liu, S. He, and J. Zhao, "How to generate a good word embedding," *IEEE Intelligent Systems*, vol. 31, no. 6, pp. 5–14, 2016.

[23] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[24] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[25] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*. PMLR, 2013, pp. 1310–1318.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] J. Brownlee, "How to develop a bidirectional lstm for sequence classification in python with keras," 2021. [Online]. Available: https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/

[28] B. D. Mwiti, "Nlp essential guide: Convolutional neural network for sentence classification — cnvrg.io," 2021. [Online]. Available: https://cnvrg.io/cnn-sentence-classification/

[29] B. Hu, Z. Lu, H. Li, and Q. Chen, "Convolutional neural network architectures for matching natural language sentences," in *Advances in neural information processing systems*, 2014, pp. 2042–2050.

[30] S. Benhur J, "A friendly introduction to siamese networks," 2021. [Online]. Available: https://towardsdatascience.com/a-friendly-introduction-to-siamese-networks-85ab17522942

[31] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[34] A. Lazar, S. Ritchey, and B. Sharif, "Generating duplicate bug datasets," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 392–395. [Online]. Available: https://doi.org/10.1145/2597073.2597128

[35] T. Solc and S. Burke, "Unidecode-ascii transliterations of unicode text," 2019.