A Soft Alignment Model for Bug Deduplication

Irving Muller Rodrigues
Polytechnique Montreal
Montreal, Canada
irving.muller-rodrigues@polymtl.ca

Daniel Aloise
Polytechnique Montreal
Montreal, Canada
daniel.aloise@polymtl.ca

Eraldo Rezende Fernandes
FACOM – UFMS
Campo Grande, Brazil
eraldo@facom.ufms.br

Michel Dagenais
Polytechnique Montreal
Montreal, Canada
michel.dagenais@polymtl.ca

ABSTRACT

Bug tracking systems (BTS) are widely used in software projects. An important task in such systems consists of identifying duplicate bug reports, i.e., distinct reports related to the same software issue. For several reasons, reporting bugs that have already been reported is quite frequent, making their manual triage impractical in large BTSs. In this paper, we present a novel deep learning network based on soft-attention alignment to improve duplicate bug report detection. For a given pair of possibly duplicate reports, the attention mechanism computes interdependent representations for each report, which is more powerful than previous approaches. We evaluate our model on four well-known datasets derived from BTSs of four popular open-source projects. Our evaluation is based on a ranking-based metric, which is more realistic than decision-making metrics used in many previous works. Achieved results demonstrate that our model outperforms state-of-the-art systems and strong baselines in different scenarios. Finally, an ablation study is performed to confirm that the proposed architecture improves the duplicate bug reports detection.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; • **Software and its engineering** → *Maintaining software.*

KEYWORDS

Bug Tracking Systems, Duplicate Bug Report Detection, Deep Learning, Attention Mechanism

ACM Reference Format:
Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. 2020. A Soft Alignment Model for Bug Deduplication. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387470

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387470

1 INTRODUCTION

Bug fixing accounts for a substantial part of any software development project. Thus, many projects make use of a *bug tracking system* (BTS) to manage and track bug reports. One important task in such systems is to identify duplicate bug reports, i.e., distinct reports describing issues caused by the same bug in the software. It is crucial to perform this task as fast as possible in order to prevent developers from spending time looking for bugs already fixed. Usually, a triage team manually labels new reports as duplicate or not [29]. However, especially in open source projects, bug reports can be submitted by developers, testers and even end users. This heterogeneous environment leads to many duplicate bug reports. For example, 12% of all reports are duplicate in one Eclipse instance [4]. Therefore, devising automatic methods to detect duplicate bug reports is crucial for efficient software maintenance. In the literature, such a task is called duplicate bug report detection, bug report deduplication or, simply, *bug deduplication.*

Typically, a bug report comprises a summary, a description, and some categorical fields (e.g., system, component and version). Regarding textual data, for simplicity, the terms *word* and *token* are considered interchangeable in this work. One core component of most methods in bug deduplication is a similarity function that compares a pair of reports. How this function is composed and used vary greatly from one method to another. A handful of studies [6, 10, 22, 37] employ deep neural networks in order to model similarity functions. Deshmukh et al. [10], Budhiraja et al. [6] and Xie et al. [37] works are based on Siamese neural networks [8] that generate the representation of one bug report without considering the other report content. This independent representation is limited specially for textual data, since it may focus on generic features that are not relevant for a specific comparison [32]. Poddar et al. [22] try to mitigate that shortcoming by employing an architecture that exchanges information between the reports during feature extraction. This approach generates a joint representation based on *attention* [1], in which the representation of a word in a report attends to a *pooled representation* of all words in the other report.

In this work, we propose a novel deep learning network that produces joint representations of reports based on a soft-attention alignment mechanism [20]. The key component of this model is a layer that compares each word in a report with a fixed-length representation of all words in the other one. While Poddar et al. [22] also use an attention mechanism, our proposed architecture is able to summarize relevant information within one report conditioned

to a specific segment of the other report. This provides a more powerful representation of textual data.

Many previous works on bug deduplication have employed an evaluation methodology called *decision-making approach* [17]. This evaluation is based on pairs of reports labeled as positive when they refer to the same bug or negative otherwise. Positive pairs comprise all possible pairs within a set of duplicate reports. While negative pairs are generated using some sampling technique. Model performance is then measured by means of ordinary classification metrics (like accuracy, precision and recall) over the *generated set* of positive and negative pairs. The decision-making approach is quite unrealistic, since the real scenario presents a much larger set of negative candidates. When a new report is submitted to a BTS, all previously submitted reports are duplicate candidates. Thus, this evaluation methodology highly overestimate performance. Another popular evaluation methodology is the ranking approach. It acknowledges that the current techniques are not accurate enough to automatically detect duplicate reports with no human intervention. Therefore, in this approach, for a given new report, the proposed methods generate a list of the K most likely duplicate reports. The triager then identify whether a report is duplicate considering only the reports from the recommendation list. Instead of searching and examining hundreds of reports in the BTS, the triager can focus on the K recommended reports.

We experimentally evaluate our model by means of a ranking methodology based on Sun et al. [30]. We report on experiments using four well-known datasets derived from BTSs of open-source projects, namely Eclipse, Mozilla, NetBeans and OpenOffice. Bug deduplication in open-source projects is particularly challenging because any user can submit a bug report in their BTS and the knowledge of these users about the system may vary significantly. State-of-the-art systems and some strong baselines are compared to the proposed model in several scenarios. Additionally, we perform an ablation study to assess different aspects of our model.

The main contributions of this paper are summarized as follows:

(1) We propose a soft-alignment model that is based on a more powerful architecture than previous methods.
(2) Our method and the baselines are evaluated using a more realistic methodology. This work is the first to compare different deep learning methods using the ranking approach.
(3) Our method achieves state-of-the-art performance on all considered datasets.

2 RELATED WORK

Several non-deep learning methods in the literature address the bug deduplication as a ranking problem. Runeson et al. [26] are the first to use NLP techniques to approach duplicate bug report detection. They measure report similarity by computing the cosine similarity between bag-of-words vectors. Wang et al. [36] detected duplicate reports by combining function calls during the system execution with textual data. Sun et al. [30] trained an SVM to estimate the probability of reports being duplicate by receiving 54 textual similarity features generated from different combinations of text origins, n-gram lengths and dictionaries. Sureka and Jalote [31] proposed a similarity function whose output is proportional to the quantity of n-gram of characters in common between two reports. Sun et al. [29] proposed $BM25F_{ext}$ and REP for bug deduplication. $BM25F_{ext}$ is an extension of BM25F specially designed to address scenarios in which queries are long sentences with few or no duplicate words. REP is a similarity function that linearly combines $BM25F_{ext}$ scores using unigram and bigram with features generated from categorical data comparisons. Prifti et al. [23] developed a method to rank reports using a time window and a unique representation for each master group. Nguyen et al. [19] proposed a method, called *DBTM*, that linearly combines the BM25F score and the topic similarity computed by a model based on Latent Dirichlet Allocation (LDA). Banerjee et al. [2] addressed the bug report deduplication by using the longest common subsequence between the bug reports. Zhou and Zhang [40] proposed a linear model, called *BugSim*, which is trained to minimize the fidelity loss of triplets using features inspired by Nallapati [18]. In Yang et al. [38], BM25 is used to weight the bag-of-words vectors which are compared by the cosine similarity. Banerjee et al. [3] generated a top-20 list for different similarity measures and aggregated them into a unique list using two fusion approaches: one retrieves the maximum score of a report in the lists while the other sums the similarity scores of the reports. Lerch and Mezini [15] proposed to use the stack trace in the bug report to better detect duplicate bug reports. Sabor et al. [27] improved Lerch and Mezini [15] by employing only packages names instead of full method names. Lin and Yang [16] combined TF-IDF with a weighting scheme based on the term relations within the clusters of reports. Lin et al. [17] trained an SVM to estimate the duplication probability using cluster-based correlation features, the BM25F score and the cosine similarity of word vectors. Yang et al. [39] designed similarity function whose output depends on product and component differences, the cosine similarity of TF-IDF vectors and the average of word embeddings. Budhiraja et al. [7] proposed LWE which combines LDA with the word embeddings.

Regarding deep learning methods, Xie et al. [37] proposed a convolutional neural network (CNN), called DBR-CNN, to classify pairs of duplicate bug reports. In their architecture, a shared CNN independently encodes the textual data of the pair of reports into two vectors. A logistic regression then classifies each pair of reports by receiving the cosine similarity of those vectors and a set of features related to categorical data. In NLP, statistical methods parse textual data from documents to discover latent themes, called topics, which are common between multiple documents [5] (e.g., bug reports that contain the words combo or font can be related to the topic *UI*). Poddar et al. [22] proposed a neural network that simultaneously learns to cluster reports based on topics while detecting duplicate pairs. A recurrent neural network (RNN) represents each word of a report as a vector. The k-first dimensions of these vectors are trained to yield high similarities to words that are in the same topic. For the classification, Poddar et al. [22] generate the joint-representation of a report as a weighted average of its word vectors. An attention mechanism calculates these weights by using the self-attention coefficients of the topic information and the element-wise multiplication of the word representations in the report with the mean pooling of all words in the other report. The authors used only summary data from the reports in their experiments.

Budhiraja et al. [6] proposed a neural network, called DWEN, in which the fixed-length representation of a report is the mean of their word vectors and the classifier is a multi-layer neural network

(MLNN) that receives only the representation of a pair of reports. Deshmukh et al. [10] proposed two siamese neural networks for bug deduplication. The authors used a feed-forward neural network, a CNN, and a bidirectional LSTM to encode, respectively, the categorical data, the description, and the summary into vectors. The concatenation of these encoder outputs generates the fixed-length representation of the reports. Deshmukh et al. [10] proposed two approaches to calculate the similarity of the report representations. The first one, called Siamese Triplet, is trained to minimize a hinge loss given a set of *triplets* and employs the cosine function to compute the similarity between two reports. The second one, called Siamese Pair, uses the binary cross-entropy loss of pair in the training and scores the similarity between reports using a MLNN.

This paper presents a method that improves the representation generation found in the previous deep learning approaches. Different from Budhiraja et al. [6], Deshmukh et al. [10], and Xie et al. [37], our model exchanges information between the reports before encoding textual data into a fixed-length vector. Moreover, in the feature extraction, our method can dynamically focus on different segments of a report instead of providing a unique set of features from it, as done by Poddar et al. [22]. This more powerful architecture can reduce information loss in the representation generation thereby improving the duplicate bug report detection.

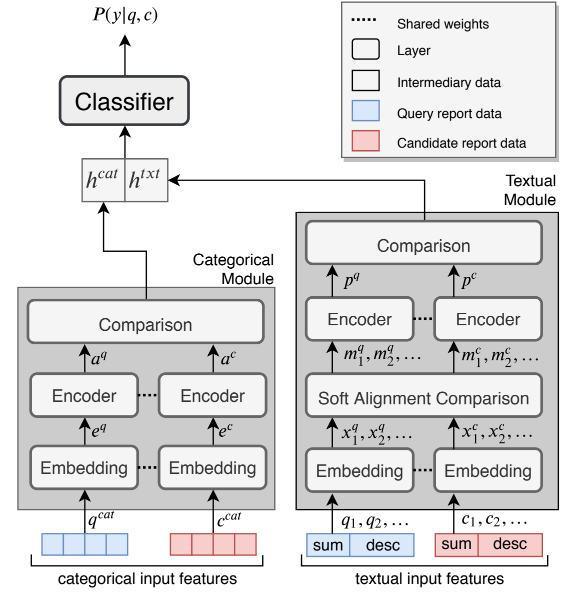

Figure 1: SABD Architecture Overview.

3 SOFT ALIGNMENT MODEL FOR BUG DEDUPLICATION

In this section, we describe our proposed *Soft Alignment Model for Bug Deduplication* (SABD). This model receives a pair of bug reports: a new query report q and a candidate report c previously submitted to the repository. The model outputs the probability $P(y|q,c)$ of q being a duplicate of c, where y indicates whether the given reports are duplicate ($y = 1$) or not ($y = 0$). We consider a bug report to be composed of the categorical fields, a summary and a description. Given a query report q, the values of its categorical fields are represented as the tuple q^{cat} while the sequence of words of its summary and description are denoted as q^s and q^d, respectively. The same notation is employed for the candidate c.

Figure 1 depicts the SABD architecture. As we can see, SABD is composed of the categorical and textual modules (two sub-networks) that independently compare the categorical and textual data from both reports, respectively. The classifier receives these module outputs and produces the final prediction $P(y|q,c)$. While the categorical module is a straightforward dense neural network, a more sophisticated architecture is employed by the textual module to handle text. The core of this module is the *soft alignment comparison layer* that allows the model to dynamically access distinct information from the text. This mechanism is expected to improve the model capacity to focus on relevant features in the textual data for the deduplication. In the remainder, we describe the details of SABD and its modules.

3.1 Categorical Module

The categorical module is composed of three layers: embedding, encoder, and comparison. In the embedding layer, each categorical field is related to a parameterized lookup table that links the field value to a real-valued vector. This representation is more powerful than using binary variables (e.g., feature is 1 if and only if field values are equal) since it allows the model to group similar field values. Given the query q, the embedding layer concatenates the real-valued vectors of each categorical value in q and outputs $e^q \in \mathbb{R}^{cl \cdot d^{cat}}$, where cl is the number of categorical fields in the report and d^{cat} is a hyperparameter indicating the categorical vector dimensions. The encoder layer receives the embedding layer output e^q and generates a fixed-representation a^q of the categorical data from q such that:

$$a^q = ReLU(W^a e^q + b^a),$$

where $W^a \in \mathbb{R}^{d^a \times (cl \cdot d^{cat})}$ is the weight matrix parameter, $b^a \in \mathbb{R}^{d^a}$ is the bias parameter, and d^a is a hyperparameter that controls the layer size. Analogously, the fixed representation a^c is produced for the categorical data of candidate c.

After encoding the categorical features into vectors a^q and a^c, the categorical comparison layer computes a comparative representation of these two vectors by a simple operation given by:

$$\text{cmp}(a^q, a^c) = [(a^q - a^c)^2; a^q \odot a^c], \tag{1}$$

where $[\,\cdot\,; \ldots; \,\cdot\,]$ is the concatenation operator and $\odot$ represents element-wise multiplication. Finally, given the comparative representation, a fully connected (FC) layer computes the comparison layer output as:

$$h^{cat} = ReLU(W^h[a^q; a^c; \text{cmp}(x^{cq}, x^{cc})] + b^h),$$

where $W^h \in \mathbb{R}^{d^h \times 4d^a}$ is the FC weight matrix, $b^h \in \mathbb{R}^{d^h}$ is the FC bias vector, and d^h is a hyperparameter.

3.2 Textual Module

Although categorical features are relevant to solving bug deduplication, the most informative features are the summary and description

texts. Thus, the core of our model is the textual module that compares the textual features of the query and candidate reports (i.e., q^s, q^d, c^s and c^d). It comprises four layers: *textual embedding*, *soft alignment comparison*, *textual encoder*, and *textual comparison*.

3.2.1 Textual Embedding Layer. This layer independently transforms the words from the query and candidate texts into real-vectors (word embeddings). A pre-trained look-up table in this layer links each word in the summary and description of a report to an embedding. This word representation loses information about the word origin since the same look-up table is used for both textual fields. Previous works [10, 29] present evidences that they are both important for bug deduplication. Moreover, it is important to distinguish summary and description words since each field presents unique characteristics. Consequently, two distinct real-vectors are employed to distinguish whether a word comes from the summary or description. Such representation is denominated as *field embedding*. Given a query q, the summary q^s and description q^d are concatenated into a single sequence q^t whose i-th word is represented as: $v_i^q = [w_i^q; f_i^q] \in \mathbb{R}^{d^w + d^f}$, where w_i^q is the word embedding, f_i^q is the field embedding, and d^w and d^f are hyperparameters indicating their respective vector dimensions.

Although the field embeddings are learned in the learning phase, word embedding vectors are not fine-tuned during the training because it increases computation cost, limits vocabulary size and can lead to overfitting [33]. Instead of updating word embedding parameters, we provide each word representation v_i^q to a fully connected layer (FC) along with residual connections:

$$x_i^q = v_i^q + \mathrm{ReLU}(W^x v_i^q + b^x), \tag{2}$$

where $W^x \in \mathbb{R}^{d^x \times d^x}$ is the FC weight matrix, $b^x \in \mathbb{R}^{d^x}$ is the FC bias vector, and $d^x = d^f + d^w$. This solution not only reduces computation cost, memory usage, and model complexity but also allows the model to project words into a more meaningful feature space. In the end, textual embedding layer receives q^s and q^d and outputs a sequence of embedding vectors $x^q = (x_1^q, x_2^q, \ldots, x_{|q^t|}^q)$, where $|q^t|$ is the length of q^t.

Similarly, for the candidate report c, a sequence of embedding vectors $x^c = (x_1^c, x_2^c, \ldots, x_{|c^t|}^c)$ are provided by an identical embedding layer, where c^t is the concatenation of summary and description words in the candidate report and $|c^t|$ is the length of c^t. Again, as indicated in Figure 1, query and candidate textual embedding layers share their parameters.

3.2.2 Soft Alignment Comparison Layer. Previous deep learning methods for bug deduplication [6, 10, 22, 37] encode query and candidate reports without or with limited data exchange. SABD overcomes this limitation with an architecture that provides a more powerful feature interaction. The core of this layer is the soft-attention alignment [20]. This attention mechanism computes a similarity score s_{ij} between query token q_i^t and candidate token c_j^t as such:

$$s_{ij} = \frac{\left(x_i^q\right)^T \cdot x_j^c}{\sqrt{d^x}}.$$

The previous equation is known as the scaled dot-product [34].

In order to accentuate important features of the words contained in a report, the soft alignment comparison layer must have access to textual information from the other report. However, texts are variable-length data and can contain a large set of potential relevant features. Thus, this layer uses the similarity score to select features from the report words that are related to a specific word in the other report. These features are encoded into a fixed-length representation. More precisely, each word vector x_i^q in the query attends to all candidate vectors $x_1^c, x_2^c, \ldots, x_{|c^t|}^c$, in order to produce a fixed-length representation:

$$\overline{x}_i^q = \sum_{j=1}^{|c^t|} \alpha_j^{q_i} x_j^c, \tag{3}$$

where $\alpha_j^{q_i} = exp(s_{ij}) / \sum_{k=1}^{|c^t|} exp(s_{ik})$ is called *attention score* and represents the normalized similarity score. $\overline{x}_i^q$ is denominated as *query contextual vector* and is a weighted average of all word vectors from the candidate. The most similar words in the candidate to a word q_i^t have the largest impact in the query contextual vector. Analogously, each candidate token vector x_j^t attends to all query token vectors in order to produce a candidate context vector:

$$\overline{x}_j^c = \sum_{i=1}^{|q^t|} \alpha_i^{c_j} x_i^q,$$

where $\alpha_i^{c_j} = exp(s_{ij}) / \sum_{k=1}^{|q^t|} exp(s_{kj})$.

Finally, inspired by Wang and Jiang [35], each token vector of the query and candidate reports is compared with its corresponding context vector by means of the comparison function defined in Equation 1. Then, a fully-connected layer with a residual connection receives the resulting comparison vector and modifies the word vectors as follows:

$$m_i^q = x_i^q + \mathrm{ReLU}(W^m \mathrm{cmp}(\overline{x}_i^q, x_i^q) + b^m), \tag{4}$$

$$m_j^c = x_j^c + \mathrm{ReLU}(W^m \mathrm{cmp}(\overline{x}_j^c, x_j^c) + b^m), \tag{5}$$

where $\mathrm{cmp}(\cdot,\cdot)$ is defined in Equation 1, $W^m \in \mathbb{R}^{d^x \times 2d^x}$ is the layer weight matrix, and $b^m \in \mathbb{R}^{d^x}$ is the layer bias.

3.2.3 Textual Encoder Layer. This layer takes the variable-size representation of a report text (query or candidate) and produces a fixed-size representation. This operation is independently performed for the query and candidate reports.

Considering a query q, a bi-directional long short-term memory (bi-LSTM) processes the soft alignment comparison output m^q :

$$\overrightarrow{o}_i^q = \overrightarrow{\mathrm{LSTM}}(m_i^q, \overrightarrow{o}_{i-1}^q)$$

$$\overleftarrow{o}_i^q = \overleftarrow{\mathrm{LSTM}}(m_i^q, \overleftarrow{o}_{i+1}^q)$$

for $i = 1, 2, \ldots, |q^t|$. The vectors $\overrightarrow{o}_i^q \in \mathbb{R}^{d^o}$ and $\overleftarrow{o}_i^q \in \mathbb{R}^{d^o}$ are concatenated into $o_i^q \in \mathbb{R}^{2d^o}$, where d^o is a hyperparameter indicating the hidden size of the forward $\overrightarrow{LSTM}$ and backward $\overleftarrow{LSTM}$. The intuition is that the bi-LSTM enriches the previous representation with contextual information and allows to capture long dependencies between the words. For the sake of brevity, we omit the technical details of bi-LSTM since it is a standard neural building block. For a detailed explanation of the model, we refer the reader to Goodfellow et al. [12].

Finally, the fixed-representation of the query text is generated as follows:

$$p^q = \text{Pooling}(o_1^q, o_2^q, \ldots, o_{|q^t|}^q),$$

where $p^q \in \mathbb{R}^{4d^o}$, and Pooling is a function that concatenates the results of the mean and max pooling operators. The first operator calculates the average vector of the sequence o^q while the second performs the max operation through each dimension of the bi-LSTM output. Similarly, the textual encoder layer generates the fixed representation of the candidate text, denoted as p^c. As depicted in Figure 1, query and candidate textual encoder layers share their parameters.

3.2.4 Textual Comparison Layer. This layer compares the textual representations of both reports. As the categorical comparison layer, the $\text{cmp}(\cdot, \cdot)$ function (Equation 1) is used to generate a comparative representation. In the sequel, a fully connected layer generates the actual textual comparison:

$$h^{txt} = \text{ReLU}(W^u[p^q; p^c; \text{cmp}(p^q, p^c)] + b^u),$$

where $W^u \in \mathbb{R}^{d^u \times 16d^o}$ is the FC weight matrix, $b^u \in \mathbb{R}^{d^u}$ is the FC bias vector, and d^u is a hyperparameter.

3.3 Classifier

The SABD output layer comprises two sub-layers: a fully-connected layer and a classification layer. The input of the FC layer is the concatenation of two vectors: the categorical comparison output h^{cat} and the query representation h^{txt}. The classification layer is a standard logistic regression. Thus, the output layer is given by:

$$P(y|c, q) = \text{sigmoid}(W^s \text{ReLU}(W^r x + b^r) + b^s),$$

where $x = [h^{cat}; h^{txt}] \in \mathbb{R}^{(d^h + d^u)}$ is the input described above; $W^r \in \mathbb{R}^{d^r \times (d^h + d^u)}$, $b^r \in \mathbb{R}^{d^r}$, $W^s \in \mathbb{R}^{1 \times d^r}$, $b^s \in \mathbb{R}$ are parameters; and d^r is a hyperparameter.

4 EXPERIMENTAL SETUP

In this section, we describe the main steps of our experimental setup: the evaluation methodology, training procedure, used datasets, and competing methods. The data used in this work and the developed code are freely available [1].

4.1 Evaluation Methodology

Towards a more realistic evaluation setup than those used by previous deep learning methods, we evaluate our models using a ranking-based methodology similar to Sun et al. [29]. First, we sort the bug reports in a BTS by their creation date. Then, the reports are chronologically read and inserted in the training set until a specific date t. All the subsequent reports are used to create the test set. Finally, we group the reports in the training set that describe the same bug into buckets. In each bucket, the first submitted report is considered the master report and the remaining ones are the duplicate reports.

In Table 1, we exemplify a BTS with five bug reports. Considering that t is 21/12/2018, we generate a training set composed of R1, R2, and R3 and a test set consisting of R4 and R5. The training set thus comprises two buckets: $B_1 = \{\text{R1}, \text{R3}\}$ and $B_2 = \{\text{R2}\}$. During evaluation, we chronologically pick each report r in the test set.

[1] https://github.com/irving-muller/soft_alignment_model_bug_deduplication

Table 1: Fictional BTS to Exemplify the Evaluation Methodology.

Bug report ID	Creation Date	Master Report
R1	01/12/2018	-
R2	12/12/2018	-
R3	20/12/2018	R1
R4	30/12/2018	-
R5	31/12/2018	R1

When r is a duplicate bug report, we generate a ranked list of the buckets in the system. In this work, the score of a bucket B_i is the highest score yielded by a method when it compares r with each report in B_i. After checking whether r is duplicate, we consider it as submitted and insert r into its correct bucket. Following this procedure, for example, we first pick the report R4 in the scenario of Table 1. A ranked list is not produced because R4 is not duplicate and a new bucket $B_3 = \{\text{R4}\}$ is created. After that, the next report R5 is selected. Since it is duplicate, we generate a ranked list composed of $B1$, $B2$, and $B3$. Then, R5 is inserted in $B1$.

Regarding the evaluation methodology used by other ranking approach methods, Budhiraja et al. [6] do not describe how the test dataset was generated nor the procedure to create the ranked list. Both are crucial elements of the evaluation methodology and can considerably impact the achieved performances. Deshmukh et al. [10] extract pairs of bug reports from a BTS and *randomly* split them into training and test datasets. In their evaluation, for each duplicate bug report, their method outputs a recommendation list composed of only reports within the test set. This artificially reduces the number of reports that must be searched for each queried report, which makes the problem much easier [4]. Furthermore, in the BTSs, we only have access to data that was reported before a current time x. Thus, the model can only be trained using data from this period. After training a model, it only examines bug reports that were created after x. Randomly shuffling the data allows the model to be trained with reports created from the future (after x) and to retrieve candidates that were submitted after the queried report. Additionally, this randomization makes the problem easier because it spreads more uniformly the features through the dataset and can mitigate the concept drift issue. Therefore, we believe our experimental setting is more realistic. We compare our methods with those proposed in Budhiraja et al. [6] and Deshmukh et al. [10]. However, due to the methodological differences aforementioned, and since source code was not provided by authors, we implemented those methods to the best of our knowledge, as described in Section 4.5.

Like Sun et al. [29], we evaluate a method using two metrics: mean average precision (MAP) and recall rate@k (RR@k). Both metrics are based on the ranking of reports according to the scores computed by a method. MAP is a general ranking metric. In our setting, rankings only need to contain one relevant item per query to be considered a hit. Thus, MAP can be simplified as:

$$MAP = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{p_i},$$

where Q is the number of duplicate bug reports in the evaluation set and p_i is the position of the correct bucket in the ranked list. RR@k is equal to the ratio of duplicate reports whose correct buckets are within the top-k buckets in the given ranking to the number of duplicate bug reports. RR@k is defined as:

$$\text{RR@}k = \frac{n_k}{Q},$$

where n_k is the number of query reports in the test set for which the corresponding bucket appears in the top-k positions of the ranking computed by a method.

4.2 Datasets

We use parts of the datasets published by Lazar et al. [14] in our experiments[2]. They retrieved and curated reports submitted until 2014 from four BTSs: OpenOffice, Eclipse, NetBeans and Mozilla. OpenOffice contains a set of open-source tools that aim to help the office activities. NetBeans and Eclipse are popular open-source integrated development environments (IDEs) that support many different languages. The Mozilla BTS manages bugs of several open source projects, such as Thunderbird (email client) and Firefox (Web browser).

Sun et al. [29] assess their methods using small portions of the aforementioned datasets. More specifically, they use reports within a three-year period for the OpenOffice dataset and within a one-year period for the other three datasets. This choice ignores reports submitted before this period, which overestimates their method [24]. For each BTS, we use the reports employed by Sun et al. [29] as our test datasets[3]. The reports submitted before these periods are split into training and validation datasets. Validation sets comprise the latest 5% reports and the remaining earlier reports comprise training sets. Statistics of these datasets are presented in Table 2.

4.3 Time Window

As the number of reports submitted increases over time, it becomes computationally expensive to detect duplicate bug reports in BTSs since each new report has to be compared with all the reports submitted before it. This growth indeed degrades the performance of the method, which can negatively affect the triage process [4]. A simple solution for this problem, called time window or time frame, consists of searching for reports that were submitted within a specific range of days before the new report. In this study, a bucket is considered to be a candidate when at least one of its reports is within the defined time window.

Table 3 shows the fraction of duplicate bug reports in the test sets for which one of the reports in their associated buckets can be reached within time windows of one and three years. We consider that three years is a reasonable time frame to be used in real environments, especially for popular software BTSs that daily receive many bug reports, e.g., Eclipse BTS receives on average around 99 reports per day [9]. Except for OpenOffice, the use of the time window significantly reduces the computation demand at the cost of a small negative impact on the performance upper bound – less than 3.4% of duplicate reports will not have their bucket in the ranked list. We also test the methods with a time window of one year to measure how its size affects performance.

[2]http://alazar.people.ysu.edu/msr14data/

[3]We decided to use the same period of Eclipse for Netbeans since [29] did not evaluate their method on this BTS.

4.4 Training

We preprocess the textual data by replacing all the non-alphanumeric characters with spaces [29]. After that, the text is converted to lowercase and tokenized on white space characters. This preprocessing separates tokens concatenated by punctuation, e.g., module paths, file paths, and function calls. Our intuition is that package, file, and class names are relevant for this task. Analyzing a small sample of long reports, we observed that many of them append lengthy stack traces and log files to their description. Thus, we limit the text length to 350 tokens in order to clean less important elements without missing much relevant information. Although we acknowledge that this value can be suboptimal, it appears to be sufficient to achieve reasonable results. Categorical features comprise the following fields: component, product, severity and priority.

Following Deshmukh et al. [10], we initialize the word embedding using an instance of pre-trained vectors[4]. Words that appear in the training dataset but not in that instance are pre-trained using *GloVe* [21] and textual data from the reports in the training dataset. To avoid overfitting, the word embeddings are not fine-tuned.

SABD is a binary classifier that takes two reports (query q and candidate c) and outputs the probability $P(y|q,c)$ of report q being a duplicate of report c. Thus, it is trained over a set of pairs of reports along with their labels in order to minimize the cross entropy loss function:

$$J(\theta) = -\frac{1}{|S|} \sum_{(q,c,y) \in S} y \log P(y|q,c) + (1-y) \log(1 - P(y|q,c)),$$

where $S = \{(q, c, y)\}$ is the training set composed of pairs of reports (q, c) along with their labels y ($y = 1$ when q and c are duplicates, otherwise $y = 0$). We optimize SABD for 12 epochs using ADAM optimizer [13] with a learning rate of 0.001 and a batch size of 256.

Building S is challenging since it is used to train a binary classifier to perform a ranking task such that, at test time, there are more negative examples than positive ones. In order to describe how S is built, let us split it into two sets $S = S^+ \bigcup S^-$ such that S^+ is the set of positive examples, i.e, those whose $y = 1$; and S^- is the set of negative examples. S^+ comprises all pairs of duplicate reports in the set of training reports. On the other hand, S^- is generated before the start of each training epoch by sampling non-duplicate pairs until $\frac{|S_+|}{|S_-|} = rt$, where rt is the rate between the pairs of duplicate reports by the non-duplicate ones. Moreover, a negative pair is only included in S^- if $-\log(P(y = 0))$ is larger than a given threshold λ, i.e., if the example is not too easy for the current classifier. This sampling is inspired by [11, 25, 28, 32] and speeds up training by providing more informative examples. SABD has achieved optimum results for $rt = 1$ and $\lambda = 10^{-3}$. The hyperparemeter values were tuned using the validation set and their values are presented as follows: $d^{cat} = 20$, $d^a = 40$, $d^h = 40$, $d^w = 300$, $d^f = 5$, $d^o = 150$, $d^u = 600$, and $d^r = 300$.

[4]http://nlp.stanford.edu/data/glove.42B.300d.zip

Table 2: Statistics of datasets. *Period* column indicates the period comprising each dataset as a whole (Train+Val+Test). *Start Date* column indicates the first day included in test datasets.

Dataset	Period	Training		Validation		Test			Total
		Duplicate	All	Duplicate	All	Start Date	Duplicate	All	
Eclipse	10/10/01 - 31/12/08	27,481	198,183	1,446	14,703	01/01/08	4,380	45,794	258,680
Mozilla	07/04/98 - 31/12/10	122,199	438,806	6,431	44,014	01/01/10	9,701	65,940	548,760
OpenOffice	16/10/00 - 31/12/10	13,570	80,786	714	4,109	01/01/08	4,664	31,333	116,228
Netbeans	21/08/98 - 31/12/08	16,639	116,351	875	5,548	01/01/08	5,009	31,667	153,566

Table 3: Percentage of Duplicate Bug Reports in the Test Set that Reaches the Correct Bucket in the Time Window of One Year and Three Years.

Dataset	1 year	3 years
Eclipse	88.53%	97.48%
OpenOffice	75.27%	90.97%
Netbeans	93.51%	98.88%
Mozilla	88.45%	96.73%

4.5 Competing Methods

We compare SABD with five other methods from the literature. The BM25F$_{ext}$ and REP are ranking-approach methods that were proposed by Sun et al. [29]. These are popular methods and their implementations are available.[5] Besides, we compare SABD with the following deep learning methods: DWEN [6], Siamese Pair [10] and Siamese Triplet [10]. Although these works have used ranking-based evaluation methodologies, as described in Section 4.1, such methodologies present relevant issues. Moreover, since their implementations are not available, we implemented them to the best of our knowledge. We found that the following minor modifications in the method architecture or training have improved their performance in the validation dataset: 1) the feed-forward neural network of DWEN receives categorical features generated in a similar way to Siamese Pair [29]; 2) bi-LSTMs followed by average and max poolings are used to encode the summary and description in Siamese Pair and Siamese Triplet; 3) the last sub-network of Siamese Pair receives, in addition to the original inputs, the squared difference and element-wise multiplication of the final report representations; and 4) we train these methods using the procedure described in Section 4.4 to generate negative examples.

Models evaluated by means of decision-making methodology cannot be fairly compared to those that employ ranking-based approaches, since the underlying problem differs. However, SABD is indirectly compared with Xie et al. [37], as this model is very similar to the Siamese Pair baseline. Both models independently generate the fixed-representation of report pairs using standard neural network blocks (e.g., CNN and LSTM) and exploit categorical data. Poddar et al. [22] propose a technique to simultaneously learn latent topics from reports and train a classifier for bug deduplication. This technique could be adapted to SABD with some minor changes due to its generality.

[5]https://chengniansun.bitbucket.io/projects/bug-report/fast-dbrd.tgz

5 EXPERIMENTAL RESULTS

Since the competing methods and SABD are stochatics, we perform *five distinct runs* for each experimental configuration[6]. We report in this section average performance in terms of MAP and RR@k, as well as standard deviations illustrated as *error bands* in figures and *inside brackets* in tables. Following Sun et al. [29], the RR@k is calculated for each $k = 1, 2, \ldots, 20$. It is important to notice that, when evaluating a model, we include duplicate reports whose buckets are not within the considered time window. These duplicate reports are considered misses for RR@k computation, and their terms $\frac{1}{p_i}$ in the MAP expression are 0.

5.1 Main Analysis

In the right column of Figure 2, we depict RR@k of all methods in the four datasets using a time window of three years. In all datasets, SABD constantly achieves the best RR@k among the compared methods. It outperforms the second best method by 3.06%–5.01% in Eclipse, 3.34%–6.35% in OpenOffice, 2.66%–6.64% in Netbeans, and 4.17%–5.19% in Mozilla. Table 4 reports the method results on the MAP metric in each test set using time window of one and three years. Considering the time window of three years, SABD also achieves higher MAP values than all methods in all datasets. The improvement of SABD over the second best method on the MAP metric is 3.5%, 4.3%, 3.9%, and 4.5% in Eclipse, OpenOffice, Netbeans, and Mozilla, respectively.

Deshmukh et al. [10] compared the Siamese Triplet with Siamese Pair using only accuracy and, according to them, the former significantly outperforms the latter. However, as mentioned before, the accuracy is less adherent than RR@k and MAP for real environments. We found that, in fact, Siamese Pair achieves significantly better RR@k and MAP values than Siamese Triplet in three of four test sets. Moreover, our results show that DWEN achieves poor MAP and RR@k values on the four datasets and it is significantly outperformed by all methods in Eclipse, Netbeans, and Mozilla repositories.

Considering only the deep learning models and the time window of three years, the improvement of SABD over the second best neural network on the RR@k metric is 6.78%–8.21%, 5.29%–6.91%, 5.97%–8.49% and 4.17%–5.19% in Eclipse, OpenOffice, Netbeans and Mozilla, respectively. In terms of MAP, SABD surpasses the second best neural network by 7.2% in Eclipse, 4.3% in OpenOffice, 6.8% in Netbeans and 4.5% in Mozilla. To the best our knowledge, we are the first to compare different neural networks in the bug

[6]BM25F$_{ext}$ and REP are run 10 times in NetBeans since they generated large standard deviation.

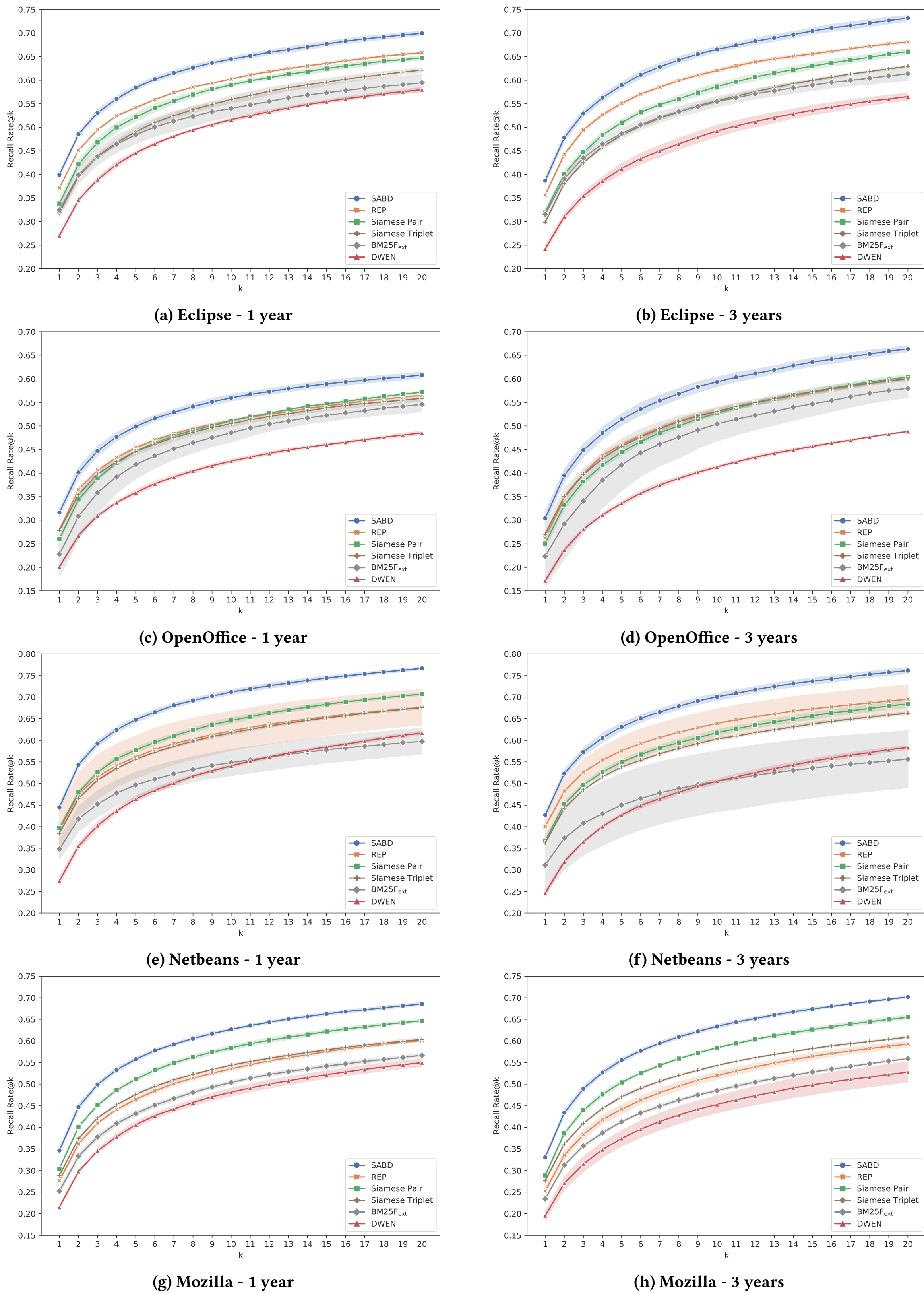

Figure 2: Recall Rate@k in Test Sets of Eclipse, OpenOffice, Netbeans and Mozilla.

Table 4: MAP in Test Sets.

Method	Eclipse		OpenOffice		Netbeans		Mozilla	
	1 year	3 years	1 year	3 years	1 year	3 years	1 year	3 years
DWEN	0.353[0.004]	0.325[0.007]	0.276[0.003]	0.252[0.004]	0.365[0.006]	0.333[0.004]	0.305[0.004]	0.282[0.012]
$BM25F_{ext}$	0.402[0.016]	0.398[0.010]	0.315[0.037]	0.313[0.054]	0.417[0.027]	0.378[0.066]	0.338[0.005]	0.320[0.002]
Siamese Triplet	0.401[0.005]	0.387[0.002]	0.356[0.005]	0.358[0.004]	0.466[0.001]	0.446[0.002]	0.376[0.002]	0.367[0.003]
Siamese Pair	0.425[0.006]	0.410[0.005]	0.346[0.003]	0.343[0.007]	0.482[0.004]	0.454[0.004]	0.401[0.003]	0.390[0.004]
REP	0.452[0.001]	0.447[0.003]	0.361[0.003]	0.355[0.003]	0.472[0.045]	0.483[0.024]	0.365[0.004]	0.343[0.007]
SABD	0.484[0.004]	0.482[0.006]	0.400[0.008]	0.401[0.011]	0.538[0.006]	0.522[0.006]	0.443[0.005]	0.435[0.005]

deduplication using the ranking methodology. Amongst all studies in the literature, Poddar et al. [22] was the first to compare different deep learning methods for this task, although they evaluate them using the decision-making methodology.

Regarding the methods proposed by Sun et al. [29], the first relevant point is that $BM25F_{ext}$ achieves a similar curve regarding RR@k and a slightly better MAP value than the Siamese Triplet in Eclipse. Moreover, REP outperforms the two siamese neural networks in Eclipse and Netbeans, and it has comparable results to Siamese Triplet and Siamese Pair in OpenOffice. Even though $BM25F_{ext}$ and REP are simpler methods than the siamese neural networks that contain thousands of parameters, they are able to perform similarly or better than these deep learning models. Finally, it is important to point that REP and $BM25F_{ext}$ have a large standard deviation in OpenOffice and, exclusively $BM25F_{ext}$, in Netbeans.

5.1.1 Time Window Analysis. In the left column of Figure 2, we present RR@k for all the considered methods, for $k = 1, 2, \ldots, 20$, on the four test sets using a time window of one year. The MAP results of these methods in the same experimental setups are reported in Table 4. Despite some minor differences, the findings using a window of one year are similar to the ones with a longer frame of three years – including the fact that SABD constantly outperforms the methods in terms of MAP and RR@k in all datasets.

Extending the window span from one to three years decreases the number of duplicate bug reports whose ranked list never contains the correct master reports. However, we found that this does not necessarily correspond to performance improvements in terms of RR@k. For instance, in Figure 3, we compare the curve of RR@k achieved by SABD in each dataset and time window. Increasing the time frame positively impacts, in general, the SABD performance in OpenOffice and Eclipse, while it marginally reduces its performance in Netbeans and, partially, in Mozilla. We believe that this occurs due to the trade-off between two factors related to the time window length. Expanding the time window raises the upper bound of the RR@k. However, at the same time, it increases the quantity of reports that are searched, making the bug deduplication more difficult [4]. Finally, as shown in Table 4, increasing the time window from one to three years reduces the performance of the methods in terms of MAP. This indicates that MAP is more sensitive to the quantity of reports that must be searched than RR@k.

5.2 Ablation Study

In this section, we perform an ablation study to evaluate the effectiveness of different components of SABD. Ablation study consist

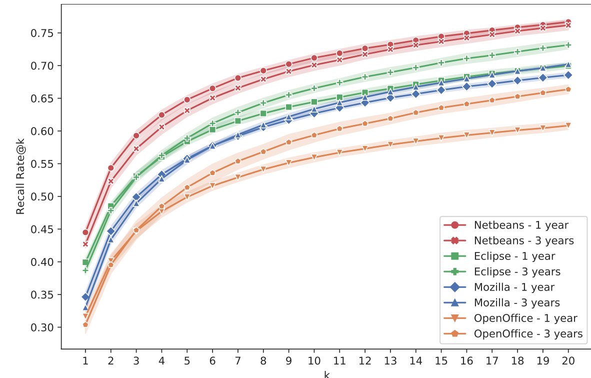

Figure 3: Comparison of SABD performance in Term of RR@k.

in removing a single component from the original architecture, and measuring how much this isolated modification impacts the model performance. The more a component affects the performance, the more effective it is considered.

We test two distinct configurations related to the soft alignment comparison layer. Setup (1) measures the impact of the data exchange by removing the soft alignment comparison layer, thus independently generating the report representations as Sun et al. [29], Xie et al. [37], and Budhiraja et al. [6]. Although this setup may show the layer importance, it is not clear which part of the layer is the most significant. Thus, one also needs to evaluate the importance of SABD capacity to dynamically focus on different parts of a report.

If the model is able to compress the report into a fixed-length vector without losing any relevant information, then SABD will achieve similar results because the FC layer in the soft alignment comparison layer produces similar outputs. However, if SABD is negatively impacted, the summarization of a report into a fixed-length representation is the bottleneck that needs to be replaced by a more powerful mechanism such as the soft-attention alignment.

Setup (2) studies the need for the soft-attention alignment by replacing it with a mechanism similar to that of Poddar et al. [22], which is more powerful than a simple mean-pooling since the fixed-length representation of a report depends on the other report. In (2), the context vector of the query $\overline{x}_i^q$ is the attention mechanism result given by Equation 3 in which the k-th attention coefficient is

proportional to the scaled dot-product:

$$\alpha_k^{q_i} \propto \frac{x_k^c \cdot \text{Mean}[x_1^q, \ldots, x_{|q^t|}^q]}{\sqrt{d^x}},$$

where x_k^c is the k-th word vector of the candidate and Mean[...] is the result of the mean-pooling operator over the word vectors in the query. The candidate context vectors $\overline{x}_j^c$ are produced likewise.

Furthermore, we test four additional setups. In (3), the categorical module is removed from SABD, i.e, only textual data is used for detecting duplicate reports. In (4), we remove the fully-connected layer (Equation 2) that modifies the concatenation of the word and field embedding vectors (v_i^q and v_j^c). In (5), we remove the bi-LSTM in the textual encoder layer, i.e., the mean and max-pooling generate the fixed-length representation of the reports. In (6), field embedding is not concatenated with word embeddings and, thus, the words from the summary and description are identically represented.

Figure 4 and Table 5 report the RR@k and MAP achieved by the seven configurations in the validation dataset of Eclipse. As shown

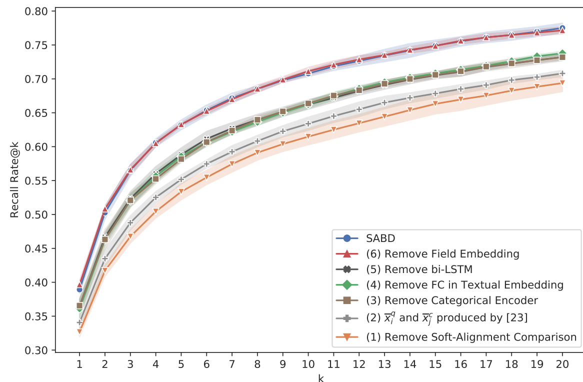

Figure 4: Ablation study in termm of RR@k.

Table 5: Ablation Study in Term of MAP.

Method	MAP	Diff.
SABD	0.500[0.005]	-0.000
(6) Remove Field Embedding	0.505[0.002]	+0.005
(5) Remove bi-LSTM	0.468[0.005]	-0.032
(4) Remove FC in Textual Embedding	0.465[0.008]	-0.035
(3) Remove Categorical Encoder	0.467[0.008]	-0.033
(2) $\overline{x}_i^q$ and $\overline{x}_j^c$ produced by [22]	0.440[0.009]	-0.060
(1) Remove Soft-alignment Comparison	0.424[0.006]	-0.076

in Figure 4 and Table 5, the soft-alignment comparison is the most crucial component of our model since removing this layer from SABD significantly degrades its performance. Besides, the setup (1) is marginally outperformed by setup (2). Both results corroborate the hypothesis that data exchange improves the representations. We also observe that the model performance substantially decreases when the soft-attention alignment is replaced by a less powerful mechanism. This confirms our assumption that summarizing report information into fixed-length representation is the bottleneck of the Poddar et al. [22] model. An architecture that dynamically focuses on distinct information from a report is less prone to lose information and, therefore, performs a better report comparison. We also observe that removing the FC sublayer from the textual embedding layer decreases SABD performance. This result confirms our hypothesis about the importance of projecting the words into a dimension space that better captures word relevance for the bug deduplication. As expected, SABD performs significantly worse when categorical data is not used. This data provides additional information about the report which can help the bug deduplication (e.g., the probability of two reports being duplicate from two different software components is usually low). Further, removing the bi-LSTM considerably decreases the model performance which demonstrates our hypothesis that contextual information about the words is useful for deduplication.

Finally, we find that removing the field embedding does not significantly affect the SABD performance. This is an unexpected result since words from different fields were supposed to have distinct relevance.

6 CONCLUDING REMARKS

We proposed SABD, a novel soft alignment method for bug deduplication. In contrast of Siamese neural networks, SABD exchanges data between the reports before generating their fixed-length representations. The mechanism responsible for this data interchange is more powerful than the one proposed in Poddar et al. [22] because it can dynamically focus on distinct information of a report during the feature extraction of the other report. We experimentally evaluate SABD and competing methods (including two non-deep learning ones) using a methodology based on Sun et al. [29]. This methodology is more adherent to real environments than the ones often used in the literature [6, 10, 22, 37]. SABD significantly outperforms the other methods in all experimental setups.

It is important to notice that, even though competing deep learning methods were implemented to the best of our knowledge, it is not possible to guarantee that those are identical to the ones used in the studies. As shown in the ablation study, the soft-aliment positively impacts the model performance. However, this mechanism has a cost in terms of runtime. As the fixed-length representations of the reports are jointly generated, it is not possible to save computation time by storing them. Therefore, our model is slower than the methods based on siamese neural networks.

ACKNOWLEDGMENTS

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson, Ciena, and EffciOS for funding this project.

REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[2] S. Banerjee, B. Cukic, and D. Adjeroh. 2012. Automated Duplicate Bug Report Classification Using Subsequence Matching. In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*. 74–81. https://doi.org/10.1109/

HASE.2012.38

[3] Sean Banerjee, Zahid Syed, Jordan Helmick, and Bojan Cukic. 2013. A fusion approach for classifying duplicate problem reports. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 208–217.

[4] Sean Banerjee, Zahid Syed, Jordan Helmick, Mark Vere Culp, Kenneth Joseph Ryan, and Bojan Cukic. 2017. Automated triaging of very large bug repositories. *Information & Software Technology* 89 (2017), 1–13.

[5] David M. Blei. 2012. Probabilistic Topic Models. *Commun. ACM* 55, 4 (April 2012), 77–84. https://doi.org/10.1145/2133806.2133826

[6] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava. 2018. Poster: DWEN: Deep Word Embedding Network for Duplicate Bug Report Detection in Software Repositories. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 193–194.

[7] Amar Budhiraja, Raghu Reddy, and Manish Shrivastava. 2018. LWE: LDA Refined Word Embeddings for Duplicate Bug Report Detection *(ICSE '18)*. ACM, New York, NY, USA, 165–166. https://doi.org/10.1145/3183440.3195078

[8] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a Similarity Metric Discriminatively, with Application to Face Verification. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01 (CVPR '05)*. IEEE Computer Society, USA, 539–546. https://doi.org/10.1109/CVPR.2005.202

[9] J. L. Davidson, N. Mohan, and C. Jensen. 2011. Coping with duplicate bug reports in free/open source software projects. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 101–108. https://doi.org/10.1109/VLHCC.2011.6070386

[10] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash. 2017. Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 115–124. https://doi.org/10.1109/ICSME.2017.69

[11] M. Feng, B. Xiang, M. R. Glass, L. Wang, and B. Zhou. 2015. Applying deep learning to answer selection: A study and an open task. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. 813–820. https://doi.org/10.1109/ASRU.2015.7404872

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[13] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).

[14] Alina Lazar, Sarah Ritchey, and Bonita Sharif. 2014. Generating Duplicate Bug Datasets *(MSR 2014)*. ACM, New York, NY, USA, 392–395. https://doi.org/10.1145/2597073.2597128

[15] Johannes Lerch and Mira Mezini. 2013. Finding Duplicates of Your Yet Unwritten Bug Report. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, Washington, DC, USA, 69–78. https://doi.org/10.1109/CSMR.2013.17

[16] Meng-Jie Lin and Cheng-Zen Yang. 2014. An Improved Discriminative Model for Duplication Detection on Bug Reports with Cluster Weighting. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC '14)*. IEEE Computer Society, Washington, DC, USA, 117–122. https://doi.org/10.1109/COMPSAC.2014.18

[17] Meng-Jie Lin, Cheng-Zen Yang, Chao-Yuan Lee, and Chun-Chang Chen. 2016. Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software* 121 (2016), 223 – 233. https://doi.org/10.1016/j.jss.2016.02.022

[18] Ramesh Nallapati. 2004. Discriminative Models for Information Retrieval *(SIGIR '04)*. ACM, New York, NY, USA, 64–71. https://doi.org/10.1145/1008992.1009006

[19] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling *(ASE 2012)*. ACM, New York, NY, USA, 70–79. https://doi.org/10.1145/2351676.2351687

[20] Ankur Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. 2016. A Decomposable Attention Model for Natural Language Inference. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 2249–2255. https://doi.org/10.18653/v1/D16-1244

[21] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[22] Lahari Poddar, Leonardo Neves, William Brendel, Luis Marujo, Sergey Tulyakov, and Pradeep Karuturi. 2019. Train One Get One Free: Partially Supervised Neural Network for Bug Report Duplicate Detection and Clustering. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Industry Papers)*. Association for Computational Linguistics, Minneapolis - Minnesota, 157–165. https://doi.org/10.18653/v1/N19-2020

[23] Tomi Prifti, Sean Banerjee, and Bojan Cukic. 2011. Detecting Bug Duplicate Reports Through Local References *(Promise '11)*. ACM, New York, NY, USA, Article 8, 9 pages. https://doi.org/10.1145/2020390.2020398

[24] Mohamed Sami Rakha, Cor-Paul Bezemer, and Ahmed E. Hassan. 2018. Revisiting the Performance of Automated Approaches for the Retrieval of Duplicate Reports in Issue Tracking Systems That Perform Just-in-time Duplicate Retrieval. *Empirical Software Engineering* 23, 5 (Oct. 2018), 2597–2621. https://doi.org/10.1007/s10664-017-9590-5

[25] Jinfeng Rao, Hua He, and Jimmy Lin. 2016. Noise-Contrastive Estimation for Answer Selection with Deep Neural Networks *(CIKM '16)*. ACM, New York, NY, USA, 1913–1916. https://doi.org/10.1145/2983323.2983872

[26] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 499–510. https://doi.org/10.1109/ICSE.2007.32

[27] K. K. Sabor, A. Hamou-Lhadj, and A. Larsson. 2017. DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 240–250. https://doi.org/10.1109/QRS.2017.35

[28] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[29] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards More Accurate Retrieval of Duplicate Bug Reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 253–262. https://doi.org/10.1109/ASE.2011.6100061

[30] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval *(ICSE '10)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/1806799.1806811

[31] Ashish Sureka and Pankaj Jalote. 2010. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference (APSEC '10)*. IEEE Computer Society, Washington, DC, USA, 366–374. https://doi.org/10.1109/APSEC.2010.49

[32] Ming Tan, Cicero dos Santos, Bing Xiang, and Bowen Zhou. 2016. Improved Representation Learning for Question Answer Matching. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 464–473. http://www.aclweb.org/anthology/P16-1044

[33] Yi Tay, Luu Anh Tuan, and Siu Cheung Hui. 2018. Multi-Cast Attention Networks *(KDD '18)*. ACM, New York, NY, USA, 2299–2308. https://doi.org/10.1145/3219819.3220048

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[35] Shuohang Wang and Jing Jiang. 2016. A Compare-Aggregate Model for Matching Text Sequences. *CoRR* abs/1611.01747 (2016).

[36] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information *(ICSE '08)*. ACM, New York, NY, USA, 461–470. https://doi.org/10.1145/1368088.1368151

[37] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng. 2018. Detecting Duplicate Bug Reports with Convolutional Neural Networks. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Nara, Japan, 416–425. https://doi.org/10.1109/APSEC.2018.00056

[38] C. Yang, H. Du, S. Wu, and I. Chen. 2012. Duplication Detection for Software Bug Reports Based on BM25 Term Weighting. In *2012 Conference on Technologies and Applications of Artificial Intelligence*. 33–38.

[39] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. 2016. Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 127–137. https://doi.org/10.1109/ISSRE.2016.33

[40] Jian Zhou and Hongyu Zhang. 2012. Learning to Rank Duplicate Bug Reports *(CIKM '12)*. ACM, New York, NY, USA, 852–861. https://doi.org/10.1145/2396761.2396869