DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging

Senthil Mani
IBM Research
sentmani@in.ibm.com

Anush Sankaran
IBM Research
anussank@in.ibm.com

Rahul Aralikatte
IBM Research
rahul.a.r@in.ibm.com

ABSTRACT

For a given software bug report, identifying an appropriate developer who could potentially fix the bug is the primary task of bug triaging. Automatic bug triaging is formulated as a classification problem, which takes the bug title and description as the input, and maps it to one of the available developers. A major challenge in doing this is that the bug description usually contains a combination of unstructured text, code snippets, and stack traces making the input data highly noisy. The existing bag-of-words (BOW) models do not consider the semantic information in the unstructured text.

In this research, we propose a novel bug report representation using a deep bidirectional recurrent neural network with attention (DBRNN-A) that learns the syntactic and semantic features from long word sequences in an unsupervised manner. Using attention enables the model to remember and attend to important parts of text in a bug report. For training the model, we use unfixed bug reports (which constitute about 70% of bugs in a typical open source bug tracking system) which were ignored in previous studies.

Another major contribution of this work is the release of a public benchmark dataset of bug reports from three open source bug tracking systems: Google Chromium, Mozilla Core, and Mozilla Firefox. The dataset consists of 383,104 bug reports from Google Chromium, 314,388 bug reports from Mozilla Core, and 162,307 bug reports from Mozilla Firefox. When compared to other systems, we observe that DBRNN-A provides a higher rank-10 average accuracy.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning algorithms*; • **Software and its engineering** → *Software organization and properties.*

KEYWORDS

Bidirectional LSTM, Attention LSTM, Bug Triaging

ACM Reference Format:
Senthil Mani, Anush Sankaran, and Rahul Aralikatte. 2019. DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging. In *6th ACM IKDD CoDS and 24th COMAD (CoDS-COMAD '19), January 3–5, 2019, Kolkata, India.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297001.3297023

CoDS-COMAD '19, January 3–5, 2019, Kolkata, India

ACM ISBN 978-1-4503-6207-8/19/01…$15.00
https://doi.org/10.1145/3297001.3297023

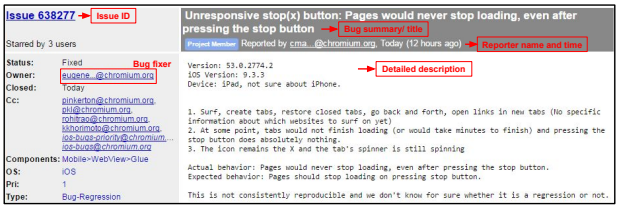

Figure 1: Example of a bug report available in the Google Chromium project, Bug ID: 638277. The bug report usually consists of a brief summary and a detailed description at the time of reporting.

1 INTRODUCTION

Typically, when end-users encounter a bug in a system, they raise an issue in a relevant bug tracking system [7]. Fig 1 shows the standard format of a bug reported in the Google Chromium project. The bug report usually contains the title (also called 'summary') and a detailed description mentioning the steps for reproduction. Once the bugs are *fixed* it's status is updated by the developer who fixed the bug, also called it's *Owner*. The process of bug triaging consists of multiple steps where the first step primarily involves assigning the bug to one of the developers who has the expertise to solve the bug. Thus, in the rest of this research, bug triaging refers to the task of assigning a developer to an open bug [1]. In large scale systems, with high rates of incoming bugs, manually analyzing and triaging a bug report is laborious. Manual bug triaging is usually performed using the report content, primarily consisting of the title and description. While additional sources of input has been explored in literature such as developer profiling from github [3] and using component information [5], majority of the research efforts have focused on leveraging the bug report content for triaging [2] [11] [21] [22] [23] [25] [26]. Using bug report content, automated bug triaging can be formulated as a classification problem, mapping the bug title and description to a developer (class label). However, bug report content contains noisy textual information including code snippets, and stack trace details, etc., as observed in Fig. 1. Processing such unstructured and noisy text data is a major challenge in training a classifier.

1.1 Motivating Example

Consider a labeled bug report example shown in Fig. 2 as a training data point. The bag-of-words (BOW) feature representation of the bug report creates a boolean array marking true (or term-frequency) for each vocabulary word in the bug report [2]. During training, a classifier will learn a mapping between this representation and the corresponding class label *brettw@chromium.org*. Now consider two

Labeled bug report: 599892
Fixed by: brettw@chromium.org
Title: GN should only load each import once
Description: In GN mutliple BUILD files can load the same import. GN caches the results of imports so we don't have to load them more than once. But if two BUILD files load the same import at the same time, there is a race. Rather than lock, the code allows each to load the file and the first one finished "wins". This is based on the theory that the race is rare and processing imports is relatively fast. On Windows, many build files end up with the visual_studio_version.gni file which ends up calling build/vs_toolchain.py. This script can be quite slow (slower than the rest of the entire GN run in some cases). The result is that the race is guaranteed to happen for basically every BUILD file that references the .gni file, and we end up running the script many times in parallel (which only slows it down more). We should add the extra locking to resolve the race before loading rather than after.

Figure 2: A bug report from the Google Chromium bug repository used as a labeled template for training the classifier.

test data points shown in Fig. 3. The actual fixer of the first example, with bug id 634446, is *brettw@chromium.org* while the second example bug with id 616034 is fixed by *machenb...@chromium.org*. However, based on BOW features there are 12 words common between test report#1 and the train report, while there are 21 words common between test report#2 and the train report. Hence, a BOW model mis-classifies the test bug report#2 with id 616034 to say that *brettw@chromium.org* should fix the bug. The reasons for the misclassification are: (i) BOW feature model considers the sentence as a bag-of-words losing the order (context) of words, and (ii) the semantic similarity between synonymous words in the sentence are not considered. Even though a bag-of-n-grams model considers a small context of word order, they suffer from high dimensionality and sparse data [10]. The semantic similarity between word tokens can be learned using a skip-gram based neural network model called *word2vec* [17]. This model relies on distributional hypothesis which claims that words that appear in the same context in the sentence are likely to share the same semantic meaning. Ye et al., [27] built a shared word representation using *word2vec* for word tokens present in code and word tokens present in natural language. The main disadvantage of *word2vec* is that it learns a semantic representation of individual word tokens, but does not consider a sequence of word tokens such as a sentence. An extension of *word2vec* called *paragraph vector* [15] considers the ordering of words, but with limited success.

2 RESEARCH CONTRIBUTIONS

Learning semantic representation from large pieces of text (such as description in bug reports), preserving the order of words, is a challenging research problem. Thus, we propose a deep learning technique, which will learn a succinct fixed-length representation of the bug report content in an unsupervised manner i.e., the representation will be learned directly using the data without the need for manual feature engineering. The main research questions (RQ) that we address in this research are as follows:

(1) **RQ1**: Is it feasible to perform automated bug triaging using deep learning?
(2) **RQ2**: How does the unsupervised feature engineering approach perform, compared to traditional feature engineering based approaches?

Bug report #1 to be triaged: 634446
Fixed by: brettw@chromium.org
Title: GN toolchain_args should be a scope rather than a function
Description: Currently in a toolchain args overrides are:
toolchain_args() {
foo = 1
bar = "baz" }
We're transitioning this to be a scope type:
toolchain_args = {
foo = 1
bar = "baz"}
which will allow the gcc_toolchain template to forward values from the invoker without it having to know about all build args ever overridden in the entire build.

Bug report #2 to be triaged: 616034
Fixed by: machenb...@chromium.org
Title: GN toolchain_args should be a scope rather than a function
Description: Can v8_use_external_startup_data be overridden in a chromium build? On the one hand, there is the default, declared as a gn arg, which is true. On the other hand, there is "v8_use_external_startup_data = !is_ios" as a build override in chromium. There is no logic to not override if the user changes the gn arg. The same would hold for v8_optimized_debug.
This would mean that the declared arg cannot be overwritten via command line.

Figure 3: Two test example bug reports from Google Chromium bug repository for which a suitable developer has to be predicted. The overlapping words with the training bug are highlighted.

(3) **RQ3**: How does the number of training samples per class affect the performance of the classifier?
(4) **RQ4**: What is the effect of using only the title of the bug report in performing triaging when compared with using the description as well?
(5) **RQ5**: Is transfer learning effective in this domain?

The main contributions of this research are summarized as follows:

- A novel approach for bug report representation [1] is proposed using DBRNN-A: Deep Bidirectional Recurrent Neural Network with Attention. The proposed algorithm is capable of remembering the context over a long sequence of words and uses Long Short-Term Memory units (LSTM) [19] as building blocks.
- The untriaged and unsolved bug reports constitute about 70% in an open source bug repository and are usually ignored in the literature [11]. In this research, we provide a mechanism to leverage all the untriaged bugs to learn the bug representation model in an unsupervised manner.
- Experimental data (bug reports) are collected from three open source bug repositories: 3, 83, 104 from Chromium, 3, 14, 388 from Mozilla Core, and 1, 62, 307 from Mozilla Firefox. Performance of various classifiers trained on different train-test splits of the datasets [11] [14] are neither comparable nor reproducible. Thus, to make our research reproducible, the entire dataset along with the exact train-test splits and source code of our approach are made publicly available for research purposes[2].
- We further study the effectiveness of the proposed method in a cross-domain testing scenario (transfer learning). By

[1] We use the terms representation learning and feature learning interchangeably
[2] Made available at: http://bugtriage.mybluemix.net/

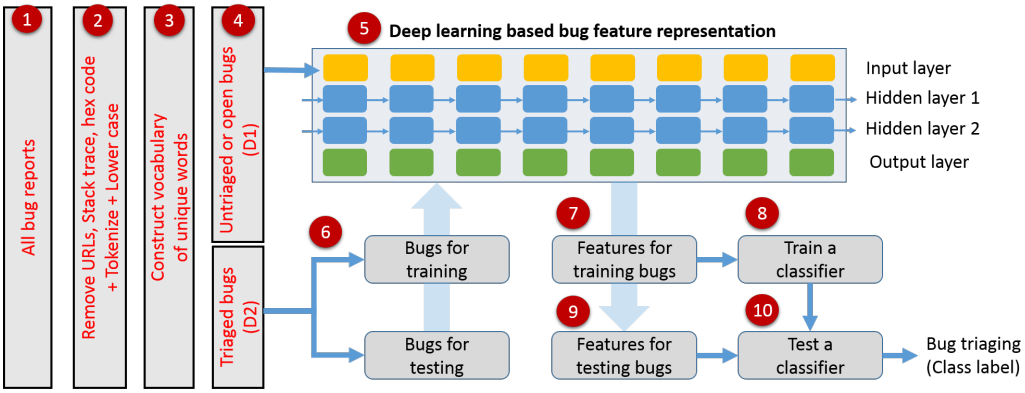

Figure 4: The flow diagram of the overall proposed algorithm highlighting important steps.

training the model with bugs from the Chromium project and we triage bugs in Core and Firefox projects (Mozilla bug repository) and articulate the results.

3 PROPOSED APPROACH

The problem of automated bug triaging of software bug reports is formulated as a supervised classification approach with the input data being the bug summary and description. Fig. 4 highlights the major steps involved the proposed automated bug triaging algorithm and are explained as follows:

(1) a bug corpus having title, description, reported time, status, and owner is extracted from an open source bug tracking system,
(2) handling the URLs, stack trace, hex code, and the code snippets in the unstructured description requires customized training of the model, and hence in this research work, such content are removed in the pre-processing stage,
(3) a set of unique words that occurred at least k-times in the corpus are extracted as the vocabulary,
(4) the triaged bugs (D2) are used for classifier training and testing, while all the untriaged/open bugs (D1) are used to train the feature extractor (DBRNN-A),
(5) the DBRNN-A learns a bug representation considering the bug title and description as a sequence of word tokens,
(6) the triaged bugs (D2) are split into train and test data and 10 fold cross validation is used to remove training bias,
(7) feature representation for the training bug reports are extracted using the learned DB-RNN algorithm,
(8) a supervised classifier is trained for performing developer assignment as a part of bug triaging process,
(9) feature representation of the testing bugs are then extracted using DBRNN-A,
(10) using the extracted features and the learned classifier, a probability score for every potential developer is predicted and the accuracy is computed on the test set.

The proposed approach is different from the traditional pipeline for automated bug triaging in the following ways: (i) in step 4, the untriaged bugs (D1) are not ignored and (ii) use of unsupervised feature learning for bug report representation instead of manual feature engineering.

3.1 Deep Bidirectional Recurrent Neural Network with Attention (DBRNN-A)

This section briefly explains the inner working of the DBRNN-A as shown in Fig. 5. For each word in the vocabulary, a $|P|$-dimensional *word2vec* representation [18] is learned. As shown in Fig. 5 (a), a DBRNN-A with LSTM units is learned over this word representation, to obtain a $|D|$-dimensional feature representation of the entire bug report (title + description). Let our RNN contain a hidden layer with m hidden units, $\mathbf{h} = \{\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_m}\}$. The input to the system is a sequence of word representations, $\mathbf{x} = \{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m}\}$, and produces a sequence of outputs $\mathbf{y} = \{\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_m}\}$. Each hidden unit is a state model converting the previous state, s_{i-1} and a word, x_i to the next state, s_i and an output word, y_i. The term "recurrent" explains that every hidden unit performs the same function in recurrent fashion, $f : \{s_{i-1}, x_i\} \rightarrow \{s_i, y_i\}$. Intuitively, the state s_i carries the cumulative information of the i previous words observed. The output y_m obtained from the last hidden node is a cumulative representation of the entire sentence. For example, consider the tokenized input sentence provided in Fig. 5. When $i = 1$, x_i is the $|P|$-dimensional *word2vec* representation of the input word, *unresponsive* and the previous state s_0 is randomly initialized. Using the LSTM function f, the current state s_1 and the word output y_1 are predicted. Given the next word *stop* and the current state s_1, the same function f is used to predict s_2 and y_2. The shared function reduces the number of learnable parameters as well as retains the context from all the words in the sequence. For language modeling or learning sentence representations, the ground truth y_i is the next word in the sequence x_{i+1}, that is, upon seeing the previous words in the sentence the network tries to predict the next word. LSTM function [8] has a memory cell to store the context information over longer sentences.

Further, to selectively remember and learn from the important words in a bug report, an attention model is employed. An attention vector is derived by performing a weighted summation of all the

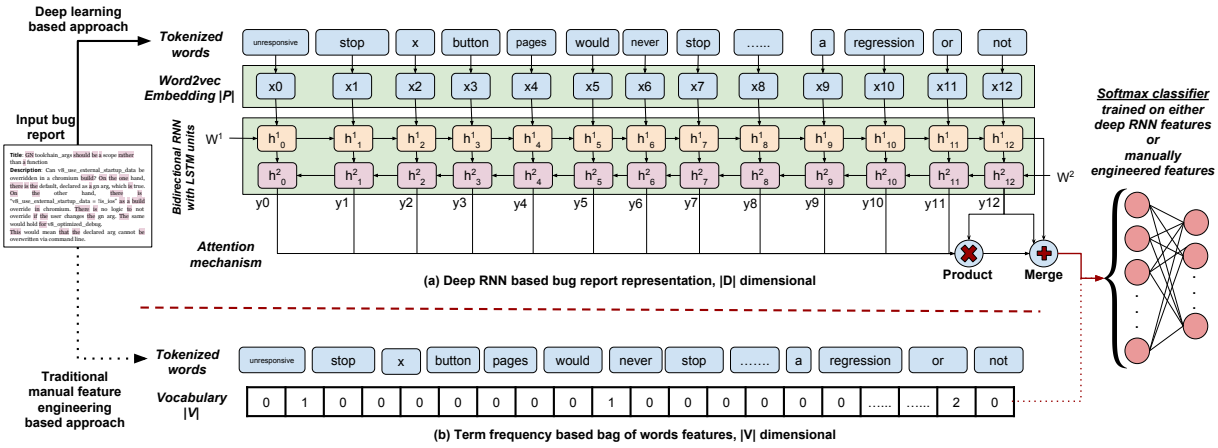

Figure 5: Working of DBRNN-A for an example bug report shown in Fig. 1. It can be seen that the deep network has multiple hidden layers, learning a complex hierarchical representation from the input data. As a comparison, tf based bag-of-words (BOW) representation for the same example sentence is also shown.

computed outputs, y_i, as follows:

$$a_m = \sum_{i=1}^{m} \alpha_i y_i \tag{1}$$

Intuitively, α_i associates a weight to each word implying the importance of that word for classification. Two different deep RNN based feature models are learned, one with input word sequence running forward and one with input word sequence running backward. The final representation, r, obtained for a bug report, is provided as follows:

$$r = \underbrace{y_m \oplus a_m}_{\text{forward LSTM}} \oplus \underbrace{y_m \oplus a_m}_{\text{backward LSTM}} \tag{2}$$

where $\oplus$ represents concatenation of the vectors. In comparison, as shown in Fig. 5 (b), a term frequency based BOW model would produce a $|V|$-dimensional representation for the same bug report, where V is the size of vocabulary. Typically, the size of $|P|$ is chosen as 300 [18] and the size of D will be less than $4|P|$ (< 1200) is much smaller than the size of $|V|$. For example, consider $10,000$ bugs used for training with $250,000$ unique words ($|V|$). BOW model representation would produce a sparse feature matrix of size $10,000 \times 250,000$, while the proposed DBRNN-A would produce a dense and compact representation with a feature matrix of size $10,000 \times 1,200$.

The entire deep learning model was implemented in Python using Keras. To the best of our knowledge, this is the first time a deep sequence learning model has been applied to learn a bug representation and use them to learn a supervised model for automated software bug triaging.

3.2 Classifying (Triaging) a Bug Report

The aim of the supervised classifier is to learn a function, C, that maps a bug feature representation to a set of appropriate developers. Formulating automated bug triaging as a supervised classification problem has been well established in literature [5] [25]. However, it is well understood that a classification is only as good as the quality of features. Hence, the major contribution in this research is to propose a better bug report representation model and to improve the performance of existing classifiers. In this research we use a simple softmax classifier, a popular choice of classifier among deep learning practitioners [8] [6] [9]. Softmax classifier is a generalization of logistic regression for multi-class classification, taking the features and providing a vector of scores with length equal to the number of the classes. A softmax classifier normalizes these score values and provides an interpretable probability value of a bug report belonging to particular lass.

Property	Chromium	Core	Firefox
Total bugs	383,104	314,388	162,307
Bugs for learning feature	263,936	186,173	138,093
Bugs for classifier	118,643	128,215	24,214
Vocabulary size $\|V\|$	71,575	122,578	57,922

Table 1: Summary of the three bug repositories used in our experiments.

4 LARGE SCALE PUBLIC BUG TRIAGE DATASET

A large corpus of bug report data is obtained from three popular open source systems: Chromium[3], Mozilla Core, and Mozilla Firefox[4] and the data collection process is explained in this section. To make this research reproducible, the entire data along with the exact train-test protocol and with source code is made available at: http://bugtriage.mybluemix.net/.

4.1 Data Extraction

Bug reports from the Google Chromium project were downloaded for the duration of August 2008 (Bug ID: 2) - July 2016 (Bug ID: 633012). A total of 383, 104 bugs where collected with the bug title, description, bug owner, and reported time. The developer in the "owner" field is considered as the ground truth triage class for the given bug[5]. Bugs with status as *Verified* or *Fixed*, and type as *bug*,

[3]https://bugs.chromium.org/p/chromium/issues/list
[4]https://bugzilla.mozilla.org/
[5]https://www.chromium.org/for-testers/bug-reporting-guidelines/triage-best-practices

and has a valid ground truth bug owner are used for training and testing the classifier while rest of the bugs are used for learning the feature representation. However, we noticed that there were a total of 11,044 *bug* reports with status as *Verified* or *Fixed* and did not have a valid owner associated. These bugs are considered as open bugs, resulting in a total of 263,936 (68.9%) bug reports being used for feature learning, and 118,643 (31%) bugs for training and testing the classifier.

Data from two popular components of Mozilla bug repository are extracted: Core and Firefox. 314,388 bug reports are extracted from Mozilla Core reported between April 1998 (Bug ID: 91) and June 2016 (Bug ID: 1278040), and 162,307 bug reports are extracted from Mozilla Firefox reported between July 1999 (Bug ID: 10954) and June 2016 (Bug ID: 1278030). The developer in the "Assigned To" field is considered as the ground truth triage class during classification. Bug reports with status as *verified fixed*, *resolved fixed*, and *closed fixed* are used for classifier training and testing. However, some of the *fixed* reports did not have a developer assigned to it; (7219/135434 = 5.33%) in Core and (3716/27930 = 13.3%) in Firefox. After ignoring these bugs, a final number of 1,28,215 bugs for Core and 24,214 bugs for Firefox are considered for classifier training and testing. The summary of the datasets is provided in Table 1.

4.2 Data Preprocessing

The three datasets are preprocessed independently using the same set of steps and a benchmark protocol is created. For every bug report, only the title and description text are considered. Preprocessing of the unstructured textual content involves removing URLs, hex code, and stack trace information, and converting all text to lower case letters. Tokenization of words is performed using Stanford's *NLTK* package[6]. A vocabulary of all words is constructed using the entire corpus. To remove rarely occurring words and reduce the vocabulary size, usually the top-*F* frequent words are considered or only those words occurring with a minimum frequency are considered [27]. For the extracted data, we experimentally observed that a minimum word frequency of 5 provided a good trade-off between the vocabulary size and performance.

4.3 Training Data for Feature Learning

In our data split mechanism, the classifier testing data is unseen data and hence cannot be used for training DBRNN-A. A design choice was taken for not using the classifier training data for training the DBRNN-A, as including them only marginally improved the accuracy but largely increased the training time. Thus, only the untriaged bugs (explained in the data extraction subsection) is used for training the feature extractor. Also, using non-overlapping datasets for training the feature model and classifier model respectively, highlights the generalization capability of the extracted features.

4.4 Training Data for Classification

For training and testing the supervised classifier, a 10-fold cross validation model as proposed by Betternburg et al [4] is followed. All the fixed bug reports are arranged in chronological order and split into 11 sets. Starting from the second fold, every fold is used as a test set, with the cumulation of 'only' the previous folds for training. Typically, in an open source project the developers keep changing overtime, and hence chronological splitting ensures that the train and test sets have highly overlapping developers. Further, in order to make the training effective, we need more training samples per developer. In a recent study, Jonsson et al., [11] trained using those developers who have at least addressed 50 bug reports i.e., minimum number of training samples per class is 50. From different studies in literature [2] [5], it is clear that this threshold parameter affects the classification performance. Thus, in this research we study the direct relation between the threshold value and the classification performance, by having four different thresholds for the minimum number of training samples per class as 0, 5, 10, 20. To perform a closed training experiment, it is made sure that all the classes available in testing are available for training while there are additional classes in training which are not available in the test set. Thus, for every test bug report with an owner, the classifier is already trained with other bugs trained by the same owner.

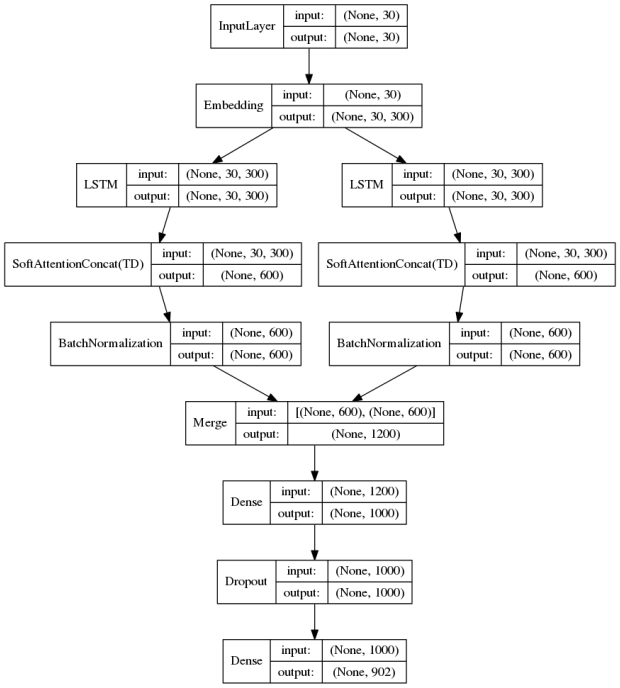

Figure 6: The architecture of DBRNN-A detailing all the parameters of the model.

5 EXPERIMENTAL EVALUATION

5.1 Evaluation Protocol and Metric

For a given bug report, the trained classifier provides a probability value for every developer, denoting their association with the bug report. Thus, the evaluation metric that is used is the top-*k* accuracy, which denotes the ratio of the bug reports for which the actual developer is present in the top-*k* retrieved results. Across the cross validation (CV) sets, varying classes or a set of developers are used. Thus during CV#1, the classes used for training and testing is different from the classes used in CV#2. Thus, as the classifier model

[6]http://www.nltk.org/api/nltk.tokenize.html

Sampl.	Classifier	CV#1	CV#10	Average
>= 0	BOW + MNB	21.9	33.3	26.0 ± 3.0
	BOW + Cosine	18.4	21.5	20.2 ± 1.2
	BOW + SVM	11.2	10.8	10.1 ± 0.6
	BOW + Softmax	12.5	08.7	09.1 ± 1.1
	DBRNN-A + Softmax	34.9	39.7	**37.9 ± 1.9**
>= 5	BOW + MNB	22.2	33.6	26.2 ± 3.1
	BOW + Cosine	18.6	22.0	20.4 ± 1.3
	BOW + SVM	11.3	09.0	09.2 ± 1.0
	BOW + Softmax	12.8	11.4	10.8 ± 0.9
	DBRNN-A + Softmax	32.2	38.2	**36.8 ± 2.2**
>= 10	BOW + MNB	22.4	34.3	26.6 ± 3.3
	BOW + Cosine	18.8	21.0	20.6 ± 1.3
	BOW + SVM	12.2	11.9	11.7 ± 0.4
	BOW + Softmax	11.9	11.5	11.3 ± 0.2
	DBRNN-A + Softmax	36.2	46.0	**41.8 ± 3.1**
>= 20	BOW + MNB	22.9	36.0	27.8 ± 3.7
	BOW + Cosine	19.3	23.0	21.5 ± 1.4
	BOW + SVM	12.2	11.9	11.7 ± 0.3
	BOW + Softmax	11.9	11.7	11.5 ± 0.3
	DBRNN-A + Softmax	36.7	47.0	**42.7 ± 3.5**

Table 2: Rank-10 accuracy on Google Chromium project.

Sampl.	Classifier	CV#1	CV#10	Average
>= 0	BOW + MNB	21.6	32.1	29.5 ± 3.6
	BOW + Cosine	16.3	29.1	22.6 ± 3.9
	BOW + SVM	13.6	14.1	13.6 ± 1.0
	BOW + Softmax	14.3	10.8	10.8 ± 1.4
	DBRNN-A + Softmax	30.1	35.1	**33.9 ± 1.7**
>= 5	BOW + MNB	20.7	36.2	31.5 ± 5.2
	BOW + Cosine	15.7	29.9	23.5 ± 4.6
	BOW + SVM	16.4	13.1	12.9 ± 1.5
	BOW + Softmax	14.9	14.0	12.7 ± 1.2
	DBRNN-A + Softmax	33.8	38.0	**35.9 ± 2.1**
>= 10	BOW + MNB	18.4	42.5	34.5 ± 7.7
	BOW + Cosine	16.0	35.5	25.1 ± 6.2
	BOW + SVM	17.5	16.2	16.7 ± 0.6
	BOW + Softmax	15.6	14.1	14.3 ± 0.6
	DBRNN-A + Softmax	32.5	39.6	**36.1 ± 2.1**
>= 20	BOW + MNB	21.3	41.8	35.1 ± 7.0
	BOW + Cosine	16.8	38.9	28.9 ± 8.2
	BOW + SVM	14.6	16.4	15.5 ± 0.9
	BOW + Softmax	18.8	15.3	14.0 ± 2.4
	DBRNN-A + Softmax	33.3	43.3	**38.8 ± 3.2**

Table 3: Rank-10 accuracy on the Mozilla Core project.

across the CV is trained on different classes, taking the average accuracy would only provide a ballpark number of the performance, while is not accurately interpretable. Thus, it is required to report the top-k accuracy of each cross validation set to understand the variance introduced in the model training [13]. Due to the lack of space, we only report the CV#1 and CV#10 scores along with the average of all the 10 CVs.

Sampl.	Classifier	CV#1	CV#10	Average
>= 0	BOW + MNB	19.1	35.55	27.4 ± 5.2
	BOW + Cosine	17.3	30.1	25.7 ± 4.1
	BOW + SVM	13.4	14.6	14.1 ± 1.0
	BOW + Softmax	11.9	13.6	14.6 ± 1.9
	DBRNN-A + Softmax	33.6	38.1	**36.5 ± 1.7**
>= 5	BOW + MNB	21.1	36.5	33.1 ± 5.1
	BOW + Cosine	20.8	35.2	28.5 ± 4.8
	BOW + SVM	14.4	15.2	16.5 ± 1.1
	BOW + Softmax	18.2	13.7	14.8 ± 1.8
	DBRNN-A + Softmax	27.6	44.5	**40.1 ± 5.3**
>= 10	BOW + MNB	21.7	38.5	33.1 ± 4.8
	BOW + Cosine	18.1	36.6	28.7 ± 5.8
	BOW + SVM	09.9	12.8	11.9 ± 1.1
	BOW + Softmax	14.3	12.7	12.1 ± 1.8
	DBRNN-A + Softmax	35.1	51.4	**44.8 ± 5.6**
>= 20	BOW + MNB	22.0	38.4	30.4 ± 6.2
	BOW + Cosine	18.4	38.3	29.8 ± 6.3
	BOW + SVM	18.7	21.9	19.6 ± 2.2
	BOW + Softmax	16.5	12.9	13.1 ± 1.3
	DBRNN-A + Softmax	38.9	55.8	**46.6 ± 6.4**

Table 4: Rank-10 accuracy on the Mozilla Firefox project.

For learning the feature representation, a DBRNN-A is constructed having 300 LSTM units and the dropout probability is 0.3. A categorical cross entropy loss function is used with Adam optimizer, learning rate as 0.001, and trained for 100 epochs with early stopping. The model architecture and parameters utilized are shown in Fig. 6

5.2 Comparison with Existing Algorithms

The major challenge in cross comparison of algorithm performance is the lack of a public benchmark dataset and open implementations of the existing research. Thus, the bug triaging accuracy obtained in the previous research works cannot be compared with the proposed approach, unless the results are shown on the same dataset. Thus, we implement some of the previously successful approaches for automated bug triaging from literature like [2] [25] [11] and compare it with our system using our benchmark dataset. For these models, tf-idf based BOW features are used to represent the title and description from a bug report, as shown in Fig. 5. Using these features, we evaluate the performance of four different classifiers: (i) Softmax classifier [20], (ii) Support Vector Machine (SVM) [24], (iii) Multinomial Naive Bayes (MNB) [12], and (iv) Cosine distance based matching [16]. The four supervised classifiers are implemented using the Python *scikit-learn*[7] package. All these four classifiers use only the triaged (labeled) portion of the dataset and do not use the untriaged bug reports.

5.3 Result Analysis

The results obtained in the Google Chromium, Mozilla Core, and Mozilla Firefox datasets are shown in Table 2, Table 3, and Table 4, respectively. The main research questions focused in this paper are answered using the obtained results.

[7] http://scikit-learn.org/

Paper	Information used	Feature extracted	Approach	Dataset	Performance
Bhattacharya et al., 2010 [5]	title, description, keywords, product, component, last developer activity	tf-idf + bag-of-words	Naive Bayes + Tossing graph	Eclipse# 306,297 Mozilla# 549,962	Rank#5 accuracy 77.43% Rank#5 accuracy 77.87%
Tamrawi et al., 2011 [22]	title, description	terms	A fuzzy-set feature for each word	Eclipse# 69829	Rank#5 accuracy 68.00%
Anvik et. Al., 2011 [2]	title, description	normalized tf	Naive Bayes, EM, SVM, C4.5, nearest neighbor, conjunctive rules	Eclipse# 7,233 Firefox# 7,596	Rank#3 prec. 60%, recall 3% Rank#3 prec. 51%, recall 24%
Xuan et. Al., 2012 [26]	title, description	tf-idf, developer prioritization	Naive Bayes, SVM	Eclipse# 49,762 Mozilla# 30,609	Rank#5 accuracy 53.10% Rank#5 accuracy 56.98%
Shokripour et al. 2013 [21]	title, description, detailed source code info	weighted unigram noun terms	Bug location prediction + developer expertise	JDT-Debug# 85 Firefox# 80	Rank#5 accuracy 89.41% Rank#5 accuracy 59.76%
Wang et al., 2014 [23]	title, description	tf	Active developer cache	Eclipse# 17,937 Mozilla# 69,195	Rank#5 accuracy 84.45% Rank#5 accuracy 55.56%
Xuan et. al., 2015 [25]	title, description	tf	feature selection with Naive Bayes	Eclipse# 50,000 Mozilla# 75,000	Rank#5 accuracy 60.40% Rank#5 accuracy 46.46%
Badashian et. al., 2015 [3]	title, description, keyword, project language, tags from stackoverflow, github	Keywords from bug and tags	Social expertise with matched keywords	20 GitHub projects, 7144 bug reports	Rank#5 accuracy 89.43%
Jonsson et. al., 2016 [11]	title, description	tf-idf	Stacked Generalization of a classifier ensemble	Industry# 35,266	Rank#1 accuracy 89%

Table 5: Summary of various machine learning based bug triaging approaches available in literature, explaining the features and approach used along with its experimental performance.

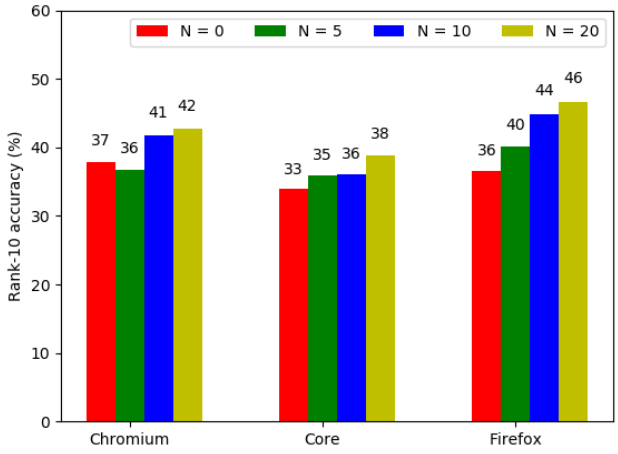

Figure 7: The rank-10 average accuracies of DBRNN-A on all three datasets.

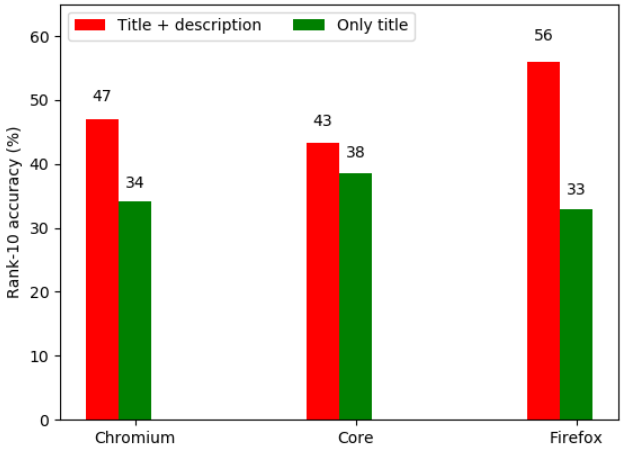

Figure 8: The rank-10 average accuracy of DBRNN-A on all three datasets by using (i) only title, and (ii) title along with the description.

RQ1: Is it feasible to perform automated bug triaging using deep learning?
From the obtained results, it can be observed that the DBRNN-A approach is potentially competent for bug triaging with a rank-10 triaging accuracy in the range of 34 – 47%. It is also clear from the results that, to build a industry standard system, we cannot discard information like stack traces and code snippets which may help in getting a better accuracy. All the experiments are executed in an Intel(R) Xeon(R) CPU E5-2660 v3, running at 2.60GHz and with a Tesla K80 GPU. Learning the feature model and training the

classifier are usually offline tasks and do not contribute towards the testing time. For example in the Google Chroimum dataset, training the DBRNN-A takes about 300 seconds per epoch. For the entire CV#10 subset, training and testing the softmax classifier takes about 121 seconds and 73 seconds respectively. However, after training the models, developer assignment for a new bug report takes only 8 milliseconds using the proposed approach (feature extraction + classification), highlighting it's efficiency.

RQ2: How does the unsupervised feature engineering approach perform, compared to traditional feature engineering approaches?

It can be concretely observed from the results that the feature learning using DBRNN-A outperforms the traditional BOW feature model. In Chromium dataset, rank-10 average accuracy of BOW + Softmax is around $9 - 12\%$, while the best performing classifier provides $26 - 28\%$. This shows the challenging nature of the bug triaging problem in the large dataset that we have created. However, DBRNN-A provides a rank-10 average accuracy in the range of $37 - 43\%$ improving results by $12 - 15\%$. Similarly in Mozilla Core, we observe a $3 - 5\%$ improvement and in Mozilla Firefox, we observe a $7 - 17\%$ improvement in rank-10 average accuracy by using deep features because the DBRNN-A could retain context from longer sentences in a bug report. From the results, we also observe that for BOW features MNB and cosine distance based matching outperforms SVM and softmax classifier. Although SVM is a popular supervised classifier, for real numbered sparse features, feature independence which is assumed both in MNB and cosine distance matching proves successful.

RQ3: How does the number of training samples per class affect the performance of the classifier?

Both intuitively and experimentally, we find that as the minimum number of training samples per class is increased, the performance of the classification improved across all the bug repositories by learning better classification boundaries. For instance in the Chromium dataset, when a classifier is trained with threshold as 0, DBRNN-A produced an average rank-10 accuracy of 37.9% and steadily increased to 42.7% when threshold is 20. Fig 7 captures the improvement in rank-10 average accuracy for all the three datasets. However, for the collected data, having a threshold greater than 20 did not improve the classification accuracy. Also, as we proceed from CV#1 from CV#10, we observe that the performance of DBRNN-A increases. Despite the fact that there are increased number of testing classes, the availability of increased training data improves the classification performance. Thus, empirically the more training data is available for the classifier, the better it's performance. Also, across the cross validations there is about $(2 - 7)\%$ standard deviation in all datasets. This emphasizes the importance of studying the performance of each cross validation set along with the average accuracy.

RQ4: What is the effect of using only the title of the bug report in performing triaging when compared with using the description as well?

The performance of DBRNN-A was studied by using only the title (summary) of the bug report and completely ignoring the description information. The experiments were conducted on all three datasets, with the minimum number of train samples N=20 and CV#10. Fig. 8 compares the rank-10 average accuracy on all three datasets with and without the description content. It can be clearly observed that discarding description significantly reduces the performance of triaging of upto 23%.

RQ5: Is transfer learning effective in this domain?

Transfer learning reduces the offline training time significantly by re-using a model trained on another dataset. However, most of the models fail while transferring the learned model across datasets. The effectiveness of DBRNN-A in transfer learning is studied, by using the pre-trained features trained on the Chromium dataset and re-training only the classifier on the Core and Firefox datasets respectively. The average rank-10 accuracy obtained on the Core and Firefox test sets when N=20 are 39.6% and 43% respectively. The obtained results are comparable with the results obtained by training and testing DBRNN-A on the same datasets from scratch. This shows that the proposed approach is capable of using a model trained on one dataset to triage bug reports in another dataset, effectively.

6 RELATED WORK

Table 5 presents a list of closely related work on bug triaging arranged in a chronological order (year 2010 to 2016). A majority of previous techniques have used bug title and description [2] [22] [25] [26] because they are available at the time of ticket submission and do not change in tickets' lifecyle. Bhattacharya et. al. [5] use additional attributes such as product, component, and the last developer activity to shortlist developers. Shokripour et al. [21] use code information for improved performance. Badashian et. al. [3] identify developers' expertise using stack overflow and keywords from bug description.

7 CONCLUSION

In this research we proposed a novel software bug report (title + description) triaging system using a Deep Bi-directional Recurrent Neural Network with Attention (DBRNN-A). The proposed system learns a paragraph level feature representation preserving the ordering of words over a longer context and also the semantic relationship. The performance of four different classifiers, multinomial naive Bayes, cosine distance, support vector machines, and softmax classifier are compared. To perform experimental analysis, bug reports from three popular open source bug repositories are collected - Google Chromium (383,104), Mozilla Core (314,388), and Mozilla Firefox (162,307). The dataset and the code is made available in http://bugtriage.mybluemix.net/. Experimental results shows DBRNN-A along with the softmax classifier outperforms the other models, improving the rank-10 average accuracy in all three datasets. Further, it was studied that using only the title information for triaging significantly reduces the classification performance highlighting the importance of description. The transfer learning ability of the deep learning model is experimentally shown, where the model learned on the Chromium dataset, competitively triaged bugs in the Mozilla datasets. Additionally, the dataset along with its complete benchmarking protocol and implemented source code is made publicly available to increase the reproducibility of this research.

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *International Conference on Software Engineering*. 361–370.

[2] John Anvik and Gail C Murphy. 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology* 20, 3 (2011), 10.

[3] Ali Sajedi Badashian, Abram Hindle, and Eleni Stroulia. 2015. Crowdsourced bug triaging. In *International Conference on Software Maintenance and Evolution*. IEEE, 506–510.

[4] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate bug reports considered harmful ... really?. In *International conference on Software maintenance*. IEEE, 337–345.

[5] Pamela Bhattacharya and Iulian Neamtiu. 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *International Conference on Software Maintenance*. 1–10.

[6] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[7] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*. 23–32.

[8] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *International Conference on Acoustics, Speech and Signal Processing*. 6645–6649.

[9] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18, 5 (2005), 602–610.

[10] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *International Conference on Software Engineering*. 837–847.

[11] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 21, 4 (2016), 1533–1578.

[12] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. 2004. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence*. Springer, 488–499.

[13] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, Vol. 14. 1137–1145.

[14] Ahmed Lamkanfi, Javier Perez, and Serge Demeyer. 2013. The Eclipse and Mozilla Defect Tracking Dataset: a Genuine Dataset for Mining Bug Information. In *Working Conference on Mining Software Repositories*.

[15] Quoc V Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents.. In *International Conference on Machine Learning*, Vol. 14. 1188–1196.

[16] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. 2006. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, Vol. 6. 775–780.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[19] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. 2014. Dropout improves recurrent neural networks for handwriting recognition. In *International Conference on Frontiers in Handwriting Recognition*. 285–290.

[20] Mark Schmidt, Nicolas Le Roux, and Francis Bach. 2013. Minimizing finite sums with the stochastic average gradient. *arXiv preprint arXiv:1309.2388* (2013).

[21] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. 2013. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Working Conference on Mining Software Repositories*. 2–11.

[22] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. 2011. Fuzzy set-based automatic bug triaging: NIER track. In *International Conference on Software Engineering*. 884–887.

[23] Song Wang, Wen Zhang, and Qing Wang. 2014. FixerCache: unsupervised caching active developers for diverse bug triage. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 25.

[24] Ting-Fan Wu, Chih-Jen Lin, and Ruby C Weng. 2004. Probability estimates for multi-class classification by pairwise coupling. *The Journal of Machine Learning Research* 5 (2004), 975–1005.

[25] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2015. Towards effective bug triage with software data reduction techniques. *IEEE Transactions on Knowledge and Data Engineering* 27, 1 (2015), 264–280.

[26] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. Developer prioritization in bug repositories. In *International Conference on Software Engineering*. 25–35.

[27] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *International Conference on Software Engineering*. 404–415.