DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging

Senthil Mani, Anush Sankaran, Rahul Aralikatte
IBM Research, India
{sentmani,anussank,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 a bug triaging process. A bug title (summary) and a detailed description is present in most of the bug tracking systems. Automatic bug triaging algorithm can be formulated as a classification problem, which takes the bug title and description as the input, mapping it to one of the available developers (class labels). The major challenge is that the bug description usually contains a combination of free unstructured text, code snippets, and stack trace making the input data highly noisy. The existing bag-of-words (BOW) feature models do not consider the syntactical and sequential word information available in the unstructured text.

In this research, we propose a novel bug report representation algorithm using an attention based deep bidirectional recurrent neural network (DBRNN-A) model that learns a syntactic and semantic feature from long word sequences in an unsupervised manner. Instead of BOW features, the DBRNN-A based bug representation is then used for training the classifier. Using an attention mechanism enables the model to learn the context representation over a long word sequence, as in a bug report. To provide a large amount of data to learn the feature learning model, the unfixed bug reports (constitute about 70% bugs in an open source bug tracking system) are leveraged upon as an important contribution of this research, which were completely ignored in the previous studies. Another major contribution is to make this research reproducible by making the source code available and creating a public benchmark dataset of bug reports from three open source bug tracking system: Google Chromium, Mozilla Core, and Mozilla Firefox. For our experiments, we use 383,104 bug reports from Google Chromium, 314,388 bug reports from Mozilla Core, and 162,307 bug reports from Mozilla Firefox. Experimentally we compare our approach with BOW model and softmax classifier, support vector machine, naive Bayes, and cosine distance and observe that DBRNN-A provides a higher rank-10 average accuracy.

1 INTRODUCTION

In an usual process, the end user encounters a bug (also called an issue or a defect) while working on the system and reports the issue in a bug tracking system [8]. Fig 1 shows a sample screenshot of a bug reported in Google Chromium project (bug ID: 638277). The bug report usually contains a bug summary and a detailed description mentioning the steps to reproduce. Bugs with *fixed* status also contains the developer who fixed the bug and is called as the owner. The process of bug triaging consists of multiple steps where first step primarily involves assigning the bug to one of the developers who could potentially solve the bug. Thus, in the rest of this research bug triaging refers to the task of developer assignment for the bug report [1]. In large scale systems, with a huge amount of incoming bugs, manually analyzing and triaging a bug report is a laborious process. Manual bug triaging is usually performed using the bug report content, primarily consisting of the summary and description. While additional sources of input has been explored in the 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] [14] [27] [28] [29] [32] [33]. Using the bug report content, automated bug triaging can be formulated as a classification problem, mapping the bug title and description to one of the developers (class labels). However, the bug report content contains noisy text information including code snippets, and stack trace details, as observed in Fig. 1. Processing such unstructured and noisy text data is a major challenge in learning a classifier.

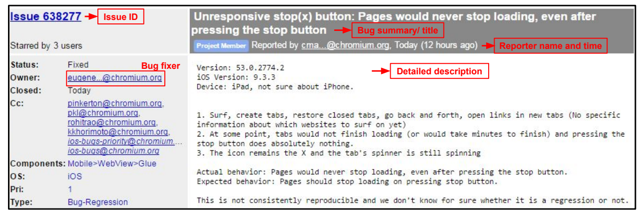

Figure 1: Screenshot of a bug report available in Google Chromium project, bug ID: 638277. The bug report usually consists of a brief summary and a detailed description at the time of reporting.

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 Google Chromium bug repository used as a labeled template for training the classifier.

Website: http://bugtriage.mybluemix.net/,
2018.

1.1 Motivating Example

Consider a labeled bug report example shown in Fig. 2. 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 this representation to the corresponding class label *brettw@chromium.org*. For the given two testing examples shown in Fig. 3, the actual fixer of first example, with bug id 634446, is *brettw@chromium.org* while the second example bug with id 616034 was fixed by *machenb...@chromium.org*. However, based on BOW features there are 12 words common between testing report#1 and the train report, while there are 21 words common between testing report#2 and the train report. Hence, a BOW model mis-classifies that the testing bug report#2 with id 616034 should be fixed by *brettw@chromium.org*. The reasons for the mis-classification are: (i) BOW feature model considers the sentence as a bag-of-words loosing the ordering (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 ordering, they suffer from high dimensionality and sparse data [12]. The semantic similarity between word tokens can be learnt using a skip-gram based neural network model called *word2vec* [22]. This model relies on distributional hypothesis which claims that words that appear in the same context in the sentence share a semantic meaning. Ye et al., [34] built a shared word representation using *word2vec* for word tokens present in code language and word tokens present in descriptive language. The main disadvantage of *word2vec* is that it learns a semantic representation of individual word tokens, however, does not consider a sequence of word tokens such as a sentence. An extension of *word2vec* called *paragraph vector* [19] considers the ordering of words, but only for a small context.

The rest of the paper is organized as follows: section 2 highlights the main research questions addressed in this research work and the key contributions, section 3 details the proposed approach including the deep learning algorithm and the classifier, section 4 talks about the experimental data collected in this research, section 5 discuss our experimental results and analysis, section 6 discusses some of the threats to validate our claims, section 7 talks about other applications that can be addressed using the proposed feature learning algorithm, section 8 explains about some closely related work and section 9 concludes our work with some future directions.

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 example bug reports from Google Chromium bug repository for which a suitable developer has to be predicted. By ground truth, bug report #1 was fixed by the same developer as in the training instance. However, the BOW feature of bug report #2 is more similar to the training instance than the bug report #1. The overlapping words with the training bug are highlighted.

2 RESEARCH CONTRIBUTIONS

Learning semantic representation from large pieces of text (such as in description of 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 a bug report content in an unsupervised fashion ie., the representation will be learnt directly using the data without the need for manual feature engineering. The main research questions (RQ) that we are trying address in this research paper 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 approaches?
(3) **RQ3**: Is there an effect on the number of training samples per class on the performance of the classifier?
(4) **RQ4**: What is the effect of using only the title (or summary) of the bug report in performing triaging compared with the using the description as well ?
(5) **RQ5**: Is transfer learning effective using deep learning, where the deep learning model is trained using one dataset and used to perform triaging in another dataset?

Recently, recurrent neural network (RNN) based deep learning algorithms have revolutionized the concept of word sequence representation and have shown promising breakthroughs in many applications such as language modeling and machine translation. Lam et al. [17] used deep neural network (DNN) with rSVM to learn a common representation between source code and the bug reports and used it for effective bug localization. White et al., [30] provided a broad perspective on how deep learning can be used in software repositories to solve some challenging problems. The main contributions of this research are summarized as follows:

- A novel bug report representation approach is proposed using DBRNN-A: Deep Bidirectional Recurrent Neural Network with Attention mechanism and with Long Short-Term Memory units (LSTM) [24]. The proposed deep algorithm algorithm is capable of remembering the context over a long sequence of words.

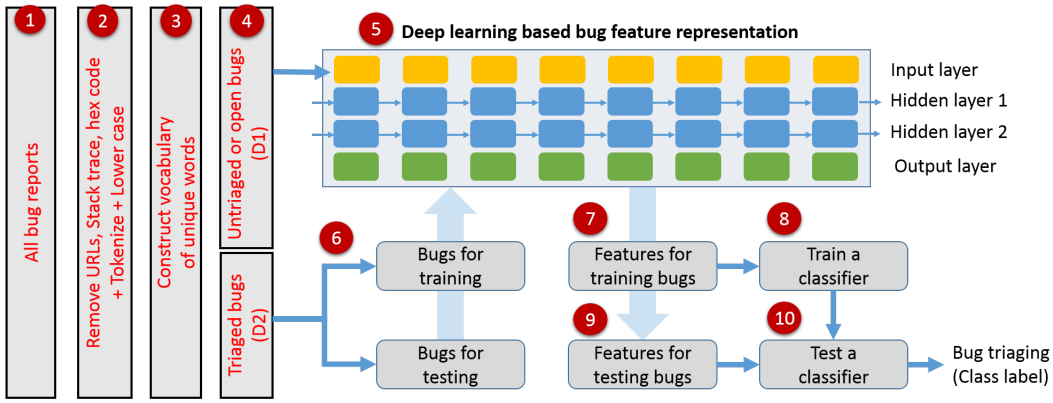

Figure 4: The flow diagram of the overall proposed algorithm highlighting the important steps.

- The untriaged and unsolved bug reports constitute about 70% in an open source bug repository and are usually ignored in the literature [14]. In this research, we provide a mechanism to leverage all the untriaged bugs to learn 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 the classifiers trained on different train-test splits of datasets [14] [18] are neither comparable nor reproducible. Thus, to enable our research reproducible, the entire dataset along with the exact train test split and the source code of our approach are made publicly available for research purpose[1].
- We further study the effectiveness of the proposed bug training in a cross-data testing scenario (transfer learning). By training the model with bugs from Chromium project and re-using the model for triaging bugs in Core and Firefox projects (Mozilla bug repository), the transfer learning ability of the deep learning model is showcased.

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 (title) and the bug 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 specialized training of the deep learning model, and hence in this research work, those contents are removed in the preprocessing stage,
(3) a set of unique words that occurred for at least k-times in the corpus is extracted as the vocabulary,
(4) the triaged bugs (D2) are used for classifier training and test, while all the untriaged/ open bugs (D1) are used to learn a deep learning model,
(5) a deep bidirectional recurrent neural network with attention mechanism technique learns a bug representation considering the combined bug title and description as a sequence of word tokens,
(6) the triaged bugs (D2) are split into train and test data with a 10 fold cross validation to remove training bias,
(7) feature representation for the training bug reports are extracted using the learnt 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 the learnt deep learning algorithm,
(10) using the extracted features and the learnt classifier, a probability score for every potential developer is predicted and the accuracy is computed in the test set.

The proposed approach varies with the traditional pipeline for automated bug triaging in (i) step 4 where the untriaged bugs (D1) are completely ignored and (ii) the deep learning based bug report representation instead of a bag-of-words representation. Addition of steps 4 and 5, enables to automatically learn a bug report representation from the data itself instead of manual engineering.

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

The proposed DBRNN-A based feature learning approach, as shown in Fig. 5, have the following key advantages:

- DBRNN-A can learn sentence representation preserving the order and syntax of words, as well as, retaining the semantic relationship. Long short-term memory (LSTM) cells [13] are used in the hidden layer which have a memory unit that can remember longer word sequences and also can solve the vanishing gradient problem [25].
- Intuitively, all the words in the content may not be useful in triaging the bug. To incorporate this, we introduce an attention mechanism [20] which learns to "attend" to only the important words in the bug report during classification.

[1]Made available at: http://bugtriage.mybluemix.net/

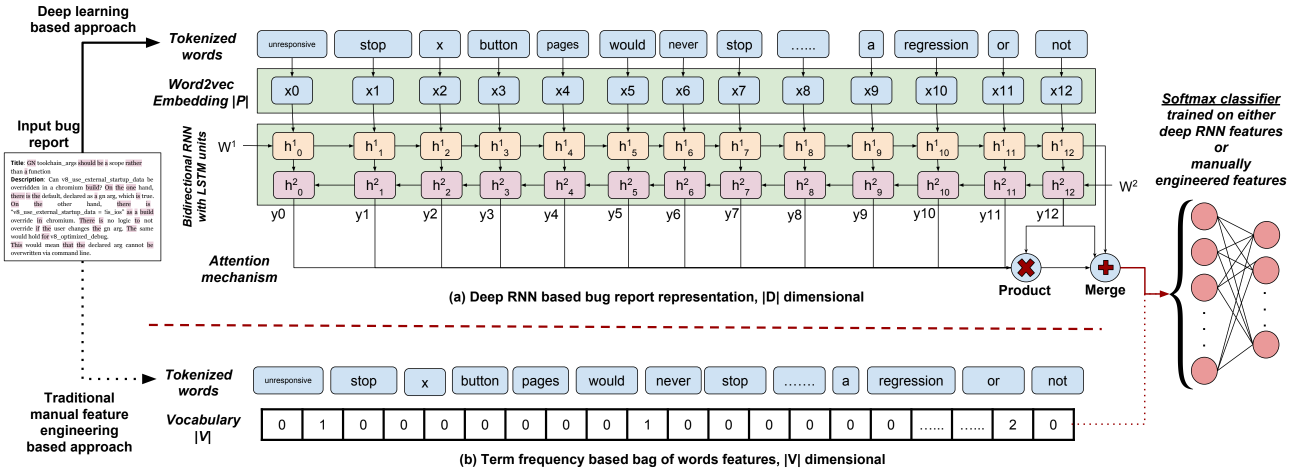

Figure 5: Detailed explanation of the working of a deep bidirectional Recurrent Neural Network (RNN) with LSTM units for an example bug report shown in Fig. 1. It can be seen that the deep network has multiple hidden layers, learning a complex hierarchical representation from the input data. As a comparison, tf based bag-of-words (BOW) representation for the same example sentence is also shown.

As the attention mechanism chooses only a few words during classification, the DBRNN-A can learn context representation from really long word sequences.

- A bidirectional RNN [9] considers the word sequence both in forward direction (first word to last word) and in backward direction (last word to first word) and merges both these representations. Thus, a context of a particular word includes both the previous few words and following few words making the representation more robust.

For each word, a one-hot $|V|$-dimensional representation is extracted using the vocabulary [2], over which a a $|P|$-dimensional *word2vec* representation [23] is learnt. As shown in Fig. 5 (a), a DBRNN-A with LSTM units is learnt over this word representation, to obtain a $|D|$-dimensional feature representation of the entire bug report (title + description). RNN is a sequence network containing 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 representation, the ground truth y_i are the next word in the sequence x_{i+1}, that is, upon memorizing the previous words in the sentence the network tries to predict the next word. LSTM function [9] have special purpose built-in memory units 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 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 model are learnt, 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 [23] 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

[2]http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

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 different bug repositories, Google Chromium, Mozilla Core, and Mozilla Firefox used in our experiments.

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 library. 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 used those features 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] [32]. 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 softmax classifier, a popular choice of classifier along with deep learning [9] [7] [10]. Softmax classifier is a generalization of logistic regression for multiclass 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 the i-th bug report belonging to the class.

4 LARGE SCALE PUBLIC BUG TRIAGE DATASET

A huge corpus of bug report data is obtained from three popular open source system : 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, the bug owner, and the 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*, 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 a bug 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 are used for deep learning, and 118,643 (31%) bugs are used for the classifier training and testing.

Data from two popular components from 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" 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, such as, in Core (7219/135434 = 5.33%) and in Firefox (3716/27930 = 13.3%). 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, the title and description text content of the bug are combined. 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 [34]. 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 Deep Learning

In our data split mechanism, the classifier testing data is unseen data and hence cannot be used for the deep learning algorithm. A design choice was taken for not using the classifier training data for training the deep learning model, 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 deep learning model. Also, using a non-overlapping dataset for training the feature model and training the classifier model highlights the generalization ability of the 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 previous folds for training.

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

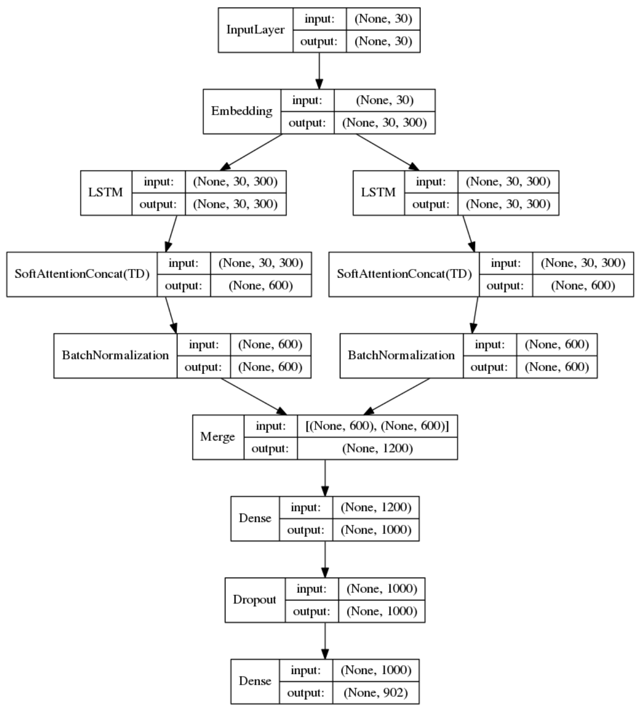

Figure 6: The architecture of DBRNN-A detailing all the parameters of the model.

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 number of training sample per developer. In a recent study, Jonsson et al., [14] 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 the literature [2] [5], it is clear that the threshold parameter affect 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.

5 EXPERIMENTAL EVALUATION

5.1 Evaluation Protocol and Metric

For a given bug report, the trained softmax 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 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 [16].

For learning the deep representation, a DBRNN-A is constructed having 300 LSTM units and the dropout probability is 0.3. A categorical cross entropy based 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 source code implementation 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 in the same dataset. Thus, we implement some of the successful approaches for automated bug triaging from the literature [2] [32] [14] and compare it with our algorithm using our benchmark dataset. Term frequency based BOW is used to represent the combined 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 [26], (ii) Support Vector Machine (SVM) [31], (iii) Multinomial Naive Bayes (MNB) [15], and (iv) Cosine distance based matching [21]. The four supervised classifiers are implemented using the Python *scikit-learn*[7] package. All these four classifiers use only the classifier training and testing data and do not use the untriaged bug reports.

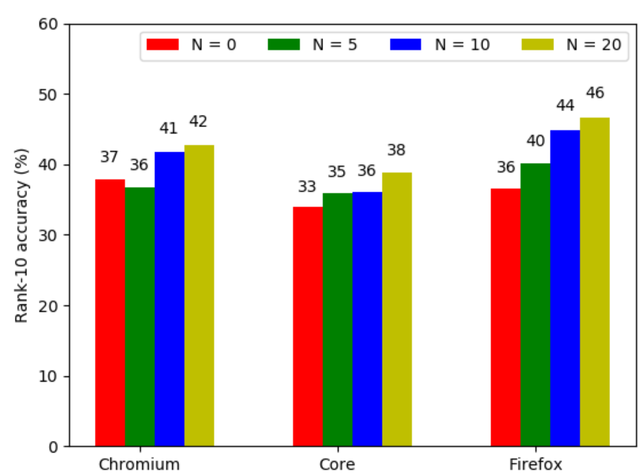

Figure 7: The rank-10 average accuracy of the deep learning algorithm on all three datasets. It can be observed that as the number of training samples per class increases, the overall triaging accuracy increases, addressing RQ3.

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

Threshold	Classifier	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
Min. train samples per class = 0	BOW + MNB	21.9	25.0	26.0	23.0	23.7	25.9	26.3	26.1	28.7	33.3	26.0 ± 3.0
	BOW + Cosine	18.4	20.1	21.3	17.8	20.0	20.6	20.4	20.9	21.1	21.5	20.2 ± 1.2
	BOW + SVM	11.2	09.3	09.5	09.5	09.4	10.1	10.4	09.9	10.5	10.8	10.1 ± 0.6
	BOW + Softmax	12.5	08.5	08.6	08.7	08.6	08.5	09.1	08.9	08.7	08.7	09.1 ± 1.1
	DBRNN-A + Softmax	34.9	36.0	39.6	35.1	36.2	39.5	39.2	39.1	39.4	39.7	**37.9 ± 1.9**
Min. train samples per class = 5	BOW + MNB	22.2	25.2	26.1	23.1	23.8	26.0	26.5	26.3	29.2	33.6	26.2 ± 3.1
	BOW + Cosine	18.6	20.2	21.4	18.2	19.1	20.7	21.1	21.0	21.6	22.0	20.4 ± 1.3
	BOW + SVM	11.3	11.1	08.1	08.3	09.2	09.0	08.9	08.7	08.5	09.0	09.2 ± 1.0
	BOW + Softmax	12.8	11.1	11.1	09.3	11.1	09.8	10.4	10.5	10.9	11.4	10.8 ± 0.9
	DBRNN-A + Softmax	32.2	33.2	37.0	36.4	37.1	37.2	38.3	39.0	39.1	38.2	**36.8 ± 2.2**
Min. train samples per class = 10	BOW + MNB	22.4	25.5	26.4	23.3	24.1	26.5	26.8	27.0	30.1	34.3	26.6 ± 3.3
	BOW + Cosine	18.8	20.5	21.7	18.5	19.6	21.2	21.4	21.1	21.8	21.0	20.6 ± 1.3
	BOW + SVM	12.2	11.4	11.8	11.6	11.5	11.3	11.8	11.0	12.1	11.9	11.7 ± 0.4
	BOW + Softmax	11.9	11.3	11.2	11.2	11.3	11.1	11.4	11.3	11.2	11.5	11.3 ± 0.2
	DBRNN-A + Softmax	36.2	37.1	40.45	42.2	41.2	41.3	44.0	44.3	45.3	46.0	**41.8 ± 3.1**
Min. train samples per class = 20	BOW + MNB	22.9	26.2	27.2	24.2	24.6	27.6	28.2	28.9	31.8	36.0	27.8 ± 3.7
	BOW + Cosine	19.3	20.9	22.2	19.4	20.0	22.3	22.3	22.9	23.1	23.0	21.5 ± 1.4
	BOW + SVM	12.2	12.0	11.9	11.9	11.6	11.5	11.3	11.6	11.6	11.9	11.7 ± 0.3
	BOW + Softmax	11.9	11.8	11.4	11.3	11.2	11.1	11.0	11.8	11.3	11.7	11.5 ± 0.3
	DBRNN-A + Softmax	36.7	37.4	41.1	42.5	41.8	42.6	44.7	46.8	46.5	47.0	**42.7 ± 3.5**

Table 2: Rank-10 accuracy obtained on the Google Chromium project across the ten cross validations. The average accuracy over the cross validation and standard deviation is also reported. The best performing results are shown in bold.

Threshold	Classifier	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
Min. train samples per class = 0	BOW + MNB	21.6	23.6	29.7	30.3	31.0	31.2	31.9	31.7	32.3	32.1	29.5 ± 3.6
	BOW + Cosine	16.3	17.4	19.5	21.3	22.5	23.2	24.0	25.5	27.5	29.1	22.6 ± 3.9
	BOW + SVM	13.6	14.6	14.9	14.0	12.1	12.9	11.7	13.7	14.4	14.1	13.6 ± 1.0
	BOW + Softmax	14.3	11.8	9.5	10.0	9.2	10.4	10.5	10.6	11.0	10.8	10.8 ± 1.4
	DBRNN-A + Softmax	30.1	31.7	35.2	33.0	34.1	35.9	34.8	34.2	34.6	35.1	**33.9 ± 1.7**
Min. train samples per class = 5	BOW + MNB	20.7	23.8	29.7	31.4	31.7	33.8	35.6	36.7	35.8	36.2	31.5 ± 5.2
	BOW + Cosine	15.7	17.7	19.9	21.4	22.8	24.7	26.4	27.5	29.4	29.9	23.5 ± 4.6
	BOW + SVM	16.4	12.9	11.5	10.4	13.4	13.8	12.7	12.0	12.8	13.1	12.9 ± 1.5
	BOW + Softmax	14.9	13.5	12.5	10.6	11.4	12.8	12.1	13.3	12.4	14.0	12.7 ± 1.2
	DBRNN-A + Softmax	33.8	31.5	35.8	35.3	34.7	36.8	37.1	38.4	37.7	38.0	**35.9 ± 2.1**
Min. train samples per class = 10	BOW + MNB	18.4	23.9	29.8	33.4	36.7	39.4	38.5	40.8	41.3	42.5	34.5 ± 7.7
	BOW + Cosine	16.0	18.0	20.0	21.4	22.7	25.7	27.8	30.4	33.1	35.5	25.1 ± 6.2
	BOW + SVM	17.5	15.6	16.5	16.4	16.4	17.0	17.2	17.4	16.9	16.2	16.7 ± 0.6
	BOW + Softmax	15.6	14.2	14.4	13.9	14.0	13.4	13.8	14.5	14.9	14.1	14.3 ± 0.6
	DBRNN-A + Softmax	32.5	33.7	35.5	36.5	36.4	34.4	36.1	37.3	38.9	39.6	**36.1 ± 2.1**
Min. train samples per class = 20	BOW + MNB	21.3	24.3	30.2	34.8	38.5	39.4	37.5	40.7	42.1	41.8	35.1 ± 7.0
	BOW + Cosine	16.8	18.4	20.4	23.3	28.6	31.3	35.7	38.6	37.3	38.9	28.9 ± 8.2
	BOW + SVM	14.6	15.2	16.4	14.5	13.9	15.7	16.8	15.6	16.1	16.4	15.5 ± 0.9
	BOW + Softmax	18.8	16.4	11.4	10.5	11.8	13.1	13.6	14.3	14.8	15.3	14.0 ± 2.4
	DBRNN-A + Softmax	33.3	34.9	36.5	36.8	37.7	39.0	41.3	42.6	41.1	43.3	**38.8 ± 3.2**

Table 3: Rank-10 accuracy obtained on the Mozilla Core project across the ten cross validations. The average accuracy over the cross validation and standard deviation is also reported. The best performing results are shown in bold.

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.

RQ1: Is it feasible to automated perform bug triaging using deep learning?

Threshold	Classifier	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
Min. train samples per class = 0	BOW + MNB	19.1	21.3	24.5	22.9	25.8	28.1	30.3	31.9	33.94	35.55	27.4 ± 5.2
	BOW + Cosine	17.3	20.3	22.9	25.4	26.9	28.3	29.8	27.5	28.9	30.1	25.7 ± 4.1
	BOW + SVM	13.4	11.4	13.8	15.5	14.5	14.5	14.3	14.4	14.6	14.6	14.1 ± 1.0
	BOW + Softmax	11.9	17.8	17.8	15.7	13.6	15.5	13.7	13.1	13.1	13.6	14.6 ± 1.9
	DBRNN-A + Softmax	33.6	34.2	34.7	36.1	38.0	37.3	38.9	36.3	37.4	38.1	**36.5 ± 1.7**
Min. train samples per class = 5	BOW + MNB	21.1	26.8	31.1	33.4	36.5	36.0	37.6	36.9	34.9	36.5	33.1 ± 5.1
	BOW + Cosine	20.8	23.0	23.7	26.2	27.4	29.2	32.3	32.7	34.1	35.2	28.5 ± 4.8
	BOW + SVM	14.4	16.0	17.8	17.8	17.8	16.3	16.7	16.7	16.7	15.2	16.5 ± 1.1
	BOW + Softmax	18.2	14.8	16.7	16.7	15.4	14.5	12.5	12.9	12.9	13.7	14.8 ± 1.8
	DBRNN-A + Softmax	27.6	34.9	37.9	38.7	40.1	42.3	45.2	44.9	45.0	44.5	**40.1 ± 5.3**
Min. train samples per class = 10	BOW + MNB	21.7	27.6	32.1	34.8	37.7	34.6	32.6	34.7	36.7	38.5	33.1 ± 4.8
	BOW + Cosine	18.1	21.2	24.4	27.0	28.3	30.1	32.3	34.00	35.4	36.6	28.7 ± 5.8
	BOW + SVM	09.9	09.9	11.8	11.8	11.8	12.8	12.9	12.9	12.9	12.8	11.9 ± 1.1
	BOW + Softmax	14.3	15.6	12.1	09.5	09.5	11.2	12.0	12.6	12.1	12.7	12.1 ± 1.8
	DBRNN-A + Softmax	35.1	36.4	40.5	42.5	45.4	47.4	48.9	49.1	51.1	51.4	**44.8 ± 5.6**
Min. train samples per class = 20	BOW + MNB	22.0	22.8	23.6	26.3	29.2	32.3	34.4	36.4	38.6	38.4	30.4 ± 6.2
	BOW + Cosine	18.4	21.9	25.1	27.5	29.1	31.4	33.8	35.9	36.7	38.3	29.8 ± 6.3
	BOW + SVM	18.7	16.9	15.4	18.2	20.6	19.1	20.3	21.8	22.7	21.9	19.6 ± 2.2
	BOW + Softmax	16.5	13.3	13.2	13.8	11.6	12.1	12.3	12.3	12.5	12.9	13.1 ± 1.3
	DBRNN-A + Softmax	38.9	37.4	39.5	43.9	45.0	47.1	50.5	53.3	54.3	55.8	**46.6 ± 6.4**

Table 4: Rank-10 accuracy obtained on the Mozilla Firefox project across the ten cross validations. The average accuracy over the cross validation and standard deviation is also reported. The best performing results are shown in bold.

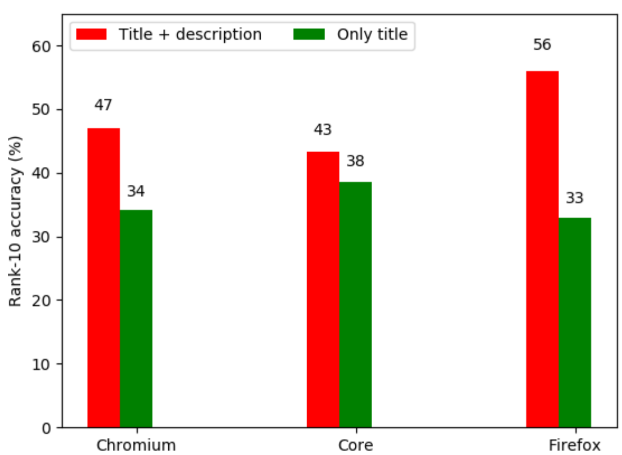

Figure 8: The rank-10 average accuracy of the deep learning algorithm on all three datasets by using only title or title along with the description in bug report. Discarding the description reduces the performance significantly, addressing RQ4.

From the obtained results, it can be clearly observed that the deep learning the representation of a bug report is a feasible and potentially competent approach for bug triaging. The proposed DBRNN-A approach provided rank-10 triaging accuracy in the range of $34-47\%$. 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 representation 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 time 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 the speed of the proposed approach.

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 learning features because the deep learning model could memorize the context from longer sentences in a bug report reasoning for the large improvement in performance. From the results, we observe that for BOW features MNB and cosine distance based matching outperforms SVM and softmax classifier. Although SVM is a popular choice of a supervised classifier, for real numbered sparse features in BOW model, feature independence which is assumed both in MNB and cosine distance matching proves successful.

RQ3: Is there an effect on the number of training samples per class on the performance of the classifier?
Both intuitively and experimentally, we find that as the minimum number of training samples per class increased, the performance of the classification improved across all the bug repositories by learning better classification boundary. For instance in Chromium dataset, when a classifier is trained with threshold as zero, 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 dataset. 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 DB-RNN increases. Despite the fact that there is 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 is the performance. Also, across the cross validations there is about $(2-7)\%$ standard deviation in all dataset. 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 (or summary) of the bug report in performing triaging compared with the using the description as well ?
The performance of the deep learning model 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 using the description content. It can be clearly observed that discarding description significantly reduces the performance of triaging of up to 23% in Firefox dataset.
RQ5: Is transfer or cross-data learning effective using deep learning, where the deep learning model is trained using one dataset and used to perform triaging in another dataset?
Transfer learning reduces the offline training time significantly by re-using a model trained using another dataset. However, most of the models fail while transferring the learnt model across datasets. The effectiveness of the deep learning model in transfer learning is studied, by training the model in Chromium dataset and testing in Core and Firefox datasets (Mozilla dataset). Using the deep learning model trained on Chromium dataset, the average rank-10 accuracy obtained when N=20 are 42.7% for Chromium test set, 39.6% for Core test set, and 43% on Firefox test set. The obtained results are comparable with the results obtained by training and testing on the same dataset. This shows that the proposed approach is capable of using a model trained on dataset to triage bug reports in another dataset, effectively.

6 THREATS TO VALIDITY

There are certain threats to establish the validity of the proposed results. While the common threats to any learning based classification system are applicable, few of these threats are specific to the advocated problem, as follows:

(1) The results are shown using three open source bug repositories with different characteristics to ensure generalization. However, commercial bug tracking systems may follow different patterns and hence our results may not be directly extensible to such repositories.
(2) Currently in our approach, only the bug report title and description are considered. While the experimental results show that these two unstructured text data are necessary, there could be other added information required to sufficiently triage a bug report.
(3) For comparison purpose, we re-implemented some of the successful algorithms in the literature for bug triaging, as there is no publicly implementation available. Although we have implemented the algorithms true to the best of our understanding, there could be some minor deviations from the original implementation.
(4) Both during training and testing of a classifier, we assumed only one developer as the rightful owner of a bug report. However, based on patterns and history of bug reports that are solved, there could be more than one active developer in the project who could potentially address the bug.

7 OTHER APPLICATIONS

The bug representation is learnt directly from the data in an unsupervised fashion and the features are task independent. This gives us a flexibility to use these features to learn a supervised classifier for any task or application. We have discussed a few of other possible applications for the proposed feature representation.

- Which bug gets fixed: It is a challenging research problem that have been addressed in literature [11]. Using the same feature representation extracted in this research, a supervised or semi-supervised binary classifier can be trained to classify fixed bugs with non-fixed bugs.
- Bug-fix time prediction: As Bhattacharya et al. [6] discuss, a predictive model can be constructed using the proposed features to learn the time required to fix the bug.
- Reopen analysis: It provides an interesting insight from the maintenance perspective to study which bugs get reopened during its lifecycle. As discussed by Zimmermann et al. [35], characterizing these bugs and predicting them can be performed using the deep learning features.
- Bug priority estimation: Priority of the bug is to be estimated before triaging happens [6]. Based on the priority, the SLA clock for the bug and the developer to be assigned might change. A 5-point scale priority can be formulated as a five class classification and using the learnt features, a supervised classifier can be learnt.

8 RELATED WORK

Table 5 presents a list of closely related works on bug triaging arranged in a chronological order (year 2010 to 2016). A majority of previous techniques have used bug summary/title and description [2] [28] [32] [33] 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. [27] use code information for improved perfomance. Badashian

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 [28]	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 [33]	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 [27]	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 [29]	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 [32]	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 [14]	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.

et. al. [3] identify developers' expertise using stack overflow and keywords from bug description.

From table 5, we observe that many different feature models such as tf, normalized tf, tf-idf, and n-grams have been employed. Choosing which feature model to use is an engineering design choice and it is challenging to choose which feature model will best represent the collected data. In this research, we address this challenge and design a deep bidirectional RNN which directly learns the best feature representation from the data in an unsupervised fashion. Further, we move beyond a word level representation model and propose a sequence of word representation model, to learn a unified representation for the entire bug report.

9 CONCLUSION

In this research we proposed a novel software bug report (title + description) representation algorithm using deep bidirectional Recurrent Neural Network with attention (DBRNN-A). The proposed deep learning algorithm learns a paragraph level 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). Experimental results shows DBRNN-A along with softmax classifier outperforms the bag-of-words model, 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 learnt on the Chromium dataset competitively triaged the bugs in the Mozilla dataset. Additionally, the dataset along with its complete benchmarking protocol and the implemented source code is made publicly available to increase the reproducibility of this research.

REFERENCES

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

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

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

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

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

[6] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: can we do better? In *Working Conference on Mining Software Repositories*, pages 207–210, 2011.
[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
[8] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, pages 23–32, 2003.
[9] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.
[10] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.
[11] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *International Conference on Software Engineering*, volume 1, pages 495–504, 2010.
[12] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, pages 837–847, 2012.
[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
[14] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.
[15] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence*, pages 488–499. Springer, 2004.
[16] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 1137–1145, 1995.
[17] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *International Conference on Automated Software Engineering*, pages 476–481, 2015.
[18] Ahmed Lamkanfi, Javier Perez, and Serge Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *Working Conference on Mining Software Repositories*, 2013.
[19] Quoc V Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, volume 14, pages 1188–1196, 2014.
[20] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.
[21] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, volume 6, pages 775–780, 2006.
[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.
[24] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In *International Conference on Frontiers in Handwriting Recognition*, pages 285–290, 2014.
[25] Hasim Sak, Andrew W Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342, 2014.
[26] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *arXiv preprint arXiv:1309.2388*, 2013.
[27] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Working Conference on Mining Software Repositories*, pages 2–11, 2013.
[28] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Fuzzy set-based automatic bug triaging: Nier track. In *International Conference on Software Engineering*, pages 884–887, 2011.
[29] Song Wang, Wen Zhang, and Qing Wang. Fixercache: unsupervised caching active developers for diverse bug triage. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 25, 2014.
[30] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Working Conference on Mining Software Repositories*, pages 334–345, 2015.
[31] Ting-Fan Wu, Chih-Jen Lin, and Ruby C Weng. Probability estimates for multi-class classification by pairwise coupling. *The Journal of Machine Learning Research*, 5:975–1005, 2004.
[32] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. Towards effective bug triage with software data reduction techniques. *IEEE Transactions on Knowledge and Data Engineering*, 27(1):264–280, 2015.
[33] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. Developer prioritization in bug repositories. In *International Conference on Software Engineering*, pages 25–35, 2012.
[34] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *International Conference on Software Engineering*, pages 404–415, 2016.
[35] Thomas Zimmermann, Nachiappan Nagappan, Philip J Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *International Conference on Software Engineering*, pages 1074–1083, 2012.