CS559 Deep Learning Project Final Report: Automated Bug Triaging

Emre Dogan
Department of Computer Engineering
Bilkent University
Ankara, Turkey
emre.dogan@bilkent.edu.tr

H. Alperen Cetin
Department of Computer Engineering
Bilkent University
Ankara, Turkey
alperen.cetin@bilkent.edu.tr

***Abstract*—This is the final report of CS559 Deep Learning Course Term Project. In this project, we implemented an automated bug triaging method, DeepTriage. Bug triaging is the process of prioritizing bugs and assigning an appropriate developer for a given bug. The main task in our project is to predict the most appropriate developer to fix a software bug from the a given bug report. This problem can be defined as a classification problem in which textual bug attributes (bug title, description etc.) are inputs and the available developer (class label) is the output. We employ Deep Bidirectional Recurrent Neural Network with Attention(DBRNN-A) approach for this classification task.**

To improve the performance of the model, three contributions are made to the original implementation: (1) Using GRU instead of LSTM to fasten the training process, (2) Combining different datasets to create a more generalized model, (3) Adding extra dense layers before the multiclass classification. From these experiments, we achieved the state of art results in *Mozilla Firefox* dataset, an accuracy of 46.6%. In the *Chromium* dataset, we get a higher accuracy(44.0%) than the original accuracy from the paper(42.7%)

***Index Terms*—recurrent neural networks, long short-term memory, gated-recurrent-unit, bug triaging, transfer learning.**

I. Introduction

It is an important task to assign an appropriate developer for a given bug report from both developer's and organization's perspectives. A significant amount of time is spent by software developers to understand, identify and fix the bug. A poor developer assignment for a bug report might reduce his/her efficiency. On the other hand, from the organization's perspective, bug fixing time is important as the corresponding bug might block the working of a product/service and cost a large amount of money. For all these reasons, assigning the most appropriate developer for a bug report, one of the primary tasks of bug triaging process, is an important and active research area.

This paper is organized as follows. Section II gives a brief information on the background. Section III gives details about the dataset. Section IV illustrates our approach. In Section V, we give the implementation details. In Section VI,our contributions are discussed. Finally, section VII and VIII states final results and conclusion.

Fig. 1. Overall process of bug triaging.

II. Background

A. Bug Triage

The term of *bug triage* is the process of going through a list of bugs to find bugs that need assistance, escalation, or follow-up [1]. There are different types of bugs and each of them needs to be treated differently.

The goal of triaging is to evaluate, prioritize and assign the resolution of bugs to the developers. The overall process of bug triaging is available in Fig 1.

One of the most time consuming parts of this process is to find the best developer for a given bug. A critical bug might cause the company to lose a large amount of money. So, it is an important task to assign the bug to the best developer who can solve the bug in the shortest amount of time.

In the earlier times, this task was completed in a manual manner. A developer was responsible for assigning the bugs to the most suitable of other developers. This method is still applied in small companies. But as the size of developers and bugs increase, it becomes harder to find the best match. This problem arises the necessity of automating this process. In the following subsection, the methods of automated bug triaging will be discussed.

B. Automated Bug Triaging Studies

There are several studies on automating the triage process. Most of these studies propose a machine learning based approach. They train their model by the data collected from open source and proprietary software projects.

Cubranic et al. [5] and Anvik et al. [2] proposed a Naive Bayessian classifier approach to apply text classification on bug reports in order to predict the relevant developers. Jeong et al. [3] proposed a bug tossing graph approach based on Markov chains from the knowledge of reassigning. Xuan et

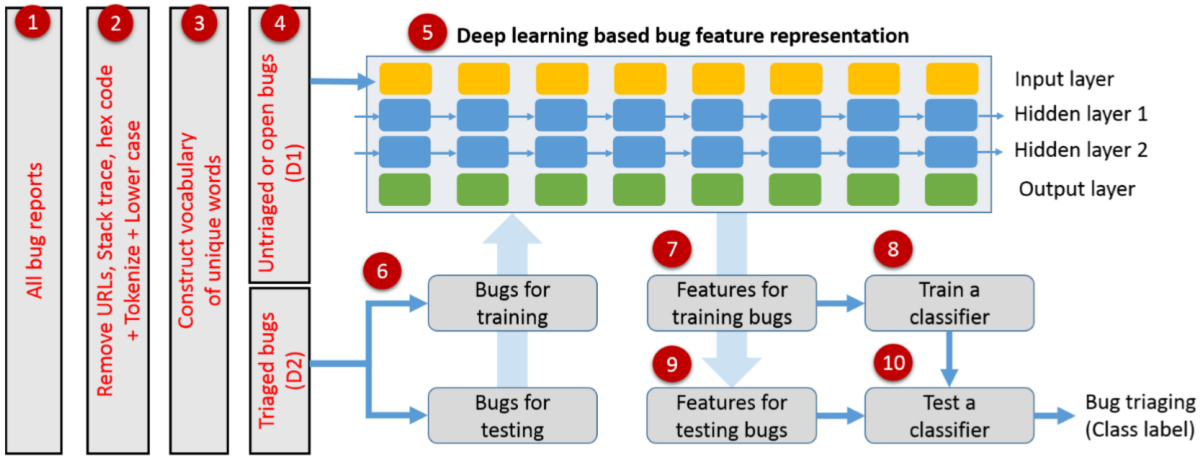

Fig. 2. The flow diagram of the overall proposed algorithm highlighting the important steps. [4]

al. [6] proposed a semi-supervised text classification model for bug triaging.

Different from the previous studies, Deep Triage [4] extracts a novel bug report representation from bug reports by using an Attention Based Deep Bidirectional Recurrent Neural Network. These extracted features are used as input to multi-class classifier in order to predict the best developer.

III. Dataset

Our dataset consists of bug reports from three open source systems: Google Chromium, Mozilla Core and Mozilla Firefox. The details of data collecting process are mentioned in the paper [4]. The authors provide the dataset available online. In total, there are 383,104 bug reports from Google Chromium, 314,388 bug reports from Mozilla Core, and 162,307 bug reports from Mozilla Firefox. All datasets have attributes *id, issue id, issue title, reported time, owner, description* and *status*. Chromium dataset has another attribute *type* and the other Mozilla datasets have another attribute *resolution*. In DBRNN-A model, only *owner, title, description* and *status* attributes are used. We shared dataset links and brief explanation about dataset contents in the GitHub Repository.[1]

IV. Approach

A. Preprocessing

The bug report datasets having title, description, reported time, status and owner are shared online by the authors in json format. Before using the text in the titles and the descriptions of the bug reports, some parts of them need to be removed.

First, URLs, stack traces and the hex codes are removed, and all characters are changed to lower case. After that, words are tokenized, then all punctuation and *None* words are deleted. The same process is followed for both open bug reports data and closed bug reports data. Second step in Fig. 2 is the preprocessing step.

[1]https://github.com/hacetin/deep-triage

B. Word Embedding

Simple approaches like bag-of-words and n-grams are not sufficient to represent the features of bug reports. The problem with bag-of-words approach is that it loses the ordering of words in the contextual manner and the semantic similarity between synonymous words. On the other hand, n-grams model offer a better bug report representation but it fails to deal with high dimensionality and sparse data. As these two representations are not good enough to represent a bug report, the authors come up with a new approach to represent bug reports, Word2Vec embedding, such that disadvantages of bag-of-words and n-gram representations will be prevented. At the end, the final vocabulary is created from a set of unique words that occurred for at least k-times in open bug reports data by using Word2Vec, and the owner information is acquired from the ”owner” label in the closed bug reports. Third step in Fig 2 is the word embedding step.

C. Model Description

Deep Bidirectional Recurrent Neural Network with Attention (DBRNN-A) [4] works in the following way:

First the model learns feature representation of bug reports from untriaged bugs in an unsupervised manner by using long-short term memory(LSTM) cells.

Then, attention mechanism is used, because all words in the content may not be useful in the classification task. Also, bidirectional RNN considers the words in both forward and backward direction, and concatenates both representations.

By training DBRNN-A model with triaged bugs with the embedding of untriaged bugs, the model learns extracting features. At the end, a softmax classifier is used for classification using these features.

Overall structure of the whole model can be seen in Fig. 2. Also, detailed structure of DBRNN-A is shown in Fig. 3.

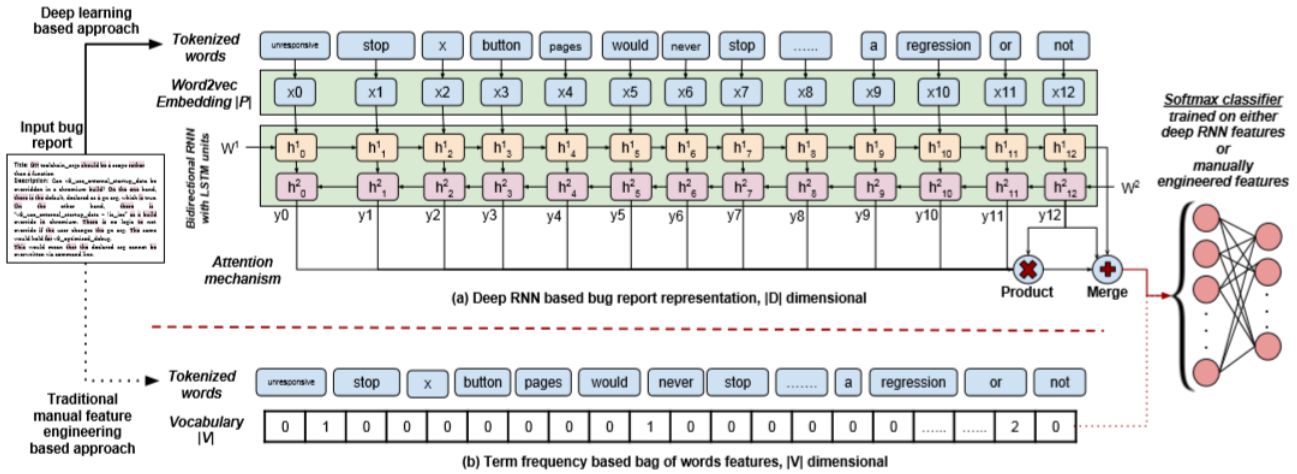

Fig. 3. Detailed explanation of the working of a deep bidirectional Recurrent Neural Network (RNN) with LSTM units. [4]

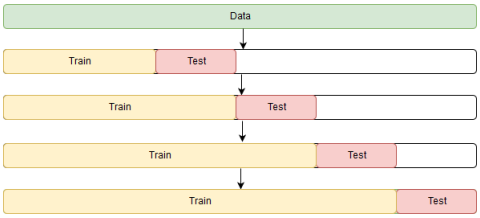

Fig. 4. Chronological cross validation

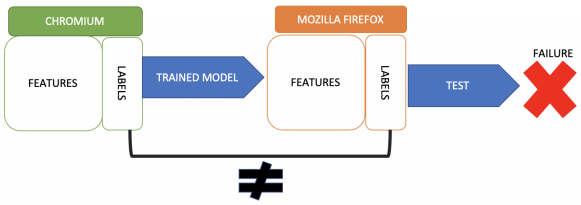

Fig. 5. Illustration of the problem with transfer learning between datasets.

V. Implementation Details

In preprocessing, unnecessary part deletions are handled by using standard Python RegEx library[2], and Stanford NLTK[3] 3.4 is used for word tokenization.

In word embedding, vocabulary is created by using Word2Vec from Gensim[4] 3.7.1.

All implementation of the model is done in Keras[5] 2.2.4 with the backend Tensorflow[6] 1.13.1. When it comes to the soft attention layer in DBRNN-A, we had problem with the soft attention layer[7] used in DeepTriage [4]. Then we implemented a similar soft attention layer with the help of a discussion[8] from StackOverflow.

All the work we have done is shared in our GitHub[9] repository.

VI. Improvements on Deep Triage

After achieving similar results with the paper [4], we tried some additions and changes to improve the DeepTriage method in the scope of this project. Because accuracy results for each chronological cross validation step are shared in the paper, we use the same validation method in our experiments to be able to compare our results. Fig. 4 shows the logic behind chronological cross validation.

A. Using GRU instead of LSTM

In the original study [4], the architecture consists of LSTM units. It is known that the GRU *(Gated Recurrent Unit)* can result with a faster training process as it has less parameters than the LSTM unit. The GRU unit can take advantage of all hidden states without any control, unlike the LSTM.

To observe the speeding factor of the GRU unit, the same architecture is implemented by GRU units. Although the chronological cross validation results are slightly different from the LSTM implementation, the final average result is exactly same. More detailed results are available in Table I.

B. Merging All Dataset Corpora

At the proposal stage, one of our proposals was to apply transfer learning between different datasets. For example, a model trained with Google Chromium dataset would be tested with Mozilla Firefox dataset. But the main problem with this approach is that class labels of different datasets are completely different as the developers of different projects are not same. The illustration of this issue is available Fig 5.

[2]https://docs.python.org/3/library/re.html
[3]http://www.nltk.org/index.html
[4]https://radimrehurek.com/project/gensim/
[5]https://keras.io/
[6]https://www.tensorflow.org/
[7]https://gist.github.com/braingineer/27c6f26755794f6544d83dec2dd27bbb
[8]https://stackoverflow.com/questions/42918446/how-to-add-an-attention-mechanism-in-keras
[9]https://github.com/hacetin/deep-triage

TABLE I
CHRONOLOGICAL CROSS VALIDATION TOP-10 ACCURACIES FOR EXPERIMENTS ON GOOGLE CHROMIUM DATASET WITH 20 MINIMUM TRAIN SAMPLES PER CLASS.

	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
DeepTriage [4]	36.7	37.4	41.1	42.5	41.8	42.6	44.7	46.8	46.5	47.0	42.7
Our LSTM Implementation	33.4	40.0	41.9	37.3	38.8	41.0	41.3	44.5	47.0	51.0	41.6
Our GRU Implementation	32.4	40.5	41.9	38.0	38.8	41.6	42.6	44.3	45.8	50.0	41.6
Our LSTM Implentation with merged corpus	33.7	41.3	44.3	39.1	40.3	43.1	43.2	47.0	48.9	53.0	43.4

TABLE II
CHRONOLOGICAL CROSS VALIDATION TOP-10 ACCURACIES FOR EXPERIMENTS ON MOZILLA FIREFOX DATASET WITH 20 MINIMUM TRAIN SAMPLES PER CLASS.

	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
DeepTriage [4]	38.9	37.4	39.5	43.9	45.0	47.1	50.5	53.3	54.3	55.8	46.6
Our Imp. with a single dense layer(1000)	38.7	37.6	45.0	43.4	43.0	39.2	50.5	49.0	48.2	46.4	44.1
Our Imp. with double dense layer(20y+10y)	37.5	39.6	48.1	43.9	44.6	40.0	52.3	49.8	47.1	46.7	44.9
Our Imp. with double dense layer(1000+1000)	38.7	36.3	44.4	43.5	44.9	37.3	51.7	51.5	49.6	49.7	44.8

TABLE III
CHRONOLOGICAL CROSS VALIDATION TOP-10 ACCURACIES FOR THE MODELS FROM THE PAPER AND OUR BEST MODEL ON MOZILLA FIREFOX DATASET WITH 20 MINIMUM TRAIN SAMPLES PER CLASS.

	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
BOW + MNB [4]	22.0	22.8	23.6	26.3	29.2	32.3	34.4	36.4	38.6	38.4	30.4
BOW + Cosine [4]	18.4	21.9	25.1	27.5	29.1	31.4	33.8	35.9	36.7	38.3	29.8
BOW + SVM [4]	18.7	16.9	15.4	18.2	20.6	19.1	20.3	21.8	22.7	21.9	19.6
BOW + Softmax [4]	16.5	13.3	13.2	13.8	11.6	12.1	12.3	12.3	12.5	12.9	13.1
DBRNN-A + Softmax [4]	38.9	37.4	39.5	43.9	45.0	47.1	50.5	53.5	54.3	55.8	46.6
Our Best Implementation	**39.1**	**34.0**	**46.4**	**46.1**	**49.2**	**44.1**	**54.1**	**52.2**	**51.7**	**48.9**	**46.6**

TABLE IV
CHRONOLOGICAL CROSS VALIDATION TOP-10 ACCURACIES FOR THE MODELS FROM THE PAPER AND OUR BEST MODEL ON GOOGLE CHROMIUM DATASET WITH 20 MINIMUM TRAIN SAMPLES PER CLASS.

	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
BOW + MNB [4]	22.9	26.2	27.2	24.2	24.6	27.6	28.2	28.9	31.8	36.0	27.8
BOW + Cosine [4]	19.3	20.9	22.2	19.4	20.0	22.3	22.3	22.9	23.1	23.0	21.5
BOW + SVM [4]	12.2	12.0	11.9	11.9	11.6	11.5	11.3	11.6	11.6	11.6	11.7
BOW + Softmax [4]	11.9	11.8	11.4	11.3	11.2	11.1	11.0	11.8	11.3	11.7	11.5
DBRNN-A + Softmax [4]	36.7	37.4	41.1	42.5	41.8	42.6	44.7	46.8	46.5	47.0	42.7
Our Best Implementation	**34.7**	**42.4**	**43.9**	**39.8**	**40.5**	**43.9**	**44.1**	**47.1**	**49.5**	**53.6**	**44.0**

Instead of applying transfer learning between datasets, a different approach is tried as a contribution. Instead of using only one dataset, we created the *word2vec* model by using three different datasets: *Chromium, Mozilla Core and Mozilla Firefox*. The results of the model trained with a combined corpus is given in Table I. It is remarkable that the accuracy results are enhanced by a factor of 2% when training with the combined corpus.

C. Changing Dense Layer Structure

Our datasets are created from very large open-source projects. These projects consist of a large number of developers(i.e. 1031 developers for Chromium). So, the number of class labels is large for our classification task.

In the paper implementation, the authors use a single dense layer of size 1000 before softmax classifier. When considering the large number of class labels, it may be considered that a single dense layer cannot be enough to represent the features of all labels. As a solution, we propose 2 different setups:

- *Double dense layers of size: 20y + 10y (where y is equal to the number of class labels)*
- *Double dense layers of size: 1000 + 1000*

By increasing the number of dense layers and the size of these layers, we achieved slightly better accuracy results compared with the paper implementation. Result accuracy values for different dense layer configurations can be seen in Table II.

VII. FINAL RESULTS

After our experiments, we decided to implement a model with GRU as RNN units, merged corpus and two dense layers with 1000 nodes. In our best model, we use the hyperparameters given in Table V. Figure 6 shows train and validation loss for CV#10 of Mozilla Firefox dataset with 20 minimum train samples per class. Validation accuracy for CV#10 is 48.9%, accuracy values for other CVs are given in Table III.

To evaluate our model with a completely different test data, we used Mozilla Firefox and Google Chromium datasets with 20 minimum train samples per class. We split them into three

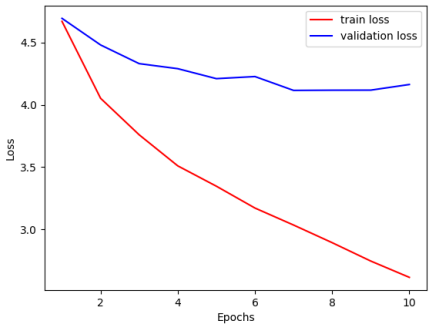

Fig. 6. Loss values for CV#10 of Mozilla Firefox dataset with 20 minimum train samples per class. Accuracy values are given in Table III.

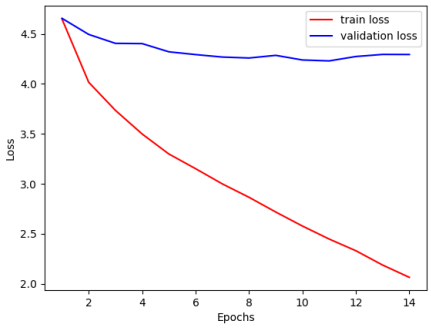

Fig. 7. Loss values of Mozilla Firefox dataset with 20 minimum train samples per class from Table for 82% train, 9% validation and 9% test split. **Top-10 test accuracy is 42.2%**

partitions: 82% train, 9% validation and 9% test. Loss graphics of Mozilla Firefox and Google Chromium are in Figure 7 and Figure 8 respectively. Top-10 test accuracy of Mozilla Firefox dataset is **42.2%**, and top-10 accuracy of Google Chromium dataset is **50.5%**.

VIII. Conclusion

In this project, we implemented the automated bug triaging model proposed by Mani et al. [4] and explored this study to enhance the results. Consequently, we resulted with 3 contributions for this study:

- *Swapping LSTM Units with GRUs:*
  - As the GRU unit keeps less parameters, the training process is fastened significantly without a loss in the accuracy results.
- *Combining different datasets to create the corpus:*
  - At the beginning of the semester, our purpose was to apply transfer learning between different datasets. But it is impossible to train the model with a dataset and test on another dataset. Because each software project consists of different developers and it is not possible to transfer the knowledge learnt from a dataset (mappings to developers) to another dataset.
  - Instead, we created a combined corpus from different datasets in order to take advantage of bug reports from different projects. The models trained by the combined corpus achieve a better accuracy with 2% increase.
- *Changing Dense Layer Structure:*
  - To represent more than 1000 class labels, we increased the number of dense layers and nodes within these layers. With this change, we achieved slightly better results.

Consequently, this study made us gain a deep understanding of RNN structures and ability of working with text data in deep learning models. We introduce 3 new contributions to the study and achieve slightly better results than the state of art.

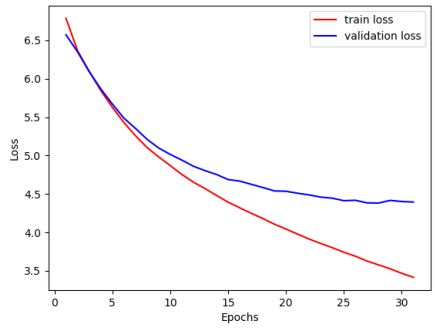

Fig. 8. Loss values of Google Chromium dataset with 20 minimum train samples per class from Table for 82% train, 9% validation and 9% test split. **Top-10 test accuracy is 50.5%**

TABLE V
Hyperparameters for our final model. Only exception is that we use 32 as batch size for Mozilla Firefox dataset.

Learning Rate for Adam Optimizer	0.0001
Patience for Early Stopping	3
Batch Size	1024
Number of RNN Units	1024
Max Sentence Length	50
Embedding size for Word2Vec	200
Minimum Word Frequency for Word2Vec	5
Context Window for Word2Vec	5

References

[1] Bug triage. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Helping_the_DOM_team/Bug_Triage.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[3] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

[4] S. Mani, A. Sankaran, and R. Aralikatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, pages 171–179. ACM, 2019.

[5] G. Murphy and D. Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[6] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo. Automatic bug triage using semi-supervised text classification. *arXiv preprint arXiv:1704.04769*, 2017.