Avrupa Bilim ve Teknoloji Dergisi
Sayı 21, S. 268-274, Ocak 2021
© Telif hakkı EJOSAT'a aittir
Araştırma Makalesi

www.ejosat.com ISSN:2148-2683

European Journal of Science and Technology
No. 21, pp. 268-274, January 2021
Copyright © 2021 EJOSAT
Research Article

An Automated Bug Triaging Approach using Deep Learning: A Replication Study

Eray Tüzün[*], Emre Doğan[2], Alperen Çetin[3]

[1*] Bilkent Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, Ankara Türkiye (ORCID: 0000-0002-5550-7816)), eraytuzun@cs.bilkent.edu.tr
[2] Bilkent Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, Ankara, Türkiye (ORCID: 0000-0002-2558-7624), emre.dogan@bilkent.edu.tr
[3] Bilkent Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, Ankara, Türkiye (ORCID: 0000-0001-9879-8599), alperen.cetin@bilkent.edu.tr

(First received 30 September 2020 and in final form 12 January 2021)

(**DOI:** 10.31590/ejosat.781341)

ATIF/REFERENCE: Tüzün E., Doğan E., Çetin A. (2021). An Automated Bug Triaging Approach using Deep Learning: A Replication Study. *European Journal of Science and Technology*, (21), 268-274.

Abstract

Bug management is the process to identify and fix bugs. In the bug management process, after a bug is identified, it needs to triaged. Bug triaging is the process of prioritizing bugs and assigning an appropriate developer for a given bug. The main task in bug triaging is to predict the most appropriate developer to fix a software bug from 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. Since manual bug triaging is a time consuming process, there have been several bug triaging algorithms to automate this process. One of the latest successful algorithms to address this problem is the Deep Triage. It employs Deep Bidirectional Recurrent Neural Network with Attention (DBRNN-A) approach for this classification task.

In this study, we implement an improved version of the DeepTriage. 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 by using a larger batch size with the same memory usage, (2) Using a corpus combining the data from different datasets to create a more generalized model, (3) Adding extra dense layers before the multiclass classification to improve the results. After running the experiments, we achieved the state of the 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%). The resulting model and its source code is made publicly available for future research in this area.

Keywords: recurrent neural networks, long short term memory, gated recurrent unit, bug triaging.

Derin Öğrenme ile Otomatik Hata Triyajlama: Bir Replikasyon Çalışması

Öz

Hata yönetimi hataları belirleme ve çözme sürecidir. Hata yönetimi sürecinde, bir hatanın belirlendikten sonra triyajlanması gerekir. Hata triyajlama süreci hatanın önceliklendirilmesi ve hatanın uygun bir geliştiriciye atanması şeklinde gerçekleşir. Bu sürecin asıl kısmı verilen bir hata raporunu çözmek için en uygun geliştiriciyi tahmin edebilmektir. Bu hata raporlarının metinsel kısımlarının (hata başlığı, hata tanımı) girdi olduğu ve önerilecek olan geliştiricilerin de çıktı olduğu bir sınıflandırma problem olarak tanımlanabilir. Otomatik olarak yapılmayan hata triyajlama zaman alan bir süreç olduğundan, hata triyajlamayı otomatik hale getirmek üzerine birçok algoritma

[*] Corresponding Author: eraytuzun@cs.bilkent.edu.tr

bulunmaktadır. Geçtiğimiz yıllarda bu problem üzerinde çalışan en son başarılı modellerden biri de Deep Triage'dır. Bu model sınıflandırma için derin, iki yönlü ve dikkatli tekrarlayan sinir ağı (DBRNN-A) kullanmaktadır.

Bu çalışmada literatürdeki başarılı bir hata triyajlama yöntemi olan Deep Triage'ın geliştirilmiş bir versiyonunu gerçekleştirilmiştir. Makalede önceden önerilen modelin performansını artırmak için original çalışmaya üç katkıda bulunduk: (1) Aynı bellek miktarıyla daha büyük veri grupları kullanarak eğitme zamanını düşürmek için LSTM yerine GRU kullanma, (2) Daha genel bir model oluşturmak için farklı veri setlerinin birleşmesinden oluşan bir sözlük kullanma ve (3) Sonuçları iyileştirmek için çok sınıflı sınıflandırmadan önce ilave sinir ağı katmanları koyma. Gerçekleştirdiğimiz deneylerin sonucunda Mozilla Firefox veri setinde %46.6 doğruluk ile original çalışmayla aynı sonuçları elde ettik. Chromium ver setinde ise orijinal çalışmadan (%42.7) daha yüksek bir doğruluk (%44.0) elde ettik. Bu konu hakkındaki ilerideki çalışmalar için geliştirilmiş model ve kaynak kodu paylaşılmıştır.

Anahtar Kelimeler: tekrarlayan sinir ağı, uzun kısa süreli bellek, kapılı tekrarlayan birim, hata triyajlama

1. Introduction

In a software project, it is an important task to assign an appropriate developer who could potentially provide a fix 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 the overall efficiency of the process. 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.

Bug triaging process is simply a classification problem in which bug title and bug description are taken as input, mapping them to one of the available developers. There are already some studies employing different machine learning algorithms to solve this problem in the literature. The major difficulty faced in these studies is that the input data (e.g. bug title and description) is in text format and hard to represent. Mani et al. (Mani, Sankaran, & Aralikatte, 2019) brought a new approach to represent bug reports by extracting features with an attention based deep bidirectional recurrent neural network (DBRNN-A) model that learns syntactic and semantic features from long word sequences in an unsupervised manner. In this study, we replicate their study and make 3 contributions to improve the original model: (1) Using Gated Recurrent Unit (GRU) instead of Long-Short Term Memory (LSTM) to fasten the training process by using fewer parameters, (2) Using a combined corpus from all datasets to improve the learning process, and (3) Changing the dense layer structure of to improve classification results.

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

2. Background

2.1. 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 ("QA: Quality assurance at Mozilla - Mozilla | MDN," n.d.). 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.

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 and in a way that it will not cause other bugs in the future. 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.

2.2. 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. (Cubranic & Murphy, 2004) and Anvik et al. (Anvik, Hiew, & Murphy, 2006) proposed a Naïve Bayessian classifier approach to apply text classification on bug reports in order to predict the relevant developers. Jeong et al. (Jeong, Kim, & Zimmermann, 2009) proposed a bug tossing graph approach based on Markov chains from the knowledge of reassigning. Xuan et al. (Xuan, Jiang, Ren, Yan, & Luo, 2010) proposed a semi-supervised text classification model for bug triaging. Different from the previous studies, Deep Triage (Mani et al., 2019) extracts a novel bug report representation from bug reports by using an Attention Based Deep Bidirectional Recurrent Neural Network (DBRNN-A). These extracted features are used as input to multi-class classifier in order to predict the best developer.

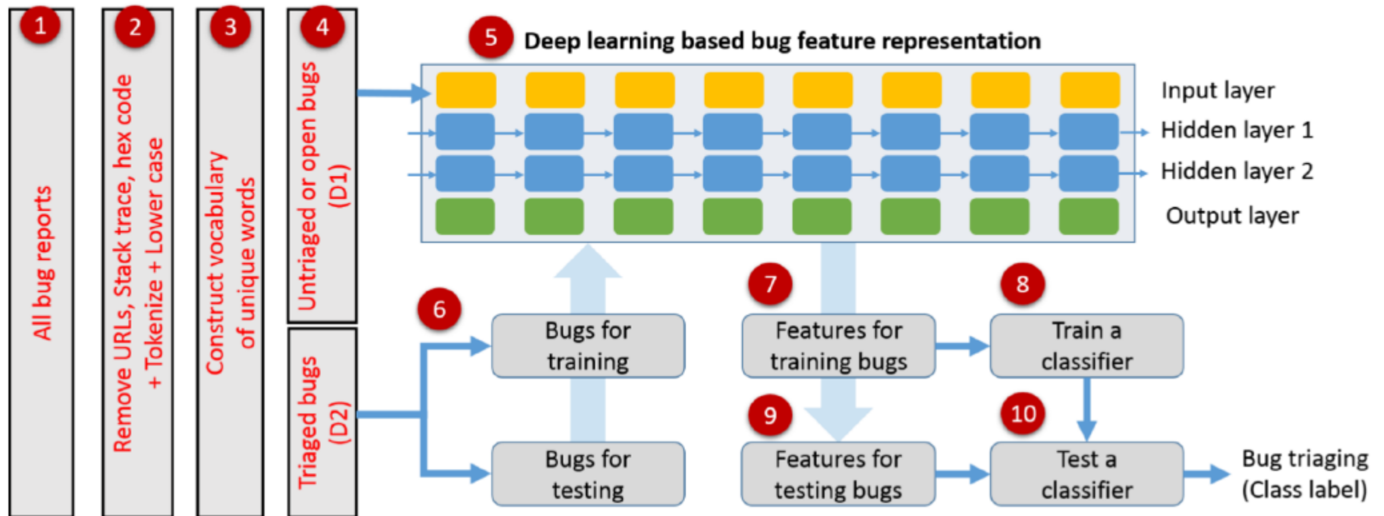

Figure 1. The flow diagram of the overall proposed algorithm highlighting the important steps (Mani et al., 2019)

3. Our Approach

3.1. 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 (Mani et al., 2019). 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.[†]

3.2. Preprocessing

The bug report datasets having *title*, *description*, *reported time*, *status* and *owner* are shared online by the authors (Mani et al., 2019) in JSON format. Before using the text in *title* and *description* fields of the bug reports, some parts of them should 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 Figure 1 is the preprocessing step.

3.3. 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 offers a better bug report representation but it fails to deal with high dimensionality and sparse data (Hindle, Barr, Su, Gabel, & Devanbu, 2012). Since these two representations are not good enough to represent a bug report, the authors (Mani et al., 2019) came up with a new approach to represent bug reports, *Word2Vec embedding* (Mikolov, Chen, Corrado, & Dean, 2013), 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 Figure 1 is the word embedding step.

3.4. Model Description

DBRNN-A (Mani et al., 2019) 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 directions and concatenates both representations. By representing the triaged bugs using the embeddings of untriaged bugs, then training DBRNN-A model with triaged bugs, the model learns how to extract features. At the end, a softmax classifier is used for classification using these features. Overall structure of the whole model can be seen in Figure 1. Also, detailed structure of DBRNN-A is shown in Figure 2.

[†] https://anonymous.4open.science/r/c1f33d60-67e9-480f-a3c9-8b9a0982a85c/

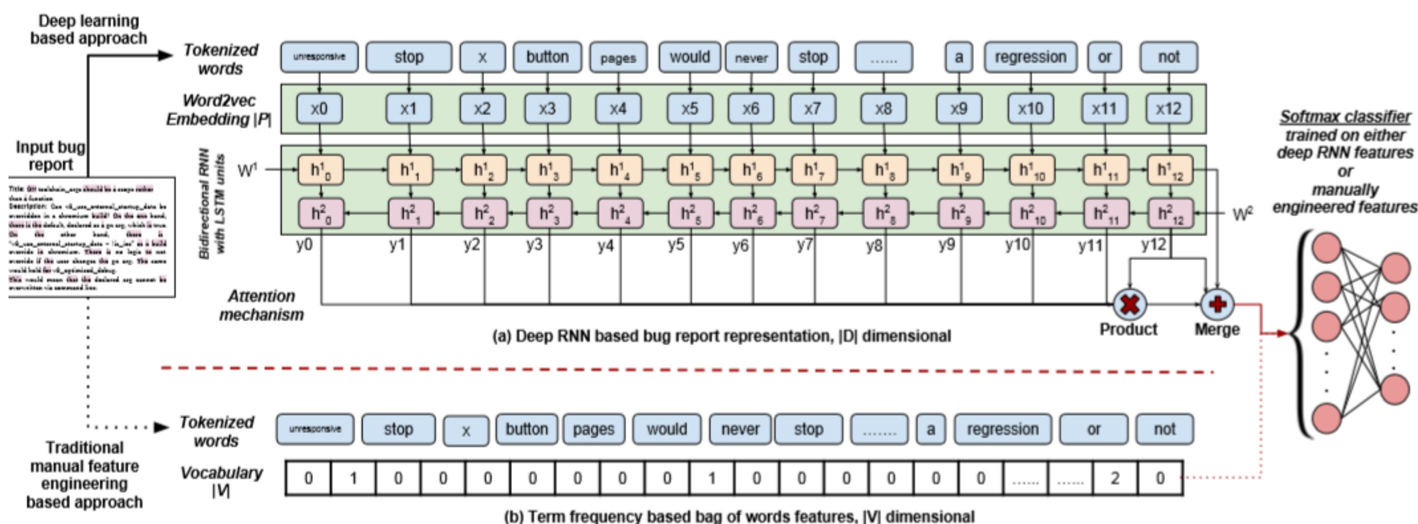

Figure 2. Detailed explanation of the working of a deep bidirectional Recurrent Neural Network (RNN) with LSTM units. (from Mani et al., 2019)

3.5. Implementation Details

The starting point of our implementation is the source code[‡] shared by the original paper (Mani et al., 2019). In the shared source code, there was no information about the versions of the Python libraries used. Also, there was no repository including all the source code files, the website shares the source code as snippets. Therefore, we replicated the results step by step and refactored some parts. In the following, we explain the details of our implementation.

In preprocessing, cleaning unnecessary parts are handled by using standard Python RegEx library[§], and Stanford NLTK[**] 3.4 is used for word tokenization. In word embedding, vocabulary is created by using Word2Vec from Gensim[††] 3.7.1. The implementation of the model is done in Keras[‡‡] 2.2.4 with the backend Tensorflow[§§] 1.13.1. When it comes to the soft attention layer in DBRNN-A, we had a problem with the soft attention layer[***] used in DeepTriage . Then, we implemented a similar soft attention layer with the help of a discussion[†††] from StackOverflow. Finally, we had a working version of the model in pure Keras. Also, the original implementation was in Python 2, but we used Python 3 in all the steps explained above. All the work we have done is shared in our GitHub[‡‡‡] repository.

4. Improvements on Deep Triage

After achieving similar results with the paper (Mani et al., 2019), we tried some additions and changes to improve the DeepTriage method in the scope of this study. Since the 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. Figure 3 shows the logic behind chronological cross validation.

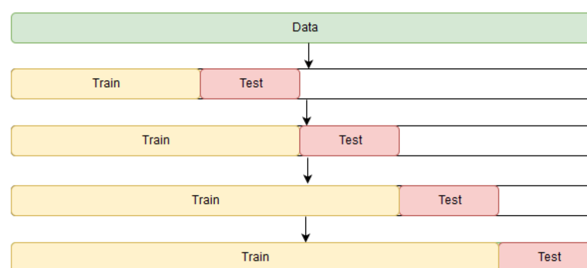

Figure 3. Chronological Cross Validation

4.1. Using GRU instead of LSTM

In the original study (Mani et al., 2019), 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 the same. More detailed results are available in Table 1.

[‡] http://bugtriage.mybluemix.net/#code
[§] https://docs.python.org/3/library/re.html
[**] http://www.nltk.org/index.html
[††] https://radimrehurek.com/project/gensim/
[‡‡] https://keras.io/
[§§] https://www.tensorflow.org/
[***] https://gist.github.com/braingineer/27c6f26755794f6544d83dec2dd27bbb
[†††] https://stackoverflow.com/questions/42918446/how-to-add-an-attentionmechanism-in-keras
[‡‡‡] https://anonymous.4open.science/r/c1f33d60-67e9-480f-a3c9-8b9a0982a85c/

Table 1. CHRONOLOGICAL CROSS VALIDATION TOP-10 ACCURACIES (IN PERCENT) 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
Deep Triage (Mani et al., 2019)	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 implementation 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 2. 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
Deep Triage (Mani et al., 2019)	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 3. 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 Implementation	**39.1**	**34.0**	**46.4**	**46.1**	**49.2**	**44.1**	**54.1**	**52.2**	**51.7**	**48.9**	**46.6**

Table 4. 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 Implementation	**34.7**	**42.4**	**43.9**	**39.8**	**40.5**	**43.9**	**44.1**	**47.1**	**49.5**	**53.6**	**44.0**

4.2. Merging all Dataset Corpora

In this section, we conduct experiments 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 can be completely different as the developers of different projects are not exactly the same. The illustration of this issue is available Figure 4.

To prevent such a situation, we employ a different approach for transfer learning. Instead of using only one dataset, we created the word2vec model by using all three datasets: Chromium, Mozilla Core and Mozilla Firefox. The results of the model trained with a merged (i.e. combined) corpus is given in Table 1. It is remarkable that the accuracy results are enhanced by a factor of 2% when trained by the combined corpus.

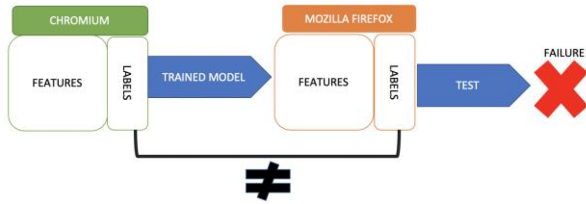

Figure 4. Illustration of the problem with transfer learning between datasets.

4.3. Changing Dense Layer Structure

Our datasets are created from large open-source projects. These projects consist of a large number of developers (.e.g. 1031 developers for Chromium). So, the number of class labels is large for the classification task. In the original paper, 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. Results for different dense layer configurations can be seen in Table 2.

5. 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 5. 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 3.

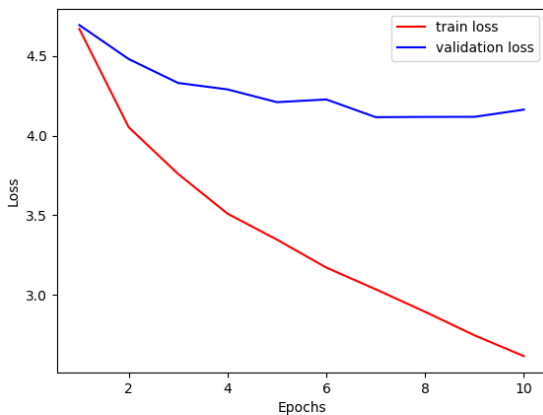

Figure 5. Loss values for CV#10 of Mozilla Firefox dataset with 20 minimum train samples per class. Accuracy values are given in Table 3.

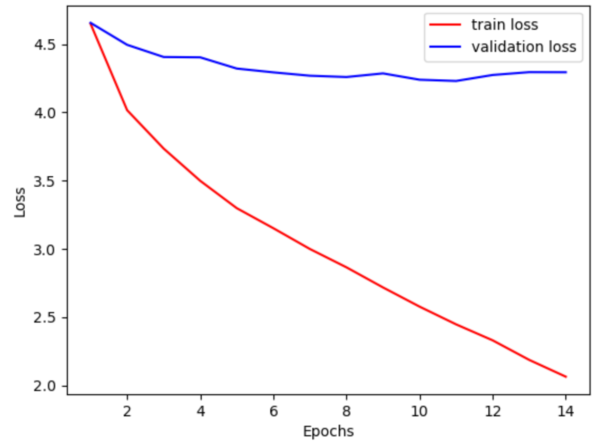

Figure 6. Loss values of Mozilla Firefox dataset with 20 minimum train samples per class for 82% train, 9% validation and 9% test splits. Top-10 test accuracy is 42.2%.

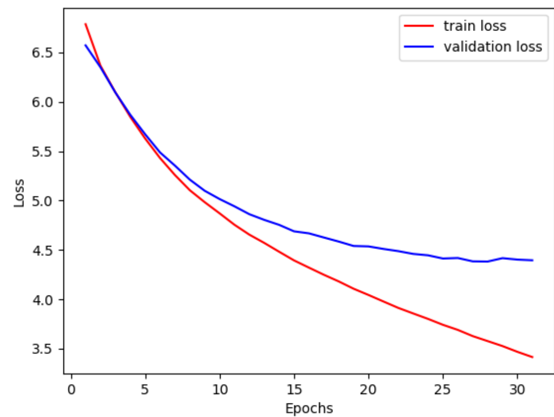

Figure 7. Loss values of Google Chromium dataset with 20 minimum train samples per class for 82% train, 9% validation and 9% test splits. Top-10 test accuracy is 50.5%.

Table 5. HYPERPARAMETERS FOR OUR FINAL MODEL. ONLY EXCEPTION IS THAT WE USE 32 AS BATCH SIZE FOR MOZILLA FIREFOX DATASET.

	Value
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

To evaluate our model with a completely different test data, we used Mozilla Firefox and Google Chromium datasets with 20

minimum training samples per class. We split them into three partitions: 82% train, 9% validation and 9% test. The loss graphs of Mozilla Firefox and Google Chromium are in Figure 6 and Figure 7 respectively. Top-10 test accuracy of Mozilla Firefox dataset is 42.2%, and top-10 accuracy of Google Chromium dataset is 50.5%.

6. Conclusions and Recommendations

Since manual bug triaging is a time-consuming process, there have been several bug triaging algorithms to automate this process. One of the latest successful algorithms to address this problem is the Deep Triage. In this study, we implemented the automated bug triaging model proposed by Mani et al. (Mani et al., 2019) and enhance the results. Consequently, we introduced the following three contributions to the original study and manage to achieve slightly better results:

- *Using GRUs instead of LSTM units:* As the GRU has less parameters, training larger batch sizes is possible with the same memory usage. Batch size can be larger up to roughly 50% because GRU has two gates while LSTM has three gates. Since we showed that using GRU does not affect the accuracy, the training process can be fastened significantly without a loss in the accuracy results.
- *Combining different datasets to create the corpus:* Because each software project consists of different developers and it is not possible to transfer the knowledge acquired from a dataset (no mapping for developers) to another dataset.
  Therefore, 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% improvement.
- *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.

In our future work, we are planning to test the enhanced algorithm with different datasets (both industrial and open-source datasets) and provide a comparative analysis with other bug triaging approaches.

References

Anvik, J., Hiew, L., & Murphy, G. C. (2006). Who should fix this bug? In *Proceedings - International Conference on Software Engineering* (Vol. 2006, pp. 361–370). New York, New York, USA: IEEE Computer Society. https://doi.org/10.1145/1134285.1134336

Cubranic, D., & Murphy, G. C. (2004). Automatic bug triage using text categorization. *16th Int. Conference on Software Engineering and Knowledge Engineering*, 92–97. Retrieved from http://www.eclipse.org.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2012). On the naturalness of software. In *Proceedings - International Conference on Software Engineering* (pp. 837–847). https://doi.org/10.1109/ICSE.2012.6227135

Jeong, G., Kim, S., & Zimmermann, T. (2009). Improving bug triage with bug tossing graphs. In *ESEC-FSE'09 - Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 111–120). https://doi.org/10.1145/1595696.1595715

Mani, S., Sankaran, A., & Aralikatte, R. (2019). 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* (pp. 171–179). https://doi.org/10.1145/3297001.3297023

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*. International Conference on Learning Representations, ICLR. Retrieved from http://ronan.collobert.com/senna/

QA: Quality assurance at Mozilla - Mozilla | MDN. (n.d.). Retrieved September 6, 2020, from https://developer.mozilla.org/en-US/docs/Mozilla/QA

Xuan, J., Jiang, H., Ren, Z., Yan, J., & Luo, Z. (2010). Automatic bug triage using semi-supervised text classification. In *SEKE 2010 - Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering* (pp. 209–214). Retrieved from http://arxiv.org/abs/1704.04769