An Approach for Predicting Bug Report Fields Using a Neural Network Learning Model

Korosh Koochekian Sabor, Mathieu Nayrolles, Abdelaziz Trabelsi, Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University, Montreal, QC, Canada
{k_kooche, m_nayrol, trabelsi, abdelw}@ece.concordia.ca

ABSTRACT

Bug fixing is a major activity in software development and maintenance. Developers use information reported in bug reports to identify bug causes and to provide a fix. Previous studies have shown that bug report fields are often reassigned to different development teams after the report is sent to developers. This can introduce considerable time delays in the bug handling process. To overcome this issue, there exist numerous methods to automatically predict bug report fields by examining historical data. In this paper, we combine a neural network based model with stack traces to predict bug report fields. When applied to bug reports of the Eclipse repository, we found that our approach achieves improvement of about 73% in product prediction accuracy, and 85% in component prediction accuracy over the well-known K-nearest neighbors (KNN) algorithm. The results of this preliminary study suggest that neural networks provide a robust alternative to traditional classification techniques.

KEYWORDS

Neural Networks, Stack Traces, Software Bug Reports, Mining Software Repositories, Software Maintenance;

1 INTRODUCTION

Bug fixing is known to be a time-consuming and costly task [1]. When a system crashes, users can report the bug by entering a bug description, attaching a stack trace, indicating the severity of the bug, the platform, the faulty product and component, etc. Developers can then use this information to provide a fix [3].

Xia et al. [14] have shown that 80% of bug reports have their fields reassigned several times after a bug report has already been sent to developers. They have also shown that bug reports with field reassignment have significantly longer fixing time than those with no reassigned fields. These fields are used by triagers to route the bug reports to appropriate development teams. Incorrect assignment of these fields can significantly delay the processing time of bug reports, which in turn affects the productivity of developers [2].

There exist several techniques to predict the likelihood that a particular bug report field gets reassigned, among which the most recent one is the study proposed by Xie et al. [15]. The authors rely on traditional machine learning algorithms, such as the K-Nearest Neighbors (KNN), to predict whether a bug report field would most likely be reassigned. Their approach achieves an average F-measure of approximately 63% and 73% when predicting the reassignment of component and product fields, respectively.

In this short paper, we propose a neural network learning algorithm, used in deep learning, to predict not only whether a specific bug report field gets reassigned, but also its new value. This is achieved by combining a neural network-based model for classification purpose with stack traces as features. The choice of stack traces is motivated by our previous work [7, 8], in which we showed that stack traces are excellent features for predicting bug severity [7] and duplicate bug reports [8].

In this study, we only focus on product and component fields, which tend to be the most often reassigned fields [4, 5, 16, 17]. When applied to bug reports of the Eclipse repository, our approach outperforms the use of KNN by an average of 73% and 85% for predicting product and component fields, respectively.

2 GOALS AND POTENTIALS

The main goal of this paper is to study the feasibility of combining neural networks with stack traces to predict component and product fields of bug reports. Many different neural network architectures have been developed throughout the years. One of the common tasks in designing such architectures is to select proper initial network weights and a suitable training algorithm. There exist many studies that leverage neural networks to help with software engineering tasks. To the best of our knowledge, this is the first attempt that a neural network

learning-based model combined with stack traces are used to address the problem of predicting bug report fields. Our preliminary results show that this approach holds real potential in building powerful techniques for predicting various aspects of bug reports, and hence improving the productivity of triagers and development teams. Software engineering researchers and practitioners can exploit these results to further explore the potential of machine learning techniques as they are adaptable, have learning capabilities and non-parametric.

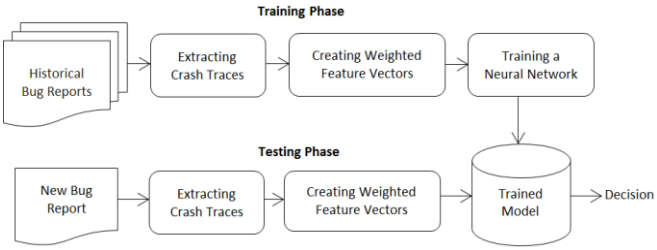

Figure 1: Proposed approach

3 APPROACH

Figure 1 shows the proposed approach, which consists of training and testing phases. The model is trained using stack traces embedded in historical bug reports. The resulting model is then used to predict fields of new bug reports. It should be noted that other features such as bug report descriptions can also be used. We chose stack traces because they tend to be more formal than bug report descriptions. Similar studies have shown that stack traces provide better accuracy than descriptions [6]. The following subsections describe in more detail the proposed approach.

3.1 Building Feature Vectors from Stack Traces

In our method, each bug report stack trace is mapped to a feature vector. The input feature vector is then weighed using term frequency-inverse document frequency (TF-IDF).

Let $T = f_1, f_2, \ldots, f_L$ be a stack trace of function calls f_1 to f_L, T of length L. The stack trace T is generated by a bug in a software system from a unique alphabet Σ of size m = |Σ| function names in the system. Let the collection of K stack traces that are extracted from the bug tracking system and then provided for designing the product and component prediction system be denoted by $\Gamma = T_1, T_2, \ldots, T_K$.

To determine the term frequency, each stack trace $T \in \Gamma$ is mapped into a vector of size m functions, $T \rightarrow \emptyset(T)_{o \epsilon \Sigma}$, where each function name $f_i \in \Sigma$ in the vector is either one (appeared in the stack trace) or zero (did not appear in the stack trace). The term vector can then be weighed by the term frequency (tf):

$$\phi_{tf}(f, T) = freq(f_i); i = 1, \ldots, m \qquad (1)$$

In Equation (1), freq(fi) is the number of times the function f_i appears in T, divided by the total number of functions in the stack trace (L).

It should be noted that not all functions have the same weight when comparing stack traces. Functions that appear rarely in stack traces must be given more weight. To give more weight to functions that rarely appear in stack traces, the IDF is used. The IDF increases the weight of rare functions while decreasing weight of popular functions. The term vector weighed by the TF-IDF is therefore given by:

$$\phi_{tf.idf}(f, T, \Gamma) = \frac{K}{df(f_i)} freq(f_i); i = 1, \ldots, m \qquad (2)$$

where the document frequency $df(f_i)$ is the number of stack traces T_k in the collection of Γ of size K that contains function name f_i. Thus, TF-IDF gives higher weight to functions that appear more in a particular stack trace $T \in \Gamma$, while appearing less in other stack traces of the collection Γ.

In an ideal scenario, the distribution of labels in the training set would be balanced (there is a similar number of samples for each label). Unfortunately, this scenario is not suitable for large systems. Since some products or components have fewer bug reports in the bug tracking system, the distribution of labels tends to be unbalanced. In this situation, any classifier, including neural networks, would be biased toward the majority class labels. Various techniques were developed to address the unbalanced dataset distribution issue. These approaches range from oversampling minority class labels, undersampling majority class label, to using cost-sensitive classifier [15].

In this paper, we used the random oversampling method to overcome the unbalance label distribution problem. This method randomly selects instances from minority class labels and adds copies of those instances to the dataset until an equal number of instances of each class label is reached in the dataset.

In practice, if the inputs are normalized, the neural network will be trained more efficiently. If the magnitude is considerably different from one input to another, then their influence will negatively affect the trained network. We use the min-max normalization to normalize the input data. We use min-max as a linear transformation technique from a range of values between $[Min_A, Max_A]$ to a range of values between [0, 1]. Considering a value V in $[Min_A, Max_A]$.

The normalization is performed using the following equation [10].

$$Normalzied\ value = \frac{V - Min_A}{Max_A - Min_A} \quad (3)$$

3.2 Training Phase

We use a three-layer model for the neural network. This network has been shown to learn any nonlinear function that is inherent between the inputs and outputs. Our model consists of three layers: an input layer, a hidden layer, and an output layer. The input to the neural network, for each bug report, is the feature vector weighted using TF-IDF corresponding to the stack trace of that bug report. The output is the faulty product or component depending on the objective of the prediction.

We used Theano[1] library to create and train our three-layer neural network. The number of nodes in the input layer corresponds to the number of distinct functions and the number of nodes in the output layer corresponds to the number of labels, e.g., products or components. The neurons in the input layer carry out no calculation, but only store the feature vector values. The output value of each node is the probability of the input instances belonging to that label.

Before training a neural network, many parameters (known as hyperparameters) should be properly initialized. The accuracy of a neural network learning-based model is closely related to the proper initialization of those parameters. Throughout the training phase, we use Relu [11] as an activation function. Since our problem is a multiclass classification problem and we have multiple output nodes, we use softmax [18] so that the output probability of each node of the output layer is between zero and one. The cross-entropy [18] is used as the cost function to measure the difference between distribution pairs. Using a backpropagation technique, which relies on a gradient descent based on cross-entropy, the weights of the neurons are updated, and the network is trained. For each neuron, the bias value is set to zero. Assume that there are p input-output pairs, s_p , t_p , available for training the network. After presentation of a pair of input-outputs, the weights for the interconnections are changed in proportion to the gradient of the error between the prediction and actual values according to the following relationship:

$$\Delta_p w_{ji} = \eta \delta_{pj} O_{pi} \quad (4)$$

where δ_{pj} for the input to hidden neurons, is defined as:

$$\delta_{pj} = O_{pj}(1 - O_{pj}) \sum_k \delta_{pk} w_{kj} \quad (5)$$

and, for the hidden to output neurons, it is defined as:

$$\delta_{pj} = (t_{pj} - O_{pj}) O_{pj}(1 - O_{pj}) \quad (6)$$

In Equation (4), η is a parameter that is decreased as the training proceeds.

It is well-known that the initial weights of neurons in a neural network are the most important factors that affect its performance [11]. Previous studies have shown that initializing the weights from a Gaussian distribution with zero mean and a small variance yields good accuracy [11]. In our model, we used a Gaussian distribution with zero mean and a standard deviation of 0.01. Another important hyperparameter is the learning rate. For a neural network to converge to an optimal solution, the learning rate should be carefully tuned to avoid the problem of overshooting. To satisfy this requirement, we tested our model with different learning rates and found that 0.01 provides the best results.

An epoch is defined as a single presentation of each training dataset to the network. After the cross-entropy calculation for all instances is performed, the backpropagation network is run to optimize the weights. However, the number of epochs is considered a critical factor in training a neural network. If the network is trained too many times, then it learns the noise when exposed to sample inputs and outputs from the training dataset. This is referred in the literature to as the overfitting problem. On the other hand, if the number of epochs is too low, the model is considered under fitted. To avoid overfitting or underfitting, various regularization techniques have been introduced [19]. In this paper, we used the early stopping technique. After each epoch, the error of the model is measured using a validation dataset separate from the training dataset, if the error rate increases after a certain number of epochs, the learning process is stopped. Then, the resulting model is tested against a testing dataset.

3.3 Testing Phase

The K-fold cross-validation [14] is used in this work to assess the performance of our model. The dataset is divided into K roughly equal partitions. Each time, the model is trained using k-1 partitions, and then it is tested using the remaining partition. However, since each bug in the bug repository is reported at a specific time, we need to make sure that a new bug report is tested against those reported earlier. We achieve this by following a similar experimental setting (called longitudinal data setup) introduced by Tamrawi et al. and Bhattacharya et al. [12, 13]. First, the bug reports are sorted based on their creation date. Then, the dataset is split into 10 non-overlapping

[1]http://deeplearning.net/software/theano

equal size folds. We train the network on the first fold, we apply the early stopping technique using the second fold, and then we test it on the third fold. In the second round, we train the neural network on the first and second fold, then we use the third fold as the validation dataset, and the fourth fold as the test dataset. In the last step, we train the network on the fold one to eight by using the ninth fold for validation, and the last fold as the test dataset.

4 EVALUATION

In this section, we show the effectiveness of our approach in predicting the product and component fields of a bug report.

We compare the performance of our approach to a traditional classification technique, namely the KNN. There exist several classification techniques proposed in the literature with various degrees of success. Among them, the KNN was chosen in this work due to its use in very recent studies that deal with the same problem of predicting bug fields reassignment [14]. The KNN is a lazy learning algorithm, based on choosing the K-nearest instances to each instance in the testing dataset. Then the label of the instance in the test dataset is selected using a majority voting algorithm. We train and validate KNN models using the exact process that we followed for training our neural network-based model.

4.1 Evaluation Metrics

To evaluate the performance of our approach, we measure the precision, recall, and F-measure [14] obtained from the prediction process of product and component fields. Precision is defined as the ratio of the number of bugs for which we correctly predicted their product field P_L or component field C_L (True Positives) to the total number of bugs predicted to have product field P_L or component field C_L (True Positives + False Positives), and is calculated as follows [14]:

$$Precision\ (p_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label\ p_L}{\#\ of\ bugs\ predicted\ to\ have\ label\ p_L} \quad (7)$$

Recall is defined as the ratio of the number of bugs for which we correctly predicted their product field P_L or component field C_L (TP) to the total number of bugs that actually have the product label P_L or component label C_L[14], and is computed as follows:

$$Recall\ (P_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label\ P_L}{\#\ of\ bugs\ actually\ having\ label\ P_L} \quad (8)$$

F-measure is the harmonic mean of precision and recall, and is defined as follows [14]:

$$F - measure\ (p_L) = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (9)$$

We compute the improvement of one approach over the other using Equation (2), which is used in [14]). In this equation, $F - measure_{KNN}$ and $F - measure_{NN}$ correspond to the F-measure of KNN and our neural network models respectively.

$$\text{Improvement} = \frac{F\text{-}measure_{NN} - F\text{-}measure_{KNN}}{F\text{-}measure_{KNN}} \quad (10)$$

4.2 Dataset

We use Eclipse bug reports between January 2009 and March 2015 to test our approach. They were automatically downloaded from the Eclipse bug repository. In the Eclipse repository, stack traces are embedded in bug report descriptions. To extract the content of the stack traces in Eclipse, we used the regular expression presented by Lerch et al. [9]. We have a total number of 199,500 bug reports out of which 18,906 bug reports have one or more stack traces. This accounts for around 10% of the total bug reports.

The extracted stack traces are preprocessed to remove noise in the data such as native methods that are used when a call to the Java library is performed. There are also lines in the stack traces that are labelled as 'unknown source'. This may have been caused by the way debugging parameters were set.

4.3 Results

The results of predicting the product field using our neural network learning-based model and the KNN algorithm are shown in Table 1. In average, using our proposed model, we outperformed KNN by 73% in average when predicting the product field. For all products, we obtained higher F-measure using the proposed neural network model compared to the KNN-based model.

For the component field, we have run the proposed neural network learning approach and the KNN method to predict the components of each product separately. Due to space limitation, the precision, recall, and F-measure obtained for each product component cannot be shown. Instead, we show an average value. If we consider precision of the first component of a given product as P_{C1} and the precision of the nth component as P_{Cn}, the average precision is calculated using the following equation. We apply the same method for measuring the average recall. The F-measure is computed based on the obtained precision and recall.

$$\text{Average precision} = \frac{P_{C1} + P_{C2} + \cdots + P_{Cn}}{n} \quad (11)$$

Based on Table 2, the proposed neural network model outperforms KNN by about 85% in average for all the components.

Table 1: Product prediction accuracy

Product	K-Nearest Neighbor	Neural Network	Improvement
	F-Measure	F-Measure	
Platform	22.15%	64.31%	190%
JDT	52.1%	60.27%	15.53%
PDE	31.11%	54.47%	75.09%
Equinox	31.50%	53.33%	69.27%
E4	18.27%	20.93%	14.15%
AVERAGE	31%	51%	73%

Table 2: Component prediction accuracy

Product	K-Nearest Neighbor	Neural Network	Improvement
	F-Measure	F-Measure	
Platform	12.91%	27.55%	113%
JDT	18.63%	55.39%	15.53%
PDE	29.31%	48.22%	64.50%
Equinox	14.13%	26.34%	86.45%
E4	19.22%	48%	148%
AVERAGE	19%	41%	85.50%

5 THREATS TO VALIDITY

In Eclipse, stack traces are manually copied by the users in the bug report description. Users may copy and paste partial stack traces inside the description. However, partial stack traces may affect the performance of our approach. Furthermore, our regular expression may have missed some of our functions when extracting stack traces.

The ratio of the number of bug reports having stack traces to the total number of bug reports in Eclipse repository is only 10%. We believe that the proposed neural network model can easily be extended to cover larger bug repositories. This is an important point to claim the generalization of our results.

6 CONCLUSION AND FUTURE WORK

In this paper, we studied the feasibility of incorporating neural network learning into the process of predicting the faulty component or product field of a bug report. Our results show that a neural network combined with stack traces yield superior accuracy compared to the existing KNN algorithm.

Over the past decade, long short term memory, which is a variant of recurrent neural networks (RNN), has been shown to be very effective in capturing dependencies in an input sequence. Efforts are currently under way to explore the potential of this network in predicting bug report fields when incorporated in a deep learning-based model.

REFERENCES

[1] M. Newman, "Software errors cost us economy $59.5 billion annually," *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[2] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," In *Proc. of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 52–56.

[3] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 308–318.

[4] A. Lamkanfi and S. Demeyer, "Predicting reassignments of bug reports an exploratory investigation," In *Proc. of the 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 327–330.

[5] A. Sureka, "Learning to classify bug reports into components" In Proc. of the *50th International Conference on Objects, Models, Components, Patterns*, 2012, pp. 288–303.

[6] K .K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic prediction of the severity of bugs using stack traces," In *Proc. of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON '16)*, 2016, pp. 96-105.

[7] Korosh K. Sabor, A. Hamou-Lhadj, A. Larsson, "DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Report," In *Proc. of the IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*, 2017, pp. 240-250.

[8] N. Ebrahimi, Md. S. Islam, A. Hamou-Lhadj, M. Hamdaqa, "An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models," In *Proc. of the IBM 26th Annual International Conference on Computer Science and Software Engineering (CASCON'16)*, 2016, pp. 75-84.

[9] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," In *Proc. of the 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 69–78.

[10] C. Saranya, G. Manikandan, "A Study on Normalization Techniques for Privacy Preserving Data Mining," *International Journal of Engineering and Technology, pp 14-18.*

[11] S. Krishna Kumar, "On weight initialization in deep neural networks", arXiv:1704.08863. Available online: https://arxiv.org/abs/1704.08863

[12] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 365–375.

[13] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," In *Proc. of the International Conference on Software Maintenance, 2010*, pp. 1–10.

[14] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction," *In IEEE Transactions on Reliability, 65(3)*, 2016, pp. 1094–1113.

[15] J. Zhang and I. Mani, "KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction," In *Proc. of the International Conference Machine Learning, Workshop on Learning from Imbalanced Data Sets*, 2003.

[16] J. Xie, M. Zhou and A. Mockus, "Impact of Triage: A Study of Mozilla and Gnome," In Proc. of the *International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 247-250.

[17] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, " "Not my bug!" and other reasons for software bug report reassignments," In *Proc. of the Computer Supported Cooperative Work*, 2011, pp. 395–404.

[18] Z. Feng, Z. Sun and L. Jin, "Learning deep neural network using max-margin minimum classification error," In Proc. of *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 2677-2681.

[19] Y. Shao, G. N. Taff and S. J. Walsh, "Comparison of Early Stopping Criteria for Neural-Network-Based Subpixel Classification," *In IEEE Geoscience and Remote Sensing Letters*, 8(1), 2011, pp. 113-11.