An Ensemble of Vector Space Model and Gated Recurrent Unit for Defect Localization

A course project for COMP767

Yue Cai Zhu
McGill University
Montreal, Quebec
yue.c.zhu@mail.mcgill.ca

ABSTRACT

TODO: add abstract in final draft

CCS CONCEPTS

• **Computing methodologies → Information extraction**; **Neural networks**; **Ensemble methods**; • **Software and its engineering → Software defect analysis**;

KEYWORDS

Defect Localization, Vector Space Model, Gated Recurrent Unit,Ensemble method

ACM Reference Format:
Yue Cai Zhu . 2018. An Ensemble of Vector Space Model and Gated Recurrent Unit for Defect Localization: A course project for COMP767. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Bug localization is one of the important steps in software maintenance. Especially in issue triaging, where triager has to predict the potential defected components based on his understanding to the issue and the system, then he needs to assign the issue to the right developer who has the required knowledge to fix it. Automating this step helps speed up bugs fixing and reduce costs in issue triaging. Lam *et al.* [8] proposed the state of the art approach to automate bug localization by ensemble revised Vector Space Model(rVSM)[19] and Deep Neural Network(DNN). Similar as most of the existed algorithms, this approach require source code information which is not always available to triagers or the QA team in large companies with code security requirement. Analyzing the code base dramatically increase the learning time of this kind of algorithm in large projects. Moreover, the fast evolution of source code further penalizes this approach that the predictive model has to be updated frequently. In this paper, we explore the possibility to predict defected components solely based on information learned from bug reports.

Recurrent neural network(RNN) has been proved to be efficient in capturing semantic as well as syntactic information from text to build statistical language models. But for source code, Hellendoorn *et al.* show that the RNN is not better than traditional IR method in language model building[6]. Bug report is different from common text and source code in that it is generally composed by common text and source code segment or error trace message. We believe that IR method and RNN can learn different information from bug reports and could complement each other. Thus the ensemble of these two could offer better performance than using only one of them.

The contribution of this paper is in two folds:

- an ensemble of naive tf-idf based VSM and Gated Recurrent Network to recommend high risk components solely based on bug reports.
- By analyzing the performence of the naive tf-idf based VSM and Gated Recurrent Network on their own, and the ensemble of these two, we show that IR method and RNN can learn different information from bug reports and could complement each other to offer better performance in the ensemble method.

2 BACKGROUND

2.1 Bug Localization In Post-release Software Quality Insurrance

One of the main purpose of software engineering is to deliver high quality product to cients. However, as software architecture getting more and more complex, the possibility of introducing bugs in the development being higher and higher. Different approaches are adopted in the industry to reduce the risk of delivering defeted product, test driven development, unit testing, test regression, peer review etc. But there are still bugs found by the client after release. In this case, being able to fastly locate the defeted components and hence assign the rigth developer to fix the customer reported bugs is critical to post-release software quality insurrance.

The traditional way is to do the assigning based on the triager's experience and his understanding of the reported issue. For experienced triager, they could spot light some potential defected location very fast and accurate. But for triager with less experience or even new to the project, the process could take longer time and the prediction would not be correct. Thus they may assign a wrong developer for the issue and reassigning would be required and a longer time to get the bug fixed. Jeon *et al.* citejeong2009improving reported that the Eclipse project takes on average 40 days for bug assignment and another 100 days for reassigning.. The numbers for Mozilla are 180 days for the first assigning and another 250 days for reassigning.

To help address this problem, recent research applies NLP techniques to learn from bug reports and suggest a ranked list of potential defect location[5][8][10][19][12][15], or to suggest the developer who could fix the bug[1][2][18][7]. All these approaches focus on combining source code and code change history from version controling tools(ex: Git) with bug reports to make the prediction. However, in companies that has a strict source code security policy, which is a common situation for large companies, source code or even code change history is not available for triagers. Bug report is the only source of information they can use to perform the triaging. Therefore, in this work we focus on bug localization with only bug reports as the input source.

2.2 Data Mining Bug Reports

The most popular technique used in information retrieval from bug reports is the VSM with TF-IDF as entry value[5][19]. Another frequent used algorithm is the Latent Dirichlet Allocation(LDA)[10]. And resent works try to ensemble IR techniques with Machine Learning to achieve better performance[7][8]. Jonsson *et al.* [7] ensemble TF-IDF with different ML algorithms with Stacked Generalization. Lam *et al.* [8]combines rVSM with DNN.

To the best of our knowledge, there is not past work to use DNN, more specifically deep Recurrent Neural Network independently to learn from bug reports and suggest potential defect location. In this work, we apply DNN with multiple layers of Gated Recurrent Unit to learning from bug reports, and compare its performance to VSM. Moreover, we ensemble the DNN with VSM to leverage the performance.

3 METHODOLOGY

3.1 Vector Space Model

Vector Space Model was first proposed by Salton *et al.* [14] to quantify documents. With a pre-defined and indexed vocabulary of size s: $V = \{t_1, t_2, ..., t_s\}$, each term t_i in a given document d could be represented by a weight quantity w_i, where i is its index in the vocabulary V. Thus the given document d could be represented by a vector $d = [w_1, w_2, ..., w_s]$. Obviously, the length of the vector d is the same as the size of the vocabulary. If a term t_j is not in document d, then $w_j = 0$. With this vector representation, we can analyze the documents with quantitative methods.

3.2 TF-IDF

The selection of weighting method to use in VSM is important to the performance. We choose TF-IDF[13] because of its well reported performance in previous works. TF is short for term frequency. There are different schemes to evaluate TF, in our implementation, we calcuate TF by fomular 1

$$tf(t, d) = f_{t,d} \tag{1}$$

where t is the given term, d is the document and $f_{t,d}$ is the count of term t in document d.

IDF is short for Inverse Document Frequency and is generated from equation 2in our implementation.

$$idf(t) = log\frac{n_d}{df(d,t)} + 1 \tag{2}$$

where n_d is the total of documents and $df(d, t)$ is the number of documents containing term t.

Then the weight w_t is given by:

$$w_t = tf(t) * idf(t) \tag{3}$$

Then the resulting vector w is normalized to get the vector d:

$$d = \frac{w}{||w||_2} = \frac{w}{\sqrt{\sum_{i=1}^{s} w_i^2}} \tag{4}$$

3.3 Our VSM Bug Localization Model

Similar as how to rank risky files based on similar bugs in rVSM[19], after we obtain the vector representation of all bug reports, for a new comming report d_n and an old bug report d_i that has been fixed in the training set D , we calculate their Cosine similarity:

$$similarity(d_n, d_i) = \frac{d_n \cdot d_i}{||d_n|| * ||d_i||} \tag{5}$$

For a given component c, define D_c to be the set of bugs that are fixed by changing code in c, then the risk of c is given by:

$$risk(c) = min(similarity(d_n, d_i))\forall d_i \in D_c \tag{6}$$

And the components are ranked by their risk score from maximum to minimum. Our VSM model achieve a MRR of 43.12%, 19.45%, 14.40% and 2.38%, and a MAP of 39.27%, 10.73%, 5.74% and 1.18% for projects ZXing, SWT,AspecJ and Eclipse respectively.

3.4 Deep Gated Recurrent Unit Network

Recurrent Neural Network(RNN) is good at modeling sequential data. As Mikolov introduced RNN to language modeling[9], more and more applications of RNN in document analysis related task are proposed, for example, machine translation. In recent years, people starts to apply RNN to model source code[11]. It is believed that RNN can learn syntactic and semantic rules from text documents. Given that bug reports are also written by human natural language, we believe that we can use RNN to learn from bug reports and make precise predictions on risky components. We apply Gated Recurrent Unit in our RNN implementation. GRU is proposed to solve the gradient vanishing problem in training a (RNN)[3]. It acheive the same performance as Long Short Term Memory but with less computations. Its stucture is show in figure 1. h is the current state and $\hat{h}$ is context state which could memorize the impact of the input history. The reset gate r controls whether the state from the current input should be memorized in $\hat{h}$. The update gate z controls whether the memory from $\hat{h}$ should considered when updating h to give out prediction.

Given the input vector x_t at step t, each gates and states are calcuated recursively as:

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \tag{7}$$

$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \tag{8}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h) \tag{9}$$

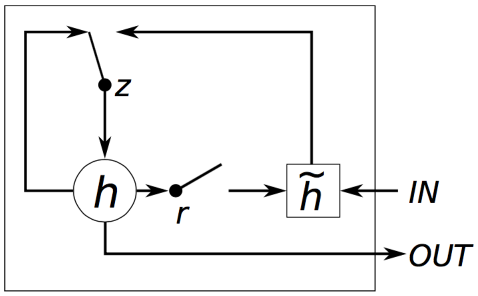

Figure 1: Gated Recurrent Unit[3]

Where o is point-wise multiplication, W and U is the weight matrices for the corresponding component, b is the bias vector. σ_g is the sigmoid activation function which has the form:

$$\sigma_g(x) = \frac{1}{1 + e^{-1}} \tag{10}$$

and σ_h is the hyperbolic tangent activation function with the form:

$$\sigma_h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{11}$$

In our DRNN implementation, the input layer pass the input vector to the embedding layer which generates the embedding of size 100. The resulted embedding will then be passed to the stacked hidden GRU layers of size 10. The encoding of the GRU hidden layers is pass to fit a logistic regression to predict the possibility of each components to be defected. TODO: draw the graph to show the deep structure

The number of layers and the embedding size is selected randomly. Increasing the number of layers could improve performance a litter with longer training time as penalty. Small change to the embedding size does not change the result siginicantly Large change (ex: doubling the embedding size) would result in huge memory requirement and failed in runtime. Our DRNN model achieve a MRR of 34.16%, 67.92%, 44.10% and 11.81%, and a MAP of 30.83%, 51.13%, 21.39% and 6.53% for projects ZXing, SWT,AspecJ and Eclipse respectively.

3.5 Ensemble of VSM and GRU

Ensemble method [4] is used to combine predictions from multiple different model to give out better performance than a single model. [8] ,[16] and [7] apply ensemble method to perform bug triage and achieve good performance. Different schemes can be used in ensembling. Stacked Generalization[17] is theorectally the most effective one which adds one more classifier to take predictions from ensembled models as input and is trained to reduce prediction error. Jonsson *et al.* [7] ensembled TF-IDF with different ML algorithms with Stacked Generalization. However, as a preliminary research, we apply a simlper scheme For a given component, we takes the minimum of the two risks given by the two aforemention models as the final risk.

$$risk(c) = min(risk_{VSM}(c), risk_{DNN}(c)) \tag{12}$$

TODO: report performace

4 EXPERIMENT DESIGN

4.1 Data Set

We perform our experiment on the exact same data set used in [16] ,[12] and [19]. The detail of the data set is summarized in table 1

There are in total 3,379 bug reports from four popular open source projects in this data set, AspectJ, Eclipse, SWT, and ZXing.

- ZXing, a barcode image processing library for Android platform, is the smallest data set. It contains 20 bugs fixed between March and September 2010 with 391 source files being changed.
- SWT is an open source widget tookit. This data set has 98 bugs fixed from October 2004 to April 2010 with 484 source files being fixed.
- AspecJ, An aspect-oriented extension to the Java programming language
- Eclipse, the famous open source integrated development environment, contains 3075 bugs fixed in the period between October 2004 and March 2011 with 12863 files being changed.

For each bug report, we can obtain its title, description from the issue reporter, and the files being changed to fixed the reported bug.

Differently from previous work, the granularity of our experiment is on component level instead of on file level. Because we don't use the source code information to simulate a situation of strict source code security, new added files cannot be captured in the prediction with either VSM or DNN. Thus we first truncate the path of a given file by removing the file name to obtain the component containing the given file. We assume that by knowing the potential defected component is enough for assigning the appropriate developer to solve the issue. Because if a developer knows a component very well, he should also has knowledge to the new added file to that component. After the truncation, there are 11, 15, 555 and 534 components in ZXing , SWT, Eclipse and AspectJ respectively. We removed one bug from the SWT project since the defected file has no component level, it locates in the top most directory. Removing one record does not impact on the performance evaluation given the relatively large number of bugs. The detail of the three data sets is summarized in table 1

4.2 Preprocessing

Bug reports is different from normal text in that it could has code elements, source file names or error log in its content. Here we list some examples to illustrate the situation. This is some bug description from the Eclipse project:

Bug 77046: If the Pattern Match job is already running when the console's partitioner reports that it is finished work, it is possible that the pattern matcher can think that it's finished before checking the entire document. This is the trace from the failed test: Incorrect number of messages logged for build. Should be 8. Was 2 junit.framework.AssertionFailedError: Incorrect number of messages logged for build. Should be 8. Was 2 at
org.eclipse.ant.tests.ui.BuildTests.testOutput(BuildTests.java:41) at
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at

sun.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:39) at
sun.reflect.DelegatingMethodAccessorImpl
.invoke(DelegatingMethodAccessorImpl.java:25) at
org.eclipse.ant.tests.ui.AbstractAntUIBuildTest.access$0
(AbstractAntUIBuildTest.java:1) at
org.eclipse.ant.tests.ui.AbstractAntUIBuildTest$1
.run(AbstractAntUIBuildTest.java:44) at
java.lang.Thread.run(Thread.java:534)

After the normal text describing the issue, the reporter also attached the error trace log as well. If we don't perform the preprocessing correctly, neither VSM or DRNN model could learn patterns from the error trace log, since the long component name would be seen as one term, but actually there are multiple important terms there. For example, the path "org.eclipse.ant.tests.ui.BuildTests.testOutput(BuildTests.java:41)" contains important terms: ui,Build,Test. Here is another example:

Bug 76208 Boolean.getBoolean((String)newValue) queries if a system property exists with the argument name...not what we want in this context.

The reporter is reporting an undesired functionality from a code segment. Again important terms would be missed if we don't separate the code segment correctly.

Thus we replace all non word character by white space to separate terms, and if a small case character followed by an upper case character, the separate them by a white space. We also remove all numbers to reduce the size of the vocabulary and hence the requirement on computation resource. After the preprocessing, we summarize the data set detail in table 1.We apply the 80-20% split to generate the training and testing set for each project.

4.3 Metrics

To evaluate the performance of different models, we apply the following common used metrics:

Top-k Bugs Hit Top-k bugs hit is the number of bugs that are correctly localized from the top k components in the output ranked list. At least one component has to been captured in the top k list for the given prediction be classified as a hit.

Mean Reciprocal Rank(MRR) is a measure to evaluate the quality of the ranked list from a recommendation system. If for all queries under test, the higer the the first hit relevant item is ranked, the higher value the MRR will be and vice versus. It's calculated by the following formula:

$$MRR = \frac{1}{|Q|} \sum^{i \in Q} \frac{1}{rank_i} \tag{13}$$

Where Q is the set of test queries, in our context would be the bugs in the testing data set. i is a particular bug in Q and $rank_i$ is the ranking of the first collect prediction in the output list.

MRR only cares about the ranking of the first correct prediction. However, there could be multiple defected components for one reported bug. We also need to check how well different models rank the other defected components.

Mean Average Precision(MAP) To evaluate how well different models rank all defected components we first calculate the average precision for one bug report:

$$AP = \frac{1}{|C|} \sum_{i=1}^{N} prec(i) * relevance(i) \tag{14}$$

Where C is the set of all defected components for the given bug. N is the total number of components in the system. $prec(i)$ is the precision for the top-i list. $relevance(i)$ is 1 if the i^{th} item is defected, otherwise 0.

Then MAP is calculated by the equation:

$$MAP = \frac{1}{|Q|} \sum^{Q_i \in Q} AP(Q_i) \tag{15}$$

The model with higher MAP score is the better.

5 EVALUATION AND DISCUSSION

5.1 Experiment Result

We report the performance of the three models in table 2, 3 and 4. For the project Zxing and SWT, because the total of their components is close to 10, we report the top-1 bug hit rate. For the project AspectJ and Eclipse, given the large number of components, we report the top-10 bug hit rate.

As shown in Table 2, All models acheive 100% bug hit for the project ZXing, DRNN achieves the best result in all other three projects. It gets 100%, 79.31% and 20.29% hit rate for the project SWT, AspectJ and Eclipse respectively. VSM achieves the lowest bug hit rate which is 50%, 41.38% and 3.6% for SWT, AspectJ and Eclipse respectively. DRNN achieve 163.82% more bugs hit on average than VSM.

As shown in Table 3, VSM achieves the best MRR of 43.12% for ZXing, but DRNN achieves the best MRR in all other three projects which are 67.92%, 44.10% and 11.81% for SWT, AspectJ and Eclipse respectively. DRNN achieve 163.82% more bugs hit on average than VSM.

5.2 Is RNN Better Than IR Methods in Mining Bug Report ?

REFERENCES

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

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

[3] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[4] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems.* Springer, 1–15.

[5] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 2009. On the use of relevance feedback in IR-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on.* IEEE, 351–360.

[6] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 763–773.

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

Table 1: Data Set Summary

Project	Description	#bugs	period	#files	#component
ZXing	barcode image processing library for Android platform	20	2010.3-2010.10	391	11
SWT v3.1	open source widget tookit	98	2004.10-2010.4	484	15
AspecJ	An aspect-oriented extension to the Java programming language	286	2002.6 -2006.10	6485	534
Eclipse v3.1	open source integrated development environment	3075	2004.10-2011.3	12863	555

Table 2: Top k bugs hit

Projects	VSM	DRNN	Ensemble
ZXing(top-1)	4(100%)	4(100%)	4(100%)
SWT(top-1)	10(50%)	**20(100%)**	18(90%)
AspecJ(top-10)	24(41.38%)	**46(79.31%)**	28(48.28)%
Eclipse(top-10)	22(3.6%)	**124(20.29%)**	26(4.26%)

Table 3: Mean Reciprocal Rank(MRR)

Projects	VSM	DRNN	Ensemble
ZXing	43.12%	34.16%	52.62%
SWT	19.45%	67.92%	58.63%
AspecJ	14.40%	44.10 %	46.22%
Eclipse	2.38%	11.81%	4.12%

Table 4: Mean Average Precision(MAP)

Projects	VSM	DRNN	Ensemble
ZXing	39.27%	30.83%	51.77%
SWT	10.73%	51.13%	39.71%
AspecJ	5.74%	21.39%	15.69%
Eclipse	1.18%	6.53%	1.97%

[8] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on.* IEEE, 218–229.

[9] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association.*

[10] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, 263–272.

[11] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[12] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on.* IEEE, 345–355.

[13] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[14] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[15] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. 2013. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories.* IEEE Press, 2–11.

[16] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 53–63.

[17] David H Wolpert. 1992. Stacked generalization. *Neural networks* 5, 2 (1992), 241–259.

[18] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. Developer prioritization in bug repositories. In *Software Engineering (ICSE), 2012 34th International Conference on.* IEEE, 25–35.

[19] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 14–24.