Mangrove Phase 2 Progress Summary

1. INTRODUCTION

In the second phase of mangrove project we are exploring machine learning techniques and tools to learn a model which can classify bug reports of static code analysis tools (SCATs) as false -i.e. there is no bug in the code as stated by the report- or true -a real bug correspondence-. Hence, our problem is a binary classification task given the source code and bug report. Obviously, there are many machine learning algorithms/methods which we may apply to this problem. A body of research suggests that natural language processing (NLP) methods can be very successful in learning properties from source code itself [9]. This mostly because programming languages are more repetitive and more predictable compared to natural languages. That repetitiveness and predictability make code written in a programming language is suitable for learning.

In the application of NLP techniques in programming languages, a source code corresponds to a text -sequence of words. Therefore, our binary classification task is very similar to sentiment analysis (i.e. inferring attitude of writer of a text). So we model our problem as a sentiment analysis problem in order to benefit from existing tools.

N-gram language model accounts for last n words when making a decision about the next word by computing a probability value for it. Thus, it cannot capture a relation between two words that are more then n words away from each other. Which is a very common case for source code -i.e. a statement which is written early in the code may effect the statements that comes very late in the code. Therefore, we chose to use a neural network-based long short term memory (LSTM) language model as they can capture such deep dependencies. In all the experiments we used a single layer LSTM (explained in *here*).

2. LIST OF EXPERIMENTS

2.1 Data preprocessing

We did the following four transformations to the code before feeding to LSTM; 1) removing comments, 2) replacing all printout strings with 'STRING', 3) reducing the code into a line, and 4) tokenizing.

2.2 C/c++ experiments

We did our first experiments on c/cpp languages. We took scan-build, the static code analyzer of clang, as the subject SCAT and Juliet is only data set we used in this experiments. Following are two research questions we aim at addressing;

RQ1 Do variable/function identifiers effect the learning task?

RQ2 Can we train a generic model that can work for all warning types? [1] Or do we need to train a model per warning type?

To answer this two research questions, we trained 3 models.

Model 1. scan-build generates 183464 bug reports for entire Juliet. 92465 of them are false and 90999 of them are true (real bugs). After preprocessing each code file, and extracting each word occurring the files into the dictionary, we feed all data points into our LSTM. The resulting accuracy [2] is just a little better than random guess.

Model 2. We then replaced all variable and function identifiers with a constant value, thus had a very small dictionary -only 130 keywords and operators of c/c++ language. The resulting accuracy is no different than the accuracy of Model 1.

Model 3. Now we only focus on one particular type bug (reports); 'CWE134 Uncontrolled format String'. We chose this bug type because, it constitutes to largest dataset; 9120 data points, 4680 false, and 4440 true. The resulting accuracy is slightly better then Model 1 & 2, and random guess.

In the experiments above, we did not make use of information from the bug reports. Now the next RQ is;

RQ3 How can we benefit from the information in the bug report?

Model 4. In order to measure the impact of report information, we only make use of warning line number information by putting a mark to the line in the code. We have also reduced the dataset by filtering out the reports which

[1]According to *MITRE*, there are 1005 different types software bugs/errors. Juliet covers 118 of them

[2]Accuracy computed as normalized number of misclassified data points; 0 acc value means perfect result, and 1 worst

Exp	lang	dictionary	mark	dataset	data size (false+true)	time	training acc.	test acc.
Model 1	c/c++	all words	none	all juliet	183464 (92465f+90999t)	26h	0.432	0.493
Model 2	c/c++	lang	none	all juliet	——— ” ———	12h	0.431	0.487
Model 3	c/c++	lang	none	CWE134	9120 (4680f+4440t)	59.2m	0.319	0.388
Model 4	c/c++	lang	yes	CWE134	1650 (930f+720t)	125m	0.105	0.098
Model 5	c	lang	yes	CWE134	1350 (780f+570t)	89m	0.078	0.074
Model 6	c	lang	none	CWE457	526 (175f+351t)	20m	0.022	0
Model 7	c	lang	yes	CWE457	526 (175f+351t)	19m	0.022	0
Model 8	c++	lang	none	CWE762	526 (175f+351t)	6m	0	0
Model 9	c++	lang	yes	CWE762	526 (175f+351t)	6m	0	0

Table 1: C/c++ experiments

differ in warning explanation. The resulting accuracy for the training is 0.098 -significantly better than Model 3.

Model 5. There might be structural differences in between c and c++ code in the dataset. Therefore we excluded c++ data points for Model 5 and the resulting accuracy is slightly better.

Model 6. Same as Model 4 but for CWE457 - Use of uninitialized variable.

Model 7. Same as Model 5 but for CWE457.

Model 8. Same as Model 4 but for CWE762 - Mismatched Memory Management Routines.

Model 9. Same as Model 5 but for CWE762.

2.3 Java experiments

Having perfect accuracy results in the latest models of c/c++ experiments raises the question of generalizability. To address this question, we need to use these models on some data points which do not come from Juliet test suite. But we are not aware of any such data points covering the CWEs we are working on and good number of false positives and true positives. Therefore we switched to java programming language as there are at least two good test suites for SCATs evaluation; Juliet for Java, and the Owasp benchmark. We can use them as two different datasets. But now we also need to change the subject SCAT from scan-build (which is for c/c++ code -not java) to FindSecBugs.

After analyzing the CWEs that occur both in Juliet4J and owasp, we decided to experiment with 'CWE78 - Command injection' first.

In the following models, we made use of 3 more peace of information from the bug report; function called the line of the warning, severity of the warning and confidence level. This information became available as switched to FindSecBugs.

The dictionary is consisted of java keywords, operators, classes and domains coming with Java 8, and numbers up to 100.

Model 10. We trained on 302 Juliet data points; 36 false and 266 true data points. We then test this model on 237 Owasp data points; 111 false and 126t. Resulting accuracy is almost as bad as random guess.

Model 11. One concern the Model 10 is that training data set does not have equal number of false and true examples. To resolve this we trained this model with a balance training data set of 72 data points; 36 false and 36 true, then test on Owasp. Which lead to an accuracy result worse than random guess.

Here we realized the the LSTM usually classifies the data points in the test set as false positive. It is very hard to understand why this has been the case.

For the following models we used 'CWE89 - SQL injection' data as it leasts to bigger data set.

Model 12. This time we used owasp data set as training data and juliet as testing

At this point our concern about generalizability turned out to be real as least for CWE78 warning type. LSTM overfits the training data and probably picks of some structures which are not really relevant to the warning. Thus thereby LSTM cannot correctly classify the data points from outside world. Now the challenge is to improve the approach to train more generic models.

Model 13. Following the idea of creating more data points suggesting in [5], we enlarged the training data set by creating number of mutants for each data point. These mutant does not effect the semantics of the code in terms of preserving the bug report legitimate. Although the accuracy for training has increased, testing accuracy is still as bad as random guess. Meaning that, LSTM still picks up structures irrelevant to the warning.

Model 14. Now create a training set which has data points both from Owasp and Juliet. For Owasp, we increased the number of mutants to have around 20K data points. Then we took 240 of Juliet data points and added them to the training set. Resulting accuarcy is now good for both training and testing.

2.4 Data dependency graph experiments

Reducing the code to a small snipped which contains only the statements that are relevant to the warning report was a challenge of the first phase of the mangrove project. TODO

3. RELATED WORK

Papers we have looked at so far;

[9] is the first paper that argues NLP can be successful for code mining. Then following body of research follows;

[1–4] code suggestion, completion, and summarization, and idiom finding. In particular, [2] uses deep (convolutional) neural networks.

[7,8] learning API usage patterns. [8] uses recurrent neural network.

[15, 16] js-nice work and code completion using n-gram and neural network language models.

[20, 21] uses recurrent neural networks to detect clones. Works on AST and proposes a method for ASTs2BinaryTree conversion. The latter one is more like a survey comparing n-gram based and deep learning based models.

[18] uses caching to improve efficiency of n-gram models.

[17] learning false positive report filter using decision trees. User labels a set of about 200 data points then the algorithm very quickly trains a model to filter false report. Experiments on a static analyzer fro java script and

[6] suggests using LSTM for project management.

Exp	training dataset	training data size (false+true)	testing data size (false+true)	time	training acc.	test acc.
Model 10	CWE78 Juliet	302 (36f+266t)	237 (111f+126t) Owasp	10m	-	0.468
Model 11	CWE78 Juliet	72 (36f+36t)	237 (111f+126t) Owasp	4m	-	0.531
Model 12	CWE89 Owasp	2400 (1193f+1207t)	240 (120f+120t) Juliet	6m	0.11	0.54
Model 13	CWE89 Owasp	10375 (5175f+5200t)	240 (120f+120t) Juliet	10h	0.03	0.504
Model 14	CWE89 Owasp & Juliet	19502 (9187f+9815)	240 (120f+120t) Juliet	36h	0.12	0.11

Table 2: Java experiments.

[5] mines static analysis rules for javascript. Counter example guided learning using decision trees.

NLP in general; [11] character level deep learning. [10] recursive neural networks for sentiment analysis.

Crowd sourcing models; [12, 13]

[19] uses juliet for C/C++ to experiment with some static analyzers

To read:

- Convolutional neural networks over tree structures for pro- gramming language processing [14]; classifying programs by functionalities
-

4. REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[2] M. Allamanis, H. Peng, and C. Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. *arXiv:1602.03001 [cs]*, Feb. 2016. arXiv: 1602.03001.

[3] M. Allamanis and C. Sutton. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[4] M. Allamanis and C. Sutton. Mining Idioms from Source Code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM.

[5] P. Bielik, V. Raychev, and M. Vechev. Learning a Static Analyzer from Data. *arXiv:1611.01752 [cs]*, Nov. 2016. arXiv: 1611.01752.

[6] H. K. Dam, T. Tran, J. Grundy, and A. Ghose. DeepSoft: A vision for a deep model of software. *arXiv:1608.00092 [cs, stat]*, July 2016. arXiv: 1608.00092.

[7] J. Fowkes and C. Sutton. Parameter-free Probabilistic API Mining Across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 254–265, New York, NY, USA, 2016. ACM.

[8] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API Learning. *arXiv:1605.08535 [cs]*, May 2016. arXiv: 1605.08535.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[10] O. Irsoy and C. Cardie. Deep Recursive Neural Networks for Compositionality in Language. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2096–2104. Curran Associates, Inc., 2014.

[11] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and Understanding Recurrent Networks. *arXiv:1506.02078 [cs]*, June 2015. arXiv: 1506.02078.

[12] T. D. LaToza and A. v. d. Hoek. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Software*, 33(1):74–80, Jan. 2016.

[13] T. D. LaToza, A. D. Lecce, F. Ricci, W. B. Towne, and A. v. d. Hoek. Ask the crowd: Scaffolding coordination and knowledge sharing in microtask programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 23–27, Oct. 2015.

[14] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293. AAAI Press, 2016.

[15] V. Raychev, M. Vechev, and A. Krause. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM.

[16] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[17] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 762–774, New York, NY, USA, 2014. ACM.

[18] Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

[19] A. Wagner and J. Sametinger. Using the Juliet Test Suite to Compare Static Security Scanners. In *SECRYPT*, pages 244–252, 2014.

[20] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[21] M. White, C. Vendome, M. Linares-VÃąsquez, and D. Poshyvanyk. Toward Deep Learning Software Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.