COM S 665 A

Homework Assignments: COM S 665 A
Paper Reading Notes & Other

Paper 7

Title: A Comprehensive Study on Deep Learning Bug Characteristics

Year: 2019

Conference: The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)

Authors: Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan

Point # 1: Frequent bug types reported on StackOverflow

As a frequent user of Keras and Tensorflow it is very interesting to see an analysis of and distribution of bug types that other developers are dealing with. Figure 1 shows that the sample of posts collected from StackOverflow pertaining to bugs in code written in Caffe, Keras, TensorFlow, Theano, and Torch were most frequently related to data pre-processing or to structural logic of the model. From my own experience I can attest to these results pertaining to Tensorflow and Keras. In my opinion, Tensorflow code is very prone to bugs in data such as incorrect data type (tensor vs a numpy array), or incorrect data shape that results in inconsistency between placeholder definition and data received to name a few. Therefore, 30% of Tensorflow bugs being data related seems representative and accurate. On the other hand, I am a little surprised that 24% posts in Keras pertain to data bugs. Keras is very high level and easy to use without needing too many specifications like Tensorflow. I myself don't encounter data bugs while using Keras too often, but the reason for the high number of posts related to Keras data bugs could be that developers without detailed deep learning knowledge choose Keras for its easy use. That is maybe why we see Keras posts related bugs in feature engineering or data shuffling, which are fairly "rookie" mistakes. The other common type of bug reported by this paper is structural logic, a bug

sub-type under Structural Bugs. While the the other three sub-types (Control and Sequence Bug, Data Flow Bug, Initialization Bug) are described more concretely the logic bug sub-type is explained very vaguely…. I wish the authors elaborated on the definition more given that 43% Caffe posts pertain to it.

Point # 2: Bug Types Encountered by Developers vs. Bug Fixes Pushed to Libraries & Impact of Bugs

The finding that the distributions of bug types posted on StackOverflow and bug types fixes pushed upstream to Github Deep Learning LIbraries projects are very similar is very interesting and also makes sense. It is likely that engineers developing and updating deep learning libraries receive bug reports from developers utilizing their libraries. Therefore, the bug types encountered by the library users are likely to get reported and fixed.

I would also like to comment on section 5 - impacts from bugs. Buggy code utilizing deep learning libraries can crash - I believe that is the good scenario, because it can be the case that ML code with incorrect logic runs without errors and may even deceitfully happen to show good performance on some data. One “rookie” mistake I noticed with a lot of deep learning projects on github or ML blogs is an incorrect choice of loss function. Some developers often incorrectly choose binary cross entropy loss function (suitable only for binary classification) for a classification problem that involves more than two classes. In some cases the model “happens to work”, but is incorrect, because the correct loss function for a multi-class problem is categorical cross entropy.

Point # 3: Future Work: Compatibility, Documentation, Error Messages

Several future work directions that are warranted by Finding 3 that shows that API bugs are common, are better compatibility between API versions, better documentation, and more meaningful error messages. In my own experience, Tensorflow suffers from incompatibility between different Tensorflow versions. I have encountered many issues related to this, which is why I am a little surprised that the percentage of API bugs reported for Tensorflow is not higher than 11%. It is also often quite difficult to find detailed documentation, especially for older Tensorflow functions. Finally, I can confirm that error messages are often not entirely clear and it is difficult to understand what the root cause of the issue is. For instance, looking at the illustrative example in section 3.3 instead of an

error message “received unknown keyword arguments”, and error message “argument epochs has been deprecated and replaced by nb_epochs” would be a lot more informative.

Paper 6

Title: Adversarial Examples are not Bugs, they are features

This paper is presented by me - link to my slides:

https://docs.google.com/presentation/d/1BiWLmIhbfNRf4MlEFfMnUoWOLrePlbo7J1tfR0MbNLM/edit?usp=sharing

Paper 5

Title:Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey

Year: 2017

Conference: ACM Computing Surveys

Authors: SEYED MOHAMMAD GHAFFARIAN and HAMID REZA SHAHRIARI

Point # 1: Hopes and Fears - Section 3.1

Section 3.1 of GHAFFARIAN & SHAHRIARI (2017) gives a very concise and unbiased overview of the merit of utilizing AI in cyber security. I appreciate that the authors included literature discussing the potential of machine learning in computer security applications and also its limitations. I think it is important to explore new areas of research that are important and seem promising, because this is how progress and innovation are achieved. However, it is also advisable to be cautious. I think that it is important that authors also included literature documenting the fact that anomaly-based security threat detection systems that use machine learning are rarely used in practice. Research is always ahead of the industry, but it is important to assess the feasibility of the applications of research.

I also agree with the statement that “ machine-learning and data-mining techniques should be tailored to suite the characteristics of security problems”, because machine learning is

not “magic” that can be applied to solving any problem. Currently, machine learning is successful at very specific tasks, so to ensure success I think that machine learning models must be tailored to a specific application.

Point # 2: Imbalance of data - Section 4.2

Section 4.2 discusses the challenge of imbalance class data, which has adverse impact on performance of machine learning models. I think that most data points in cyber security related datasets the vast majority of code examples are non-vulnerable, which means that a model trained on data primarily consisting of training data labeled as non-vulnerable will be inherently biased to categorize any code as non-vulnerable. I think that this is a problem to consider, because the accuracy of an ML model applied to a security problem may have relatively high accuracy, but any false negative classification may have a huge impact. In addition to imbalanced datasets, I wonder if there is also a lack of training data. All developers strive to release secure code, which doesn’t contain security vulnerabilities. The paper mentions that labeled datasets consisting of software metrics are readily available, but none of the various model types trained using this type of data are not very successful. This seems to be evidence that it is the data or feature engineering part of the process of model development that have hindered the performance. Therefore, I wonder if there is a lack of suitable training data for learning a classifier that can distinguish and also pinpoint vulnerable code.

Point # 3: Future Work

Future work could focus on finding and building better datasets, because datasets containing meaningful information will allow us to extract high quality features. This effort could aid developing more accurate and reliable machine learning models for software vulnerability detection since it is the quality of features extracted from training data that have a significant impact on the success of the ML model. The application of machine learning in computer/cyber security is very exciting and is used in the industry. Nevertheless, a decision to deploy a new ML model in production with the aim to protect computer systems from adversaries should be made with extreme caution and only after very rigorous testing. For example, neural nets, the most common model type that has achieved state-of-the-art results in various disciplines, is still a “black box”. If we don’t know

how to interpret a model, applying that model to a new problem increases the complexity even more. In conclusion, utilizing ML and data mining in cyber security definitely warrants further research - I personally would enjoy research this topic. However, deployment of these new techniques to real problems must be done with extreme caution.

Group Project Idea

Team Members: Eliska Kloberdanz, Jeremy Roghair, Hung Phan

Proposal:

The purpose of this project is to apply machine learning to software engineering. In particular, we propose to use natural language processing (NLP) for estimating/predicting the development effort of software projects. The dataset we plan to use consists of several projects and their and their storypoints, a common unit of measure that comes from agile development terminology used for estimating the effort involved in implementing a user story or resolving an issue. We intend to use this training set to learn a recurrent neural net (RNN) model that can predict the size of new issues and hence, estimate the development effort required for a particular project. The inputs into the model are user stories and issues that are expressed in the form of plain text. Therefore, we propose to use an RNN, which is naturally suitable for NLP tasks that involve processing sequence data.

Accurate dates of milestones, timelines, and human resources are very difficult to estimate, but also very important in order to properly plan the development process. Therefore, a tool for automatic software development prediction would alleviate many issues stemming from poor project planning. Research on automatically predicting the cost of software development was proposed [1], however this approach confronts several challenges [2]. Typically software projects are initiated with a particular business objective in mind. This objective is usually formulated as functional and non-functional requirements, use cases and high level descriptions of various features. Software teams working on the project would typically break these down into stories (agile terminology) and assess the time and effort to complete each story. The challenge is, even as humans, estimating this effort and

assessing a timeline for completion is difficult. Therefore, we aim to utilize machine learning to build a model that can take the previously mentioned text as an input and output a prediction for the amount of required effort of a software project.

References

1. Boehm, B. W. et al. (2010). Software Cost Estimation with Cocomo II with Cdrom. 1st. Ed
2. Natural Language Processing and Machine Learning Methods for Software Development Effort Estimation - Vlad-Sebastian Ionescu, Horia DEMIAN, Istvan Gergely Czibula (2017)

Dataset: IEEE TSE 2018

Paper Reading Notes & Thoughts

Paper 4

Title: Property Inference for Deep Neural Networks

Year: 2019

Conference: Automated Software Engineering (ASE 2019)

Authors: Divya Gopinath, Hayes Converse, Corina S. Păsăreanu, and Ankur Taly

Point # 1: Merit

The idea that the behaviour/characteristics of a neural net can be described by its neuron decision patterns (e.i.: whether a neuron is activated or not) makes sense, because it is the learnable weights and biases and also the chosen non-linear activation functions that determine the extent of activation of all neurons. The paper argues that this approach for analyzing neural nets is analogous to studying path constraints in program analysis. I agree with that - following the neurons that get activated forms a path that should be characteristic to a particular label input. I think that these neuron activation patterns are indeed characteristic properties in neural nets since we can re-use the same network architecture for different tasks as long as we learn the correct weights, which in turn determine the activation patterns. Also, utilizing program analysis to better understand neural nets is a great example of leveraging well developed techniques from software

engineering in machine learning, which lacks formal methods for testing, analysis, and debugging.

Point # 2: Computing Network Properties

The choice of decision trees for learning layer patterns from data has its advantages and disadvantages, which the authors don't mention. Decision trees can be very powerful and accurate, but they are also very unstable. This means that learning a decision tree for the same task on the same data does not yield the same decision tree each time. In fact, my personal experience is that the first (and most important) nodes of the decision trees are the same, but the lower layers of the trees can differ significantly. On the other hand, since decision trees consist of nodes and edges representing "yes" or "no" answers to their parent node and leading to further nodes, decision trees seem to be a natural choice for modelling paths in binary problems. Neuron patterns describe which neurons are activated (on) and which are not (off). Therefore, decision trees are an appropriate model type choice for this task. However, because they are extremely unstable they should be compared with other model types to see if the advantages of using decision trees outweigh the disadvantages.

Point # 3: Theorem 1, Layer Patterns as Interpolants, and Experiments

Theorem 1 shown under part III in section A says that "… the value of any neuron in the pattern can be expressed as a linear combination of the inputs and that each on/off activation adds a linear constraint to the input predicate …". The theorem and its proof by induction seem correct to me, but I don't see how is Theorem 1 a new finding. The formula essentially expresses the fact that all neurons in a feed-forward neural net are connected and can have two types of activations: *on* or *off*, which in turn depend on **W** . **X + bias**. In particular, if this linear combination of input, weights, and biases is greater than 0 the neuron is activated, otherwise it is off. All of this is standard knowledge about feed-forward fully connected neural nets…

Part III also talks about using patterns found in the network layers as interpolants to prove that input A implies output B. They propose to use the layer patterns to decompose the proof into more manageable smaller sub-proofs as follows: (A ==⇒ σl), (σl ==⇒ B) / (A ==⇒ B). This seems like a good idea, however its correctness hinges on the premise that activation patterns in layers **are** important characteristics that determine the network outputs.

The fact that the experiments in this paper were conducted not just on MNIST, but also on ACASXU gives the results more credibility. In my experience, there are many papers that perform experiments only on MNIST to support their hypothesis. However, MNIST is a very simple dataset, and it has been my experience that replicating the same experiments on more complex datasets often doesn't yield the expected results. I also appreciate that the authors say " this work as a first step in the study of formal properties of DNN", which helps to make the reader aware that this is one of the first attempts at automatically inferring formal properties of feed-forward neural networks. I think that it is important to take the findings of a paper with a grain of salt and look for further evidence

Paper 3

Title: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks

Year: 2019

Conference: NeurIPS

Authors:Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, Yang Liu

Point # 1: Merit and Future Work

This paper proposes a graph neural net model that can be used to classify code as vulnerable or not vulnerable at a function level. Therefore, it could be viewed as unit testing aimed at detecting security vulnerabilities. While software testing at a granular level is important it should be coupled with integration tests to be rigorous. The idea of learning a general model that can detect vulnerabilities in functions within code is very interesting.

However, I wonder if such a model is language specific. For example, the authors mention detecting memory leaks, which can occur in C or C++ functions, but not in, for example, Python. The goal of this paper is similar to Long and Rinard (2016) in that they attempt to learn a model of "correct code". However, this paper is focused on binary classification of individual functions, which seems a lot more manageable task. I think that it is a good idea to break up a problem into small components, solve them, and then build on those solutions. Future work could build on this paper and develop a model of correct code consisting of several levels of abstraction with the first level being individual functions, the next could be larger blocks of code such as individual classes, the third could be individual scripts, and the fourth could be interactions of all scripts within a codebase. Such a model could be used for very rigorous software testing, which encompases unit and integration testing.

Point # 2: Architecture: Embedding layer and Gated Graph Recurrent Layers

It seems that the two main novel ideas pertaining to the model architecture are: (1) the combination of its three components: the graph embedding layer, gated graph recurrent layers, and the Conv module, (2) and the design of the Conv module. The graph embedding layer does not contain any new type of code representation, but it does combine many existing code representations to achieve better expressiveness by capturing very detailed semantics. The second component of Devign architecture are the gated graph recurrent layers. The choice of recurrent layers is very appropriate and standard since the model input is textual sequence data. The interesting part of this architecture layer is the fact that the layers are gated graph layers. GNNs were first proposed by Scarselli et al. (2009) and extend neural networks for processing the data represented in graph domains. GNNs have several subcategories that tailor the graph model for specific domains such as: graph convolutional networks, graph attention networks, graph auto-encoders, graph generative networks and graph spatial-temporal networks. While GNNs have the advantage of having the ability to process graph structures, they also suffer from some serious shortcomings. Zhou et al. (2019) claim that "graph neural networks suffer from over-smoothing and scaling problems. There are still no effective methods for dealing with dynamic graphs as well as modeling non-structural sensory data." Additionally, the graph recurrent neural net layers are gated, which allows the model to have "long memory", which seems helpful for

analyzing larger blocks of code, but may not be necessary for analysis of individual functions without considering their interactions.

Point # 3: Architecture: Conv layer & Conclusion

The paper claims that the Conv layer, the last layer in their architecture model is the novel contribution. However, it seems that it is only a convolutional layer taken from a standard convolutional neural net, which was adapted for GNNs. Applying 1-D convolution and max pooling are very standard techniques used for feature extraction from textual data. (Note, 2-D convolution is used for image input data.) The authors apply a sigmoid function to obtain the final prediction whether a function has or hasn't vulnerabilities, which is also the standard approach for any neural net binary classifier. Overall, I don't perceive a novel contribution in the Conv module. Nevertheless, I do like the approach of splitting a difficult problem into more manageable components by focusing on individual functions and classifying them as vulnerable or not vulnerable. This is a good stepping stone to more sophisticated models for code testing and analysis.

Paper 2

Title: DeepStellar: Model-Based Quantitative Analysis of Stateful Deep Learning Systems

Year: 2019

Conference: ESEC/FSE

Authors: Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, Jianjun Zhao

Point # 1

The merit of this paper is that it adds to a very important recently emerged issue of security vulnerabilities, robustness, and testing of deep neural net systems. The paper focuses on developing a better understanding and quality assurance of recurrent neural networks (RNNs), which are most often used in natural language processing for their natural ability to process data consisting of sequences. Speech recognition deep neural net systems are used in many critical applications. Therefore, a failure, poor quality, or a malicious attack on those systems can have catastrophic consequences. For example, a company called

Nuance developed a speech recognition software, which is commonly used in American hospitals. The speech recognition system that transforms speech into text allows doctors to save time by dictating patient's diagnosis, recommended procedures and treatments, and other critical information to create and update medical records. The Nuance system was hit by the NotPetya cyber attack in 2017, which caused serious operational disruptions at many hospitals due to lost medical records and slower processing of patients. This illustrates that considering only training and testing accuracy is not enough to evaluate a deep neural net system. Especially the systems used in production warrant a lot more rigorous testing and this paper takes a step forward to aiding this issue.

Point # 2

Even though machine learning made the most significant advancements in the last 10 years or so the field has been around since the 1960's, which makes it quite surprising that there are not many testing techniques except for analyzing accuracy. It seems that until now the focus during machine learning models development has been very myopic. As illustrated by the example of Nuance and NotPetya, accuracy is only one of the many performance metrics that should be considered to evaluate ML systems. It is interesting that the field of software engineering has a well developed toolbox of testing methods, while machine learning seems to be lagging behind in this regard.

Point # 3

The abstract model of RNNs developed in this paper is (in my opinion) very impressing. The notion of abstraction is a very important and widely used concept in computer science. Abstraction allows us to simplify complex systems to understand them better conceptually. That is why I think that developing a model of RNNs that abstracts away unnecessary details allows us to gain a better understanding of it. It is another step away from treating deep neural nets as black box systems. Additionally, I think that it is not only the idea of an abstract model for RNNs, but also its execution that is impressive and makes a lot of sense. RNNs maintain and utilize internal states, so using a finite state machine to model the system seems very appropriate. Finally, the quantitative metrics and algorithms developed in the paper also seem very logical. Overall, this paper is very current in terms of the issues it explores, its ideas are logical and impressive, and it is also well written.

Paper 1

Title: Automatic Patch Generation by Learning Correct Code

Year: 2016

Conference: POPL

Authors: Fang Long & Martin Rinard

Point # 1

Given that supervised machine learning has been successfully applied to various complex tasks (e.g.: image classification, object detection, natural language processing) why couldn't be those ML techniques applied to learning a model of correct code and generating patches to buggy software. Prophet learns a model of correct code on a training set of software bugs and corresponding patches utilizing maximum likelihood estimation (MLE). This is a very standard approach in supervised ML. After Prophet "develops an understanding of what correct code looks like" it creates another model for ranking the suitability of various possible patches with the goal of learning to select software patches that actually fix the issue with the code. I think that from a high level point of view both of these steps (learning a model of correct code and learning correct patches) is possible. Code can be thought of as mathematical operations on data and I believe that those operations share common rules and principles across different applications and programming languages. Therefore, I think that there exists universal principles of correct code and correct patches that can be learned.

Point # 2

To achieve learning models of correct code and suitable patches that are universal it is necessary to abstract away semantic and other programming language details (as the paper points out.) The issue that I see here is that software bugs can be language specific, so a patch generator based on a model that is too abstracted away may not be successful. And even though the authors claim that Prophet is universal, they admit that it was mostly

trained and applied to C code. I think that it would be interesting to try to develop automatic patch generation using ML that is specific to a certain language.

Point # 3

I assume that the models developed were neural nets, which are the model types that most commonly achieve the best results. It has been shown that transfer learning and reprogramming of neural nets is possible, which shows that neural nets performing similar tasks learn similar parameters and can be reused. Perhaps Prophet, a universal patch generator, could be further fine tuned for specific applications and languages to perform better.

Survey Topics: Selection of 5 papers that interest me

1. Paper 1
    a. Title: Zero-shot Learning via Simultaneous Generating and Learning
    b. Conference: NIPS
    c. Year: 2019
    d. Topic: generative modelling, missing/unseen data
    e. Authors: Beomhee Lee, Hyeonwoo Yu
2. Paper 2
    a. Title: Deep ReLU Networks Have Surprisingly Few Activation Patterns
    b. Conference: NIPS
    c. Year: 2019
    d. Topic: understanding of neural nets, neural nets expressiveness
    e. Authors: Boris Hanin, David Rolnick
3. Paper 3
    a. Title: Adversarial Examples Are Not Bugs, They Are Features
    b. Conference: NIPS
    c. Year: 2019
    d. Topic: adversarial machine learning, robustness of ML models
    e. Authors: Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, Aleksander Madry

4. Paper 4
    a. Title: Generalization in Generative Adversarial Networks: A Novel Perspective from Privacy Protection
    b. Conference: NIPS
    c. Year: 2019
    d. Topic: generative adversarial networks (GANs), private learning algorithms, information leakage
    e. Authors: Bingzhe Wu. Shiwan Zhao, Chaochao Chen, Haoyang Xu, Li Wang, Xiaolu Zhang, Guangyu Sun, Jun Zhou
5. Paper 5
    a. Title: Machine Learning Based Automated Method Name Recommendation: How Far Are We
    b. Conference: ASE
    c. Year: 2019
    d. Topic: ML for SE, using ML for better readability and maintainability of programs
    e. Authors: Lin Jiang, Hui Liu, He Jiang

In addition to these topics, I am also very interested in libraries for machine learning - I would like to learn more of the inner workings of Tensorflow and also learn about best software practices for developing ML libraries.