2019 International Conference on Mining Software Repositories, Montreal, Canada, 26-27 May 2019

Lessons learned from using a deep tree-based model for software defect prediction in practice

Hoa Khanh Dam
University of Wollongong
Australia
hoa@uow.edu.au

Trang Pham
Deakin University
Australia
phtra@deakin.edu.au

Shien Wee Ng
University of Wollongong
Australia
swn881@uowmail.edu.au

Truyen Tran
Deakin University
Australia
truyen.tran@deakin.edu.au

John Grundy
Monash University
Australia
john.grundy@monash.edu

Aditya Ghose
University of Wollongong
Australia
aditya@uow.edu.au

Taeksu Kim
Samsung Electronics
Republic of Korea
taeksu.kim@samsung.com

Chul-Joo Kim
Samsung Electronics
Republic of Korea
chuljoo1.kim@samsung.com

***Abstract*—Defects are common in software systems and cause many problems for software users. Different methods have been developed to make early prediction about the most likely defective modules in large codebases. Most focus on designing features (e.g. complexity metrics) that correlate with potentially defective code. Those approaches however do not sufficiently capture the syntax and multiple levels of semantics of source code, a potentially important capability for building accurate prediction models. In this paper, we report on our experience of deploying a new deep learning tree-based defect prediction model in practice. This model is built upon the tree-structured Long Short Term Memory network which directly matches with the Abstract Syntax Tree representation of source code. We discuss a number of lessons learned from developing the model and evaluating it on two datasets, one from open source projects contributed by our industry partner Samsung and the other from the public PROMISE repository.**

***Index Terms*—defect prediction, deep learning**

I. Introduction

As software systems continue playing a critical role in all areas of our society, software defects have significant impact onto businesses and people's lives. Substantial research has gone into developing predictive models and tools which help software engineers and testers to quickly narrow down the most likely defective modules of a software codebase [1, 2]. Early defect prediction helps prioritize and optimize effort for inspection and testing, especially when facing with cost and deadline pressures. Identifying defects in software however becomes increasingly difficult due to the significant growth of software codebases in both size and complexity.

The growth in size and complexity of codebases has led to the use of machine learning techniques in building defect prediction models. Common machine learning techniques derive features (i.e. predictors) from software code and feed them to classical classifiers such as Naive Bayes, Support Vector Machine and Random Forests. Substantial research (e.g. [3–5]) have gone into carefully crafting features which are able to discriminate defective code from non-defective code such as code size, code complexity (e.g. Halstead features, McAbe, CK features, MOOD features), code churn metrics (e.g. the number of code lines changed), and process metrics. However, those features do not truly reflect the syntax and semantics of code. In addition, software metric features do not generalize well: features that work well in a software project may not perform well in other projects [6].

An alternative is using Natural Language Processing (NLP) techniques to generate features from code. A common technique is using Bag-of-Words (BoW) which treats code tokens as terms and represents a source file as term-frequencies. The BoW approach is however unable to detect differences in the semantics of source code due to differences in code order or syntactic structure (e.g. $x \geq y$ vs. $y \geq x$). Hence, recent trends started to focus on persevering code structure information in representing source code. However, recent work such as [7] does not fully encode the syntactic structure of code nor the semantics of code tokens, e.g. fails to recognize the semantic relations between "for" and "while".

Our industry partner, Samsung, is the leading provider of Android platforms and has substantial software development teams and very large codebases. Motivated by Samsung's defect prediction needs for these large codebases, we have developed a new deep tree-based model for defect prediction. The model was built upon the Long Short-Term Memory (LSTM) [8], a powerful deep learning architecture to capture the long context relationships in code where dependent code elements are scattered far apart. The syntax and different levels of semantics in source code are usually represented by tree-based structures such as Abstract Syntax Trees (ASTs). Hence, we adapted a *tree-structured LSTM network* (Tree-LSTM) [9] in which the LSTM tree in our prediction system *matches exactly* with the AST of an input source file, i.e. each AST node corresponds to an LSTM unit in the tree-based network.

This tree-based LSTM model for source code aims to preserve both syntactic and structural information of the programs (in terms of ASTs). Through an AST node embedding mechanism, our representation of code tokens also aims to preserve their semantic relations. Our prediction system takes as input a "raw" Abstract Syntax Tree representing a source file and predict if the file is defective or clean. The features are automatically learned through the LSTM model, thus eliminating the need for manual feature engineering which occupies most of the effort in traditional approaches. We evaluated this model using real projects provided by Samsung and the PROMISE repository. We report on the lessons we learned from our experience in developing our new defect prediction models in this industry context.

We firstly provide a motivating example in Section II. Section III describes how our prediction model is built. We describe how the model is implemented and trained (Section IV), and evaluated (Section V). Section VI serves to discuss the lessons learned. In Section VII, we discuss related work before summarizing the contributions of the paper and outlines future work in Section VIII.

II. Motivating example

The following example illustrates the limitations when using existing approaches for defect prediction. Figure 1 shows two simple code listings written in Java. Both contain a *while* loop in which the integer at the top of a given *stack* is repeatedly removed through the *pop* operation. Listing 1 has a defect: if the given stack's size is smaller than 10, underflow exception can occur when the stack is empty and the *pop* operation is executed. Listing 2 rectifies this issue by checking if the stack is not empty just before invoking the *pop* operation.

```
 1  int x = 0;
 2  if (!stack.empty())
 3  {
 4    while (x < 10)
 5    {
 6      int y;
 7      y = stack.pop();
 8      x++;
 9    }
10  }
```

Listing 1. A.java

```
 1  int x = 0;
 2  while (x < 10)
 3  {
 4    int y;
 5    if (!stack.empty())
 6    {
 7      y = stack.pop();
 8    }
 9    x++;
10  }
```

Listing 2. B.java

Fig. 1. A motivating example

Using existing techniques for such defect prediction [3–5, 10–12]) suffers from the following limitations.

Similar software metrics: The two code listings are identical with respect to the number of code lines, conditions, variables, loops, and branches. Thus, they would be indistinguishable if software metrics (as widely used in existing approaches [3]) are used as features. In many other cases, two pieces of code may have the same metrics, but they behave differently and thus have different likelihood of defectiveness.

Similar code tokens and frequencies: Recent approaches use the Bag-of-Word (BoW) technique to process the actual code content and represent a source code file as a collection of code tokens (e.g. *int*, *x*, *if*, etc.) associated with frequencies (e.g. 2 for *int* in Listing 1). The term-frequencies are then used as the predictors for defect prediction. However, this is not necessarily the best presentation for code. In fact, the code tokens and their frequencies are also identical in both code listings. Hence, relying only on the term-frequency features would fail to recognize that Listing 1 has a defect while the Listing 2 does not.

The deep learning Long Short-Term Memory (LSTM) model [8] potentially offers a powerful alternative to software metrics and BoW in representing software code. LSTM is currently the key technology behind many recent breakthroughs in machine translation and speech recognition [13]. The syntax and different levels of semantics in source code are usually represented by tree-based structures such as Abstract Syntax Trees (ASTs). Thus a tree-structured LSTM network [9] has the potential to learn the following properties in source code.

Syntactic and semantic structure: The two code listings are different in their structure and thus would behave differently. The location of the *if statement* makes a significant difference in causing or removing a defect. Syntactic structure also requires pairs of code elements to appear together, but these dependent code elements may scatter far apart (e.g., *try* and *catch* may be separated by many lines of code). LSTM has been proven to be highly effective in learning such long-term dependencies. In addition, code elements are not always required to follow a specific order, e.g. in code listing 1, lines 5 and 6 can be swapped without changing the code's behaviour. A tree-based LSTM can cater for this flexibility.

Semantic code tokens: Code elements have their own semantics. For example, in Java "for" and "while" are semantically similar, e.g. the while loop in the above code listings can be replaced with a for loop without changing the code behaviour. Existing approaches (e.g. [7, 14]) often overlook those semantics of code tokens. By contrast, LSTM offers the ability to automatically learn a vector representation of code elements that reflect their semantic.

Motivated by the above observations, we have developed a tree-based LSTM model which aims to generate useful defect-predicting features. In the following sections, we will discuss how this model was built, trained and tested, and the lessons we learned from applying it into practice.

III. Model building

Most existing work in large-scale software defect prediction focuses on determining whether a *source file* is likely to be defective or not. This level of granularity has become the standard in the literature of software defect prediction [1]. Once a likely defective file is located, more precise local defect identification mechanisms, including testing and inspection, can be used to identify specific statement-level defects. Determining if a source file is defective can be considered as a function $predict(f)$ which takes as input a file f and returns either 1 for defective and 0 for clean. We approximate this classification function $predict(x)$ (or also referred to as the model) by learning from a number of examples (i.e. files known to be defective or clean) provided in a *training set*. After training, the learned function is used to automatically determine the defectiveness of new files in the same project

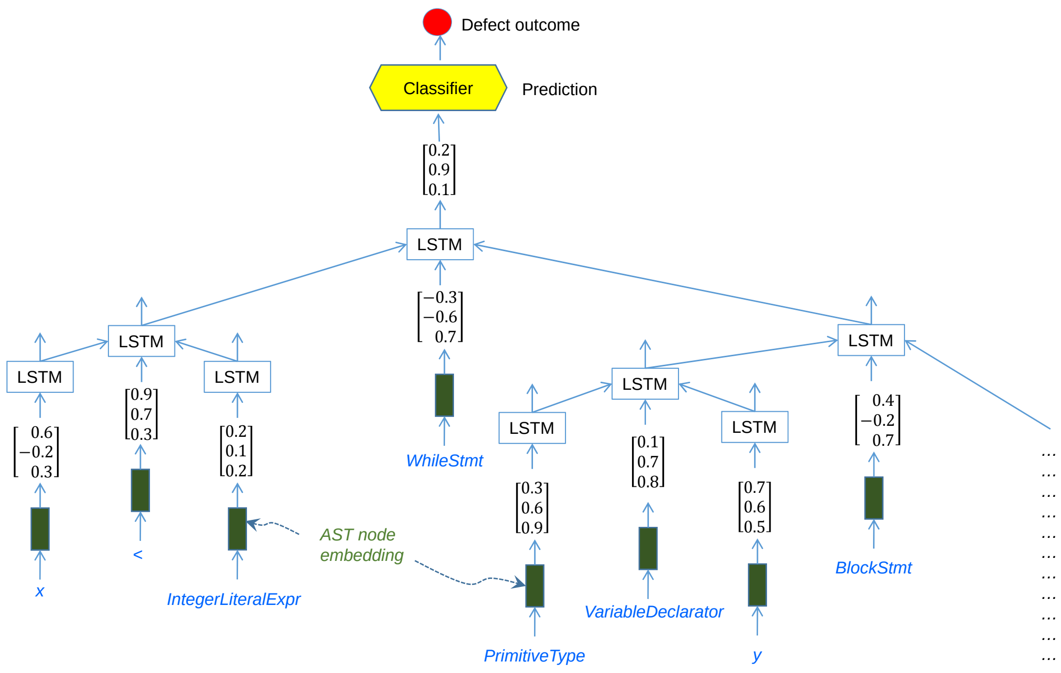

Fig. 2. An example of how our tree-based LSTM model is used for defect prediction

(within-project prediction) or in a different project (cross-project prediction).

Numerous machine learning techniques have been widely used to learn function $predict(x)$. Since machine learning algorithms need input that are mathematically and computationally convenient, file x is often represented as a n-dimensional vector where each dimension represents a feature (or predictor). The feature vector representation of file x affects the accuracy of a defect prediction model.

Our prediction model uses a tree-structured network of LSTM units (Tree-LSTM) [9] to automatically learn a feature vector representation of source file x. The key steps of our approach (see Figure 2) is as below.

1) Parse a source code file into an Abstract Syntax Tree (see Section III-A for details).
2) Map AST nodes to continuous-valued vectors called embeddings (Section III-B).
3) Input the AST embeddings to a tree-based network of LSTMs to obtain a vector representation of the whole source file. This vector is then used by a classifier to predict defect outcomes (Section III-C).

A. *Parsing source code*

We parse each source code file into an Abstract Syntax Tree (AST). This process ignores comments, blank lines, punctuation and delimiters (e.g. braces, semicolons, and parentheses). Each node of the AST represents a construct occurring in the source code. For example, the root of the AST represents a whole source file, and its children are all the top elements of the file such as import and class declarations. Each class declaration node (i.e. `ClassOrInterfaceDeclaration`) has multiple children nodes which represent the fields (`FieldDeclaration`) or the methods (`MethodDeclaration`) of the class. A method declaration node also has multiple child nodes, representing name, arguments, return type, and body.

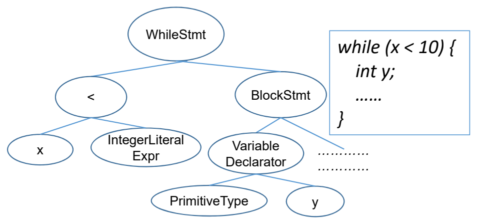

Fig. 3. An example of an Abstract Syntax Tree (AST) for a Java program

We label each tree node with its AST type (e.g. `FieldDeclaration`, `MethodDeclaration`, `BlockStmt`, and `WhileStmt`) or its AST name (e.g. variable name, class name, and method name) in the case of SimpleName nodes (see Figure 3). Constant integers, real numbers, exponential notation, hexadecimal numbers and strings are represented as AST nodes of their type (rather than the actual number or string) since they are specific to a method or class. For example, integer 10 is represented as an `IntegerLiteralExpr` node (Figure 3), while string “Hello World” is represented as a `StringLiteralExpr`.

The unique label names collected from all AST tree nodes in the entire corpus are used to form a vocabulary. Following standard practice, we also replace less popular tokens (e.g. occurring only once in the corpus) and tokens which exist in test sets but do not exist in the training set with a special token $\langle unk \rangle$. A fixed-size vocabulary $\mathscr{V}$ is constructed based on top N popular tokens, and rare tokens are assigned to $\langle unk \rangle$. Doing this makes our corpus compact but still provides partial semantic information.

B. Embedding AST nodes

Each AST node is input to an LSTM unit. Since the LSTM unit only takes input in the form of vectors, we map the label name of each AST node into a fixed-length continuous-valued vector. We refer to this embedding process as *ast2vec*.

This process makes use of an embedding matrix $\mathcal{M} \in \mathbb{R}^{d \times |\mathscr{V}|}$ where d is the size of an AST node embedding vector and $|\mathscr{V}|$ is the size of vocabulary $\mathscr{V}$. Each AST node label has an index in the vocabulary (i.e. encoded as one-hot vector). The embedding matrix acts as a look-up table: an AST node label i^{th} is mapped to column vector i^{th} in matrix $\mathcal{M}$. For example in Figure 2, a `WhileStmt` node is embedded in vector $[-0.3, -0.6, 0.7]$, while `IntegerLiteralExpr` is mapped to vector $[0.2, 0.1, 0.2]$. The embedding process offers two benefits. First, an embedding vector has lower dimensions than a one-hot vector (i.e. $d < |\mathscr{V}|$)). Second, in the embedding space, AST nodes that frequently appear in similar context are close to each other. This often leads to code elements with similar semantic being neighbours. For example, the embeddings of `WhileStmt` and `ForStmt` would be close to each other in the embedding space. The embedding matrix is randomly initialized, and then is adjusted as part of the training process, which we will discussed in Section IV.

C. Defect prediction model

Our prediction model is represented as function $predict()$ which takes as input a source file and returns 1 if the file is defective and 0 otherwise (see Algorithm 1). It first parses the source file into an Abstract Syntax Tree (line 2 in Algorithm 1). The root of the AST is fed into a Tree-LSTM unit to obtain a vector representation $\boldsymbol{h}_{root}$ (line 3). This vector is fed into to a traditional classifier to compute the probability of the file being defective. If this probability is not smaller than 0.5, the function returns 1. Otherwise, it returns 0 (lines 4–6).

A Tree-LSTM unit (see Figure 4) is modeled as function $t\text{-}lstm()$, which takes as input an AST node t and outputs two vectors: $\boldsymbol{h}$ (representing the hidden output state) and $\boldsymbol{c}$ (representing the context it remembers so far in the AST). This is done by aggregating those outputs from the descendants, i.e. calling $t\text{-}lstm()$ recursively on the children nodes (lines 11–26). This function first obtains the embedding $\boldsymbol{w}_t$ of the input AST node t (using $ast2vec$ as discussed in Section III-B). It then obtains all the children node $C(t)$ of node t, and each child node $k \in C(t)$ is fed into an LSTM unit to obtain the pair of hidden output state and context vectors $(\boldsymbol{h}_k, \boldsymbol{c}_k)$ for each child node. These are used to compute the pair of hidden output state and context vectors $(\boldsymbol{h}_t, \boldsymbol{c}_t)$ for the parent node.

How information embedded in $\boldsymbol{w}_t$ and $(\boldsymbol{h}_k, \boldsymbol{c}_k)$ (for all $k \in C(t)$) flows through a Tree-LSTM unit is controlled by three important components: an input gate (represented as $\boldsymbol{i}_t$), an output gates ($\boldsymbol{o}_t$) and a number of forget gates (one $\boldsymbol{f}_{tk}$ for each child node k). These components depend on the input $\boldsymbol{w}_t$ and the output state $\boldsymbol{h}_k$ of the children. These correlations are encoded in groups of parameter matrices: $(W_{for}, U_{for}, b_{for})$ for the forget gates, (W_{in}, U_{in}, b_{in}) for the input gate, and $(W_{out}, U_{out}, \boldsymbol{b}_{out})$ for the output gates.

Algorithm 1 Tree-based defect prediction. Model parameters include $(W_{for}, U_{for}, b_{for})$, (W_{in}, U_{in}, b_{in}), (W_{ce}, U_{ce}, b_{ce}), and $(W_{out}, U_{out}, \boldsymbol{b}_{out})$ shared by all Tree-LSTM units.

```
1:  function PREDICT(File f)
2:      $root \leftarrow parseFile2AST(f)$
3:      $(\boldsymbol{h}_{root}, \boldsymbol{c}_{root}) \leftarrow t\text{-}lstm\,(root)$
4:      $\hat{p} \leftarrow classifier\,(\boldsymbol{h}_{root})$
5:      if $\hat{p} \geq 0.5$ then
6:          return $1$
7:      else
8:          return $0$
9:      end if
10: end function

11: function T-LSTM(ASTnode t)
12:     $\boldsymbol{w}_t \leftarrow ast2vec(getNodeName(t))$
13:     $C(t) \leftarrow getChildrenNodes(t)$
14:     $(\boldsymbol{h}_k, \boldsymbol{c}_k) \leftarrow (\vec{0}, \vec{0})$
15:     for all $ASTNode\ k \in C(t)$ do
16:         $(\boldsymbol{h}_k, \boldsymbol{c}_k) \leftarrow t\text{-}lstm(k)$
17:         $\boldsymbol{f}_{tk} = sigmoid\,(W_{for}\boldsymbol{w}_t + U_{for}\boldsymbol{h}_k + b_{for})$
18:     end for
19:     $\tilde{\boldsymbol{h}} \leftarrow \sum_{k \in C(t)} \boldsymbol{h}_k$
20:     $\boldsymbol{i}_t \leftarrow sigmoid\left(W_{in}\boldsymbol{w}_t + U_{in}\tilde{\boldsymbol{h}} + b_{in}\right)$
21:     $\tilde{\boldsymbol{c}}_t \leftarrow tanh\left(W_{ce}\boldsymbol{w}_t + U_{ce}\tilde{\boldsymbol{h}} + b_{ce}\right)$
22:     $\boldsymbol{c}_t = \boldsymbol{i}_k * \tilde{\boldsymbol{c}}_t + \sum_{k \in C(t)} \boldsymbol{f}_{tk} * \boldsymbol{c}_k$
23:     $\boldsymbol{o}_t = sigmoid\left(W_{out}\boldsymbol{w}_t + U_{out}\tilde{\boldsymbol{h}} + b_{out}\right)$
24:     $\boldsymbol{h}_t = \boldsymbol{o}_t * tanh\,(\boldsymbol{c}_t)$
25:     return $(\boldsymbol{h}_t, \boldsymbol{c}_t)$
26: end function
```

A Tree-LSTM unit has a number of forget gates $\boldsymbol{f}_{tk}$, one for each child node k and is computed as a sigmoid function over $\boldsymbol{w}_t$ and $\boldsymbol{h}_k$ (line 17). A forget gate $\boldsymbol{f}_{tk}$ has a value between 0 and 1, which enables the Tree-LSTM unit to selectively include information from each child. The output from children nodes are combined to serve as an input the the parent LSTM unit (line 19). How much of these new information is stored in the memory cell is controlled by two mechanisms (lines 20–22). First, the input gate $\boldsymbol{i}_k$, represented as a sigmoid function, decides which values will be updated. Second, a vector of new candidate values $\tilde{\boldsymbol{c}}_t$, which will be added to the memory cell, is created using a $tanh$ function.

The new memory is updated by multiplying the old memory of each child by $\boldsymbol{f}_{tk}$, leaving out the things we decided to forget earlier. We sum it over all the child node and then add this with $\tilde{\boldsymbol{c}}_t$. Finally, the output is a filtered version of the memory, which is controlled by the output gate $\boldsymbol{o}_t$ (line 23).

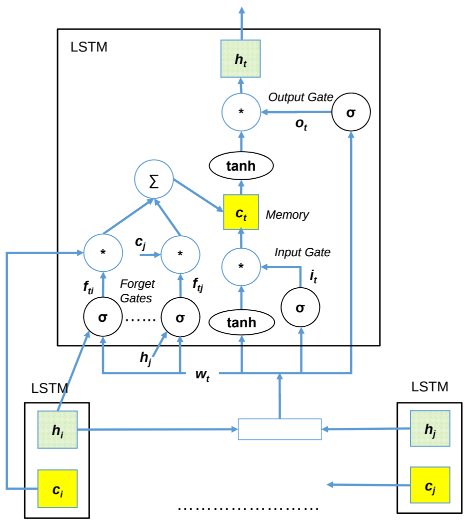

Fig. 4. The internal structure of a Tree-LSTM unit

We apply $tanh$ function to the memory (to scale the values to be between -1 and 1) and multiply it by the output of the sigmoid gate so that only selected parts are output (line 24).

IV. Model Implementation and Training

A. Training Tree-LSTM

We train the Tree-LSTM unit in an unsupervised manner, i.e. *not* using the ground-truth defect labels. We leverage the strong predictiveness of AST, i.e. if we know the label name of all the children, we can predict the label name of its parent. Using a large number of AST branches, we train the Tree-LSTM unit through making such a prediction. For example, the parent of “<” and ”VariableDeclarator” is “WhileStmt”, while the parent of “x” and “IntegerLiteralExpr” is “<” (see Figure 5). Specifically, each AST node w_t has a set of children $C(t)$, and each $c_k \in C(t)$ has an output state h_k. We can predict the label name of the parent node using all its children hidden states through the softmax function.

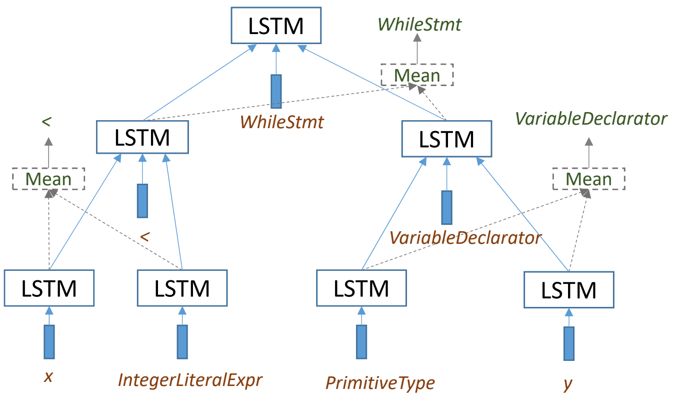

Fig. 5. Training Tree-LSTM by predicting the label name of a parent node from its children nodes

Let θ be the set of all parameters in the LSTM unit, which includes the embedding matrix $\mathcal{M}$ and weight matrices $(W_{for}, U_{for}, b_{for})$, (W_{in}, U_{in}, b_{in}), (W_{ce}, U_{ce}, b_{ce}), and $(W_{out}, U_{out}, \boldsymbol{b}_{out})$. These parameters are initialized randomly and then learned through a training process. Training involves three main steps: (i) input an AST branch in the training data to the LSTM units to obtain a prediction for the label name of the parent node in that branch; (ii) compare the difference δ between the predicted outcome and the actual outcome; (iii) adjusting the values of the model parameters such that the difference δ is minimized. This process is done iteratively for all files in the training data.

To measure the quality of a specific set of values for the model parameters, we define a loss function $L(\theta)$ which is based on the difference δ between the predicted outcome and the actual outcome. A setting of the model parameters θ that produces a correct prediction (e.g. the label name of a parent node is correctly predicted) would have a very low loss L. Hence, learning is achieved through the optimization process of finding the set of parameters θ that minimizes the loss function. We use the popular cross-entropy loss which measures the information-theoretical distance empirical distribution of the true outcome and the softmax distribution computed by the model.

Since every component in the model is differentiable, we employ the widely-used stochastic gradient descent to perform optimization. The optimization process is done through backpropagation: the model parameters θ are updated in the opposite direction of the gradient of the loss function $L(\theta)$. A learning rate η is used to control how fast or slow we will move towards the optimal parameters. We used RMSprop, an adaptive stochastic gradient method, and implemented *dropout* [15] into our model, an effective mechanism to prevent overfitting in neural networks.

We implemented the model in Theano [16] and Keras [17] frameworks, running in Python. Theano supports automatic differentiation of the loss function and a host of powerful adaptive gradient descent methods. Keras is a wrapper making model building much easier. We use the standard learning rate of 0.02, and smoothing hyper-parameters: $\rho = 0.99$, and $\epsilon = 1e - 7$. The model parameters are updated in a stochastic fashion, i.e. after every mini-batch of size 50. We use $|\mathscr{V}| = 5,000$ most frequent tokens for the vocabulary. We use *dropout* rate of 0.5 at the hidden output of LSTM layer. These parameter settings are the standard ones used in the literature. We experimented with different embedding sizes: 32, 64 and 128. We employed Noise-Contrastive Estimation [18] to compute the softmax function since it has a fixed time complexity regardless of the vocabulary size. We also run multiple epoches against a validation set to choose the best model. We use *perplexity*, a common intrinsic evaluation metric based on the log-loss, as a criterion for choosing the best model and early stopping.

B. Training defect prediction model

The above process enables us to automatically generate features for all the source files in the training set. These files with their features and labels (i.e. defective or clean) are then used to train machine learning classifiers by learning from a number of examples (i.e. files known to be defective or clean) provided in a *training set*. We tried two alternative classifiers: Logistic Regression and Random Forests. Logistic Regression uses the logistic function (also called the sigmoid function) to approximate the probability of a source file being defective given its AST feature vector representation. Random Forests (RFs) is a randomized ensemble method which combines the estimates from many decision trees to make a prediction.

V. Experiments

We describe a number of experiments conducted to evaluate our defect prediction model using both industrial and benchmark examples, and feedback from Samsung engineers.

A. *Experimental design and datasets*

Static analysis tools have been routinely used by many software companies as part of the software quality assurance process. Most of them are designed to be used after the code is fully completed and run in batch mode since running those tools against an entire codebase often takes a long time. Using a defect prediction model does not require a full completion of the code, enabling early identification of defective modules in a codebase, e.g. before other analyses are applied. Those modules then receive priority attention when subsequently static analysis, testing, and manual inspection are applied to locate the defects. Defect prediction models are therefore able to help prioritize effort and optimize inspection and testing costs.

We thus design two experiment settings to evaluate our approach. In the first experiment, we evaluate if our prediction model correctly identifies the modules which contain defects discovered by a static analysis tool. This experiment uses the dataset containing open source projects contributed by Samsung and the source files were labelled using reports from a static analysis tool used at Samsung. Since static analysis tools may generate false-positive alerts, we have also conducted a second experiment using the PROMISE dataset [19] where the files were labelled using information from bug reports and code patches. We now describe these two datasets in more details.

1) ***Open source projects contributed by Samsung****:* There are many open source projects contributed by Samsung Electronics such as Tizen, an open source operating system. Tizen runs on a wide range of Samsung devices including smartphones, tablets, in-vehicle infotainment devices, smart TVs, smart cameras, smart watches, and smart home appliances. We collected potential defects from those open source projects. To identify defective files, we employed a static analysis tool[1] used by Samsung that has specific support for target projects. This tool scans the source code of those projects and generates a report describing all the potential defects (i.e. warnings) that it can discover.

There are different types and severity levels of warnings reported by the tool. In this study, we focused on critical resource leakage warnings (e.g. a handle was created but lost without releasing it). We used this information to label files as defective or clean: a file is considered defective if the tool reported at least one resource leakage warning associated with that file. We built up a dataset of 8,118 files written in C, 2,887 of which (35.6%) are labelled as defective and 5,231 (64.4%) labelled as clean.

2) ***PROMISE dataset****:* We also used a dataset for defect prediction which is publicly available from the PROMISE data repository [19]. To facilitate comparison, we selected the same 10 Java projects and release versions from this dataset as in [7]. These projects cover a diversity of application domains such as XML parser, text editor, enterprise integration framework, and text search engine library (see Table I). The provided dataset only contained the project names, their release versions, and the file names and their defective labels. It did not have the source code for the files, which is needed for our study. Using the provided file names and version numbers, we then retrieved the relevant source files from the code repository of each application.

TABLE I
Dataset statistics

App	#Versions	#Files	Mean files	Mean LOC	Mean defective	% Defective
lucene	3	750	250	47091	145	57.18
synapse	3	635	211	30442	54	23.60
xerces	2	891	445	132934	70	15.72
camel	3	2379	793	81183	183	24.54
xalan	2	1438	719	256625	248	33.53
ivy	2	593	296	44288	28	9.00
ant	3	1383	461	123452	96	19.88
jedit	3	853	284	94696	81	28.85
poi	3	1053	351	87611	223	63.14
log4j	2	223	111	16979	35	32.07

When processing the CSV spreadsheets provided with the PROMISE dataset, we have found that there were entries for inner classes. Since inner classes are included in an AST of their parent, we removed those entries from our dataset. We also removed entries for source files written in Scalar and entries that we could not retrieve the corresponding source files. In total, 264 entries were removed from the CSV spreadsheet. Table I provides some descriptive statistics.

B. *Performance measures*

Reporting the average of precision/recall across the two classes (defective and clean) is likely to overestimate the true performance, since our dataset is imbalanced (i.e. the number of defective files are small). More importantly, predicting defective files is of more of interest than predicting clean files. Hence, our evaluation focuses on the defective class.

A confusion matrix is used to store the correct and incorrect decisions made by a prediction model. The values stored in the confusion matrix are used to compute the widely-used Precision, Recall, and F-measure of the defective class. In addition, we also use the Area Under the ROC Curve (AUC) to evaluate the degree of discrimination achieved by the model. The value of AUC is ranged from 0 to 1 and random prediction

[1]Name is not revealed due to non-disclosure agreement.

has AUC of 0.5. The advantage of AUC is that it is insensitive to decision threshold like precision and recall.

C. Results

1) Within-project prediction: This experiment[2] used data from the same project for both training and testing but training and testing files are non-overlapping. For the Samsung dataset, we could not trace back which project a source file belonged to, and thus we treated all the source files in the dataset as belonging to a single project. We employed cross-fold validation and divided the files in this dataset into ten folds, each of which have the approximately same ratio between defective files and clean files (also known as stratified sampling). Each fold is used as the test set and the remaining folds are used for training. As a result, we built ten different prediction models and the performance indicators are averaged out of the ten folds. We also tested with two different classifiers: Random Forests and Logistic Regression. Figure 6 shows the predictive performance of our approach for the Samsung dataset. The predictive model which uses Random Forests (RF) as the classifier produced an impressive result with all four performance indicators (F-measure, Precision, Recall and AUC) being well above 0.9. Using Logistic Regression (LR) achieved very high recall, but at the same time it appeared to produce many false positives, and thus its precision is much lower than the precision produced by RF. Both classifiers achieved an AUC well above the 0.5 threshold (0.98 for RF and 0.60 for RF), suggesting that our approach is significantly better than random prediction.

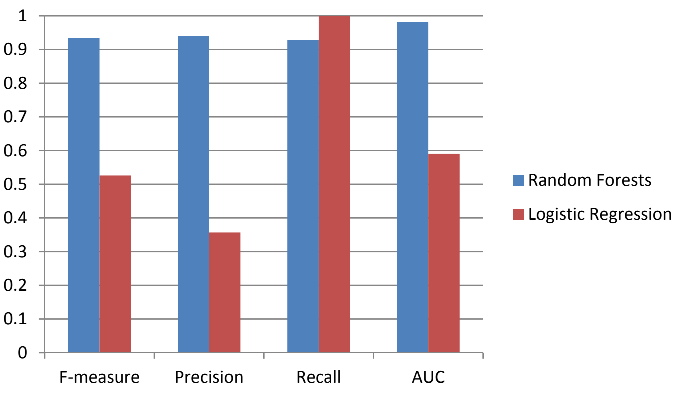

Fig. 6. Predictive performance of our approach for the Samsung dataset

For the PROMISE dataset, since it contains different versions of the same applications, we followed the setting in Wang *et. al.* [7] and used two consecutive versions of each project for training and testing. Specifically, the source code of an older version is used to training the model and the later version is used for testing the model. In total, we conducted 16 sets of experiments exactly as in Wang *et. al.*. We also tested with Random Forests and Logistic Regression as the classifier, and observed a different result (compared to the result for the Samsung dataset): using LR produced better predictive performance than using RF. This can be explained by the fact that the PROMISE dataset has small number of data points, which fits better with LR.

Figure 7 shows the results from using LR as the classifier. Our prediction model produced an average AUC of 0.6, well above the random prediction threshold. More importantly, it achieved a very good recall of 0.86 (averaging across 16 cases), which is 23% improvement over Wang *et. al.*'s approach. However, our approach has lower precision, leading to a deduction in F-measure (17%) compared against Wang *et. al.*'s approach. High recall is preferable in predicting defects since the cost of missing defects is much higher than having false positives.

2) Cross-project prediction: Predicting defects in new projects is often difficult due to lack of training data. One common technique to address this problem is training a model using data from a (source) project, and applying it to the new (target) project. We conducted this experiment by selecting one version from a project in our PROMISE dataset as the source project (e.g. ant 1.6) and one version from another project as the target project (e.g. camel 1.4). Figure 8 summarizes the results in cross-project prediction for the twenty-two pairs of source and target Java projects.

Our approach again achieved very high recall, with an average of 0.8 across 22 cases in cross-project prediction. There are 15 cases where the recall was above 0.8. The average F-measure is however 0.5, due to the low precision as seen in within-project prediction. However, the average AUC is still well above the 0.5 threshold, demonstrating the overall effectiveness of our approach in predicting defects.

D. Feedback from Samsung engineers

The good performance of our defect prediction tool on the Samsung projects was well received by its engineers. The recommended defect-containing files generally corresponded well to the defective parts of the codebase. The software engineers found having a tool to locate likely defective files in a large codebase very helpful in narrowing down detailed code analysis, prioritising warnings to view, prioritising and focusing whitebox inspections, and complimenting their existing toolset.

A major concern expressed was the "black box" nature of the recommendation. As our tool uses deep learning it produces a set of likely defective files but no explanation about what the defects are nor precise locations of defects for larger files. Some source code files may have many hundreds to thousands lines of code. Hence, without a sufficient explanation, the software engineers found it difficult to understand why the prediction model suggests that a particular source file is defective or clean. Thus, they hesitated to trust the model's predictions, especially when the predictions were different from their expectation, e.g. flagging files that they expected to be "clean". Understanding the reasons why the model predicts a file is defective would also help the software engineers

[2]All experiments were run on Intel(R) Xeon(R) CPU E5-2670 0 @ 2.6GHz. There machine has two CPUs, each has 8 physical cores or 16 threads, with a RAM of 128GB.

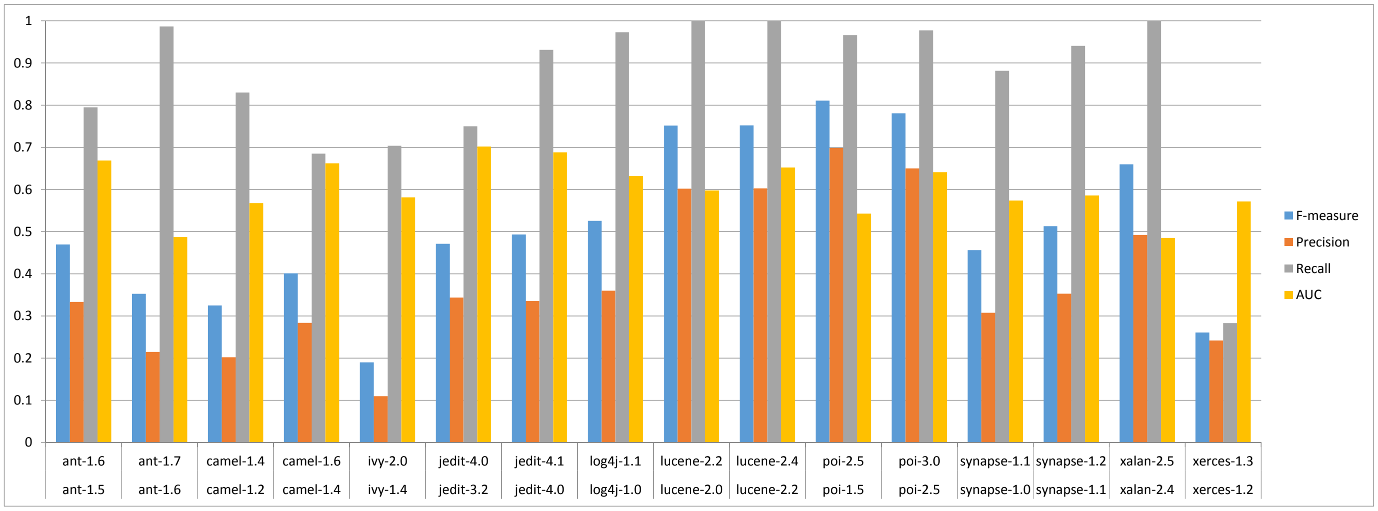

Fig. 7. Predictive performance of our approach for the PROMISE dataset (within-project prediction). The X-axis has pairs of training (lower version) and testing data (the newer version) in each project. For example, in the first pair our model was trained using version 1.5 of Apache Ant project and tested using its version 1.6.

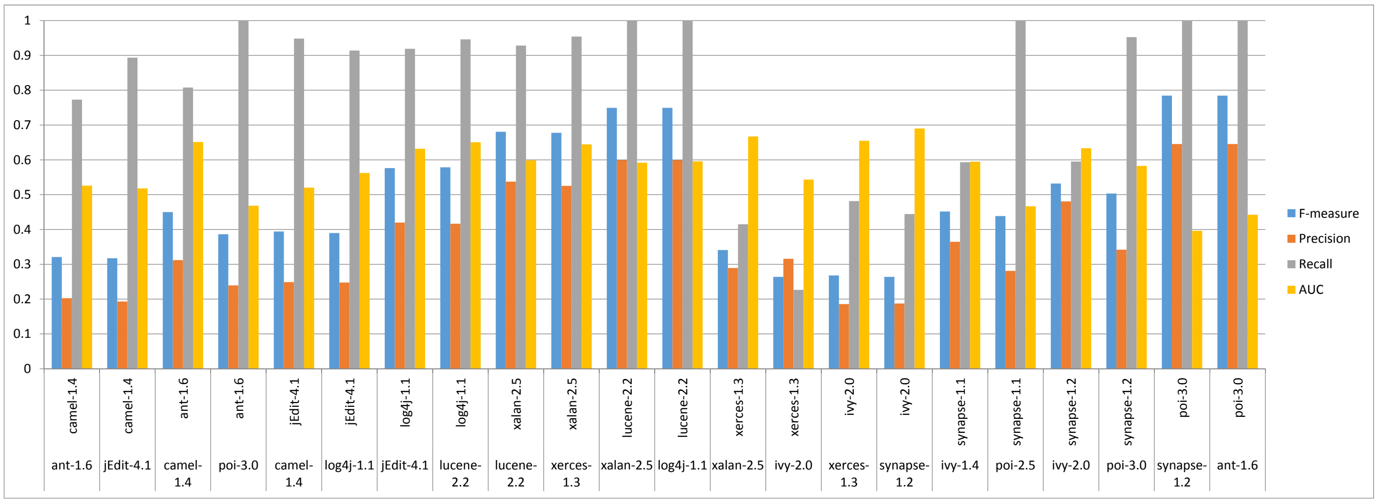

Fig. 8. Predictive performance of our approach for the PROMISE dataset (cross-project prediction). The X-axis has pairs of training (source project) and testing data (target project). For example, in the first pair our model was trained using version 1.6 of Apache Ant project and tested using version 1.4 of the Camel project.

generate fixes. The overall feedback was that merely flagging a file as defective is often not sufficiently useful. In Section VI, we will discuss this and the other lessons learned in more details.

E. Threats to validity

We mitigated construct validity concerns by evaluating our approach not just only on Samsung datasets but also on a publicly available dataset (the PROMISE dataset). Both datasets contains real projects. The PROMISE dataset did not unfortunately contain the source files. However, we have carefully used the information (e.g. application details, version numbers and date) provided with the dataset to retrieve the relevant source files from the code repository of those applications. We tried to minimize threats to conclusion validity by using standard performance measures for defect prediction. We however acknowledge that a number of statistical tests can be applied to verify the statistical significance of our conclusions, which we plan to do in our future work.

With regard to internal validity, the Samsung dataset we used contains defective labels which were derived from warnings provided by a static analysis tool used internally at Samsung. We acknowledge that those warnings may contain false positives, and thus future work would involve investigating those warnings and confirming their validity. In addition, we did not have the source code to replicate Wang *et. al.*'s experiments [7], and thus had to rely on the results they reported to make a comparison with our approach. In terms of external validity, we have considered a large number of applications which differ significantly in programming language, size, complexity, domain, popularity and revision history. We however acknowledge that our data set may not be representative of all kinds of software applications, and further investigation is need to confirm our findings for other types of applications.

VI. Lessons learned

The development of defect prediction models in the context with our industry partner has provided us with valuable experiences and insight. In this section, we will discuss the key lessons that we have learned and also propose potential solutions that are useful for future research in this area.

A. *Lesson #1: Explainability*

Although the predictive performance of our model is high (especially for the Samsung dataset), we found that the software engineers were still reluctant to adopt our defect prediction model into their day-to-day work. The major reason was the limited explainability in our model. One potential solution is predicting at a more fine-grained level. For example, predicting if a line of code or several lines of code are defective. Software engineers may then find it easier to inspect a line of code (compared to a source file) and form their own understanding of the model's predictions. Our current model can be extended to operate at this fine-grained level of granularity. For example, a line of code can be parsed into an AST, which can be input into function *t-lstm(ASTnode t)* in Algorithm 1 to derive features for defect prediction. In a similar manner, our model can also be extended to support Just-In-Time (JIT) quality assurance by performing predictions at the change level (e.g. commits). The JIT defect prediction model can be invoked as soon as the software engineers commit their code, which helps them identify the defects early. To do so, we need to train the model with a new dataset where code lines are labelled as defective or clean. Accurate prediction at the code line level is however challenging since defects usually arise from interactions between multiple sometimes highly distributed code statements.

Another solution is keeping the prediction at the file level and designing new architectures that self-explain decision making at each step. For example, a popular strategy in neural networks is using *attention* [20] where model components compete to contribute to an outcome. This mechanism can be implemented into our tree-based LSTM network so that "attention" can be distributed from the parent node to the children nodes, and to their own children and so on. This mechanism enables our model to locate the parts (e.g. code lines) in a source file that are likely the cause of a defect. This helps understand and diagnose exactly what the model is considering and to what degree for specific defects.

Alternatively, we might employ an interpretable model to explain the behaviours of the complex neural network using an interpretable model. LIME [21] can be used to derive an explanation about a data instance locally. For example, given the input AST in Figure 3, we can locally change it by (e.g.) removing a node, and observe the behaviour of the network. If behaviour changes, we can infer a node is a relevant feature.

B. *Lesson #2: Training time*

Once our model has been trained, it can quickly generate the features (in the order of milliseconds). If all the source files in a codebase have been parsed into ASTs, the trained model can be fairly fast in making predictions and is able to scale to large codebases. The bottleneck is training the Tree-LSTM model (see Section IV-A) which may take a long time, from a few hours to a few days, depending on the number of ASTs in the training set, the size of those ASTs, and a number of hyper-parameters such as the vocabulary size and the embedding vector size. Local learners such as those proposed in [22] may help reduce training time, but they still rely on vector representations of source code which are obtained from other means (including deep learning methods like our model). Although model training is done offline, it is important to beware of the long training time in using deep learning methods, especially if fine tuning a model (e.g. hyper-parameter optimization) requires many repeated training runs.

C. *Lesson #3: Vocabulary*

We have followed traditional NLP approaches and fixed the size of the vocabulary (see section III-A). A fixed-size vocabulary V is constructed based on the top number of popular tokens. Code tokens outside the vocabulary are assigned a special $\langle unk \rangle$ token. This technique is effective in NLP since new words outside a defined vocabulary is rarely found. This technique however does not seem to work well with source code. Although the keywords and operators are finite with respect to each programming language, new identifier names (e.g. variables) are constantly introduced by the developers. This leads to a large number of unique tokens. A small vocabulary size has adverse impact on the predictive performance of the model. Increasing the size of the vocabulary will increase the model training time and computational resources since the models need to tune the parameters for each word.

There are a few solutions which can be adopted here. First, we can develop a character-level model, which is increasingly popular in NLP to handle text with rare words (such as proper names and scientific names). Since the number of characters are small, the vocabulary size will be small. Second, rather than replacing the tokens outside the vocabulary with $\langle unk \rangle$, we can replace it with some other meaningful in-vocabulary tokens (e.g. the type of the identifier). The third solution is to rename variables using certain rules, e.g. *var1-context2*, where $var1$ can be "first temporary variable of type int", $context2$ can be "within a for-loop". Doing this will convert all variable names into a fixed set of name-templates. The fourth solution is describing the variables in terms of the descriptors of the contexts in which they occur, then performing clustering to assign the variable to the nearest cluster ID. A combination of these techniques are also applicable.

D. *Lesson #4: Tree size*

A challenge is the high degree of variation in the number of children per parent in ASTs. While most of the nodes has around 2–5 children, some may have up to 200 children nodes.

This results in an excessive computing resources since we need to pre-define the tensor size (which depends on the maximum number of children) to effectively exploit the capacity of GPU. We therefore convert the ASTs to binary trees to create a better balance tree. Doing this however results in an increase in the number of tree nodes (300 nodes per tree on average) and our model does not scale well with the number of tree nodes. Our future work will investigate how to overcome this issue.

E. Lesson #5: End-to-end model

Our current model has two separate steps: learning the features representing source code files then using these to build a classifier (e.g. Random Forests) for defect prediction. An alternative approach is building an end-to-end model where the two steps can be done at the same time. This can be achieved by substituting the final classifier (see Figure 2) with a simple feedforward neural network. In that setting, the whole system is trained in an end-to-end manner, i.e. defective outcomes are used to train the LSTM models. We have experimented this approach, however its predictive performance was worse than the current model and it took even more time for training. We have found that end-to-end training usually requires significantly large amounts of labelled data and this may not be suitable for defect prediction where the labelled data is relatively limited.

F. Lesson #6: Heterogeneity of codebases

We also learned from this project that codebases are sometimes heterogenous, e.g. some parts of the codebase were written in C, while others were written in C++. As long as we can parse a source code file into an AST, it can be input into our model, regardless of which programming language it was written in. This has been demonstrated in our evaluation on two different datasets containing applications in C and Java.

VII. Related Work

A. Defect prediction

Defect prediction is a very active area in software analytics. Since defect prediction is a broad area, we highlight some of the major work here, and refer the readers to other comprehensive reviews (e.g. [2, 23]) for more details. Code metrics were commonly used as features for building defect prediction models (e.g. [3]). Various other metrics have also been employed such as change-related metrics [10], developer-related metrics [4], organization metrics [12], and change process metrics [5].

Recently, a number of approaches (e.g. [7, 14]) have leveraged a deep learning model called Deep Belief Network (DBN) [24] to automatically learn features for defect prediction and have demonstrated an improvement in predictive performance. In fact, according to the evaluation reported by Wang *et. al.* [7] their DBN approach outperformed both the software metrics and Bag-of-Word approaches. DBN however does not naturally capture the sequential order and long-term dependencies in source code. The work in [25] also builds a vector representation of code from ASTs for bug detection. Unlike our tree-based LSTM model, the classical neural network that they used does not however maintain the structure of the ASTs.

Recent approaches have attempted to predict defects at more fine-grained levels such as method level (e.g. [26]) or line level (e.g. [27]). Our approach can be extend to operate at those at those finer level of granularity since it is technically able to learn features at the code token level and from any arbitrary ASTs. To do this study, we would need to develop new datasets which contain methods and codelines with defect labels, which we leave for future work.

B. Deep learning in code modeling

Deep learning has been emerged as a effective alternative for code modelling (see [28] for an extensive review). Recent work (e.g. [29]) have used recurrent neural networks (RNN) to model source code and use this modeling to perform various software engineering tasks such as detecting code clones (e.g. [30]). Classical LSTM models (not Tree-LSTM) have also been used for code modelling [31] and vulnerability prediction [32]. Convolutional Neural Networks (CNN) [33], another well-known deep learning architecture, has also been adapted for bug localization [34]. Deep learning-based machine translation models (e.g. RNN Encoder–Decoder) have been used for querying API usage sequences [35] and fixing common errors in C programs [36]. Previous work (e.g. [37–40]) have transformed software code into ASTs and program graphs to learn representations. Unlike our approach, their models do not use a tree-structured LSTM which directly matches with the AST representation of source code.

VIII. Conclusions and future work

We have reported our experience in developing and applying new deep learning tree-based model which takes as input an Abstract Syntax Tree (AST) representing a source file, a common representation for source code, and predict if the file is defective or clean. Our prediction system is built upon Long Short-Term Memory (LSTM) architecture to capture the long-term dependencies which often exist between code elements. Our use of the tree-structured LSTM network (Tree-LSTM) naturally matches the AST representation to capture the syntax and different levels of semantics in source code.

An evaluation on two different datasets provided by Samsung and the PROMISE repository and feedback from Samsung engineers provided us with a number of lessons to improve the explainability, scalability and usefulness of our approach in future work. We also plan to apply this approach to other types of applications (e.g. Web) and programming languages (e.g. PHP or C++). We want to extend our approach to predict defects at method and code change levels. In addition, we plan to explore how our approach can be extended to predicting specific types of defects such as security vulnerability and safety-critical hazards in code.

Acknowledgement

The authors gratefully acknowledge support from Samsung through its Global Research Outreach Program.

References

[1] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*, 2016, pp. 33–45.

[2] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9173-9

[3] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.103

[4] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453105

[5] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[6] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[7] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, 2015, pp. 1556–1566.

[10] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[11] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514

[12] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 521–530. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368160

[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[14] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, ser. QRS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 17–26. [Online]. Available: http://dx.doi.org/10.1109/QRS.2015.14

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[16] Theano, "Theano," http://deeplearning.net/software/theano/, Accessed on 01 May 2017.

[17] Keras, "Keras: Deep Learning library for Theano and TensorFlow," https://keras.io/, Accessed on 01 May 2017.

[18] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 307–361, 2012.

[19] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The PROMISE Repository of empirical software engineering data," Jun. 2012. [Online]. Available: http://promisedata.googlecode.com

[20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ICLR*, 2015.

[21] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," *arXiv preprint arXiv:1602.04938*, 2016.

[22] S. Majumder, N. Balaji, K. Brey, W. Fu, and T. Menzies, "500+ times faster than deep learning: A case study exploring faster methods for text mining stackoverflow," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 554–563. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196424

[23] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7346–7354, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2008.10.027

[24] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504 – 507, 2006.

[25] M. Pradel and K. Sen, "Deep learning to find bugs," TU Darmstadt, Technical Report TUD-CS-2017-0295, November 2017.

[26] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '12. New York, NY, USA: ACM, 2012, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/2372251.2372285

[27] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[28] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017. [Online]. Available: http://arxiv.org/abs/1709.06182

[29] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345.

[30] 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*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[31] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," in *Workshop on Naturalness of Software (NL+SE), co-located with the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[32] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering).*

[33] Y. L. Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson, "Advances in neural information processing systems 2," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Handwritten Digit Recognition with a Back-propagation Network, pp. 396–404. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.109279

[34] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 1606–1612. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060832.3060845

[35] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334

[36] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San*

Francisco, California, USA. AAAI Press, 2017, pp. 1345–1351. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[37] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403*, ser. KSEM 2015. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 547–553. [Online]. Available: https://doi.org/10.1007/978-3-319-25159-2_49

[38] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236085

[39] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

[40] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740