TÉCNICO LISBOA

Prioritizing Facebook's Infer Static Analysis Tool Warnings

João Francisco Roberto Martins

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

November 2020

Acknowledgments

Firstly, I would like to express my gratitude to both my advisors Prof. João Ferreira and Prof. Rui Maranhão for their support and guidance in all the stages of my Master thesis. Their patience, insight and high standards throughout provided a strong example for my personal and academic development and helped me achieve the results presented in this thesis.

I am immensely grateful for my family, for their constant support and love throughout my entire course. In particular, to my parents, João Carlos and Maria Elisabete, my sister Ana Catarina, and to all my grandparents, João Francisco and Maria Luísa, and António Roberto and Maria Adélia.

I want to thank all my friends that were present during these five years, for their support and care in endless adventures.

Finally, but most importantly, I want to thank my God, that is most certainly the reason why I am here today. For from Him and through Him and to Him are all things. To Him be glory forever. Amén.

Abstract

Infer is a Static Analysis tool that analyses code statically and returns warnings on the errors and possible bugs. Infer reports these warnings to the developers without a specific order and without an assigned priority. In this work, we focus on the problem concerning the fact that a significant number of these warnings can turn out to be False Positives. It is important to keep control of false positives since they negatively impact users of this kind of tools, making them lose confidence on the tool they are using and wasting time looking at issues that are not real errors. In this work, we perform a review of state-of-the-art Static Analysis tools warning classifying techniques and develop a Neural Language Model based on LSTM networks, capable of successfully modelling Infer's symbolic execution thus allowing the discovery of patterns that lead to the creation of false positive warnings. We evaluate the model on two different case scenarios and evaluate different data preparation routines for the use of LSTM networks, showing that these routines have a substantial impact in classification accuracy. The experimental results show a warning classification accuracy of approximately 86% when analysing the same program over time, which is Infer's most common application scenario. Overall, our study shows that the proposed model is capable of correctly identifying false positive Infer warnings and consequently improve the usability and effectiveness of Infer Static Analysis tool by prioritizing true positives over false positives warnings.

Keywords

Infer; Static Analysis; False Positive Identification; Warning Prioritization; Machine Learning.

Resumo

O Infer é uma ferramenta de Análise Estática que analisa código estaticamente e retorna avisos sobre erros e possíveis bugs. O Infer devolve estes avisos aos programadores sem uma ordem especifica e sem atribuir a cada aviso um nível de prioridade. Neste trabalho, focamos-nos no problema relativo ao facto de um número significativo destes avisos possam ser falsos positivos. É importante manter o controlo sobre o número de falsos positivos pelo facto de que eles podem influenciar negativamente os utilizadores deste tipo de ferramentas, fazendo-os perder confiança na própria ferramenta que estão a usar e perder tempo na análise de avisos que na realidade não são erros. Neste trabalho, fazemos uma revisão do estado-da-arte relativo à classificação de avisos provenientes de ferramentas de análise estática e desenvolvemos um Modelo de Linguagem Neuronal, baseado em redes LSTM, capaz de modelar com sucesso o resultado da execução simbólica do Infer, permitindo por sua vez a deteção de padrões que conduzem à origem de avisos que são falsos positivos. Avaliamos este modelo em dois cenários diferentes e considera-se o uso de diferentes técnicas de preparação de dados para o uso de redes LSTM mostrando que esta preparação tem um impacto significativo na precisão da classificação. Os resultados experimentais mostram uma precisão de classificação de aproximadamente 86% quando lidamos com o caso mais comum em que o Infer é usado, isto é, quando é usado para analisar o mesmo programa ao longo do tempo. Em geral, este estudo mostra que o modelo proposto é capaz de identificar corretamente avisos do Infer que são falsos positivos e consequentemente capaz de melhorar a usabilidade e a eficácia da ferramenta de análise estática Infer ao priorizar os avisos que são verdadeiros positivos.

Palavras Chave

Infer; Análise Estática; Identificação de Falsos Positivos; Priorização de Avisos; Aprendizagem.

Contents

List of Figures

List of Tables

Acronyms

SA	Static Analysis
ACM	Association for Computing Machinery
FP	False Positives
RNN	Recurrent Neural Network
RNNs	Recurrent Neural Networks
LSTM	Long Short Term Memory
NLP	Natural Language Processing
AST	Abstract Syntax Trees
NLM	Neural Language Modeling
LM	Language Modeling
NLP	Natural Language Processing
IR	Information Retrieval
CI	Continuous Integration
LOC	Lines of Code
SIQR	Semi-interquartile Range

1

Introduction

Contents

Nowadays, the majority of software development companies take advantage of automated bug finding tools to ensure that their products achieve a certain level of quality. However, given the size of real-world systems and its complexity, the number of errors identified can grow too large to properly be considered and addressed by the developers.

Infer is a Static Analysis (SA) tool that analyses code statically and returns warnings on the errors and possible bugs [1]. This tool is one the most relevant and exciting automated bug finding tools of today. It grew out of academic work on Separation Logic [2, 3], which aimed to scale algorithms for reasoning about memory safety of programs that manipulate pointers. It arrived at Facebook with the acquisition of the program proof startup Monoidics in 2013, and its deployment has resulted in tens of thousands of bugs being fixed by Facebook's developers before they reach production. It is open source [1] and is currently used in several major companies including Amazon, Spotify, and Mozilla.

Infer returns the generated error reports to the developers without a specific order and also without assigning to each report a level of priority. This can turn out to be problematic since developers are expected to judge the importance of each error report on their own, having only their experience and some context of the system to make a decision. Many times this is insufficient, leading to a waste of time looking at issues that will not be materialized in true errors or are not as important as others.

One of the most critical challenges faced regarding this matter involves the accuracy of the reported warnings. Since static program analysis is performed without executing the software under analysis, static analysis tools must speculate on what the actual program behaviour will be, often resulting in a over-estimation of possible program behaviours. This leads to spurious warnings called False Positives (FP), that do not correspond to true defects. This also happens because these kind of tools rely on approximations and assumptions that help their analyses scale to large and complex software systems. The trade-off is that analysis results become imprecise, leading to false positives. For example, *Kremenek et al.* reported that at least 30% of the warnings reported by sophisticated static analysis tools are false positives [4]. This is an issue to developers since they will waste their time analysing and evaluating false positives while trying to find the real errors that affect the system. In some cases developers stop inspecting the tool's output since it is not reliable.

1.1 Work Objectives

The main objectives of this work are to improve the reliability of Infer and increase its performance by dealing with false positive warnings presents in the output.

To address these objectives, we develop a Neural Language Model, based on LSTM networks, capable of successfully modeling Infer's symbolic execution thus allowing the discovery of false positive patterns that lead to spurious warnings. This solution involves the analysis of the symbolic execution

performed by Infer while analyzing a program. The reasoning behind this idea is to look out for patterns in the code that would correspond to a false positive warning. We decided to specifically use Long Short Term Memory (LSTM) networks to perform neural language modeling. This is because LSTM networks are better suited to model longer sequences than other similar machine learning techniques, e.g., Recurrent Neural Networks (RNNs), being able to capture much longer dependencies which are common in source code. In this work we study the use of this kind of networks applied to our context and find out the optimal network parameters for our specific problem. We create a benchmark of more than 500 Infer reports manually labeled as false/true positives in order to train and test our model. We discuss a set of transformations to be applied to Infer's intermediate language with its respective symbolic execution, and evaluate the impact each one of them has in the performance of the machine learning model. As well as that, we study two different application scenarios where this model could be applied.

Our experimental results provide understanding into the applicability and performance of the developed models as well as the impact of each data preparations in two distinct application scenarios. First, we concluded that the data preparation for the LSTM networks has a significant impact on the performance of the model and that more detailed data preparations leads to better performance. Second, with the two application scenarios studied, we demonstrated that the abstraction aspect is key when dealing with cross-project warning classification, so that the generalizability of the model allows the correct classification of samples original from programs never seen before. Lastly, our model is able to correctly classify Infer output warnings, especially when dealing with within-project warning classification, which is the most common use case for Infer. This translates to a significant improvement in Infer's usability and effectiveness.

In summary, in this work we explore how neural language models can be used to prioritize bug reports produced by Facebook's Infer. In particular, we propose to address the following research questions:

- **RQ1:** What is the overall performance of the model in classifying Infer Static Analysis Tool warnings?
  In this first research question, we are interested in measuring the accuracy of the model when classifying Infer Static Analysis Tool and make a comparison with existent techniques.
- **RQ2:** What is the effect of different data preparations on performance?
  In this research question, our objective is to assess the impact that each data preparation had in the performance of the model.
- **RQ3:** What is the variability in the results?
  In this research question, we analyse the variance in the Accuracy, Recall and Precision measures of the different model variations.
- **RQ4:** How do different application scenarios impact the performance of the model?
  Finally, in this last research question we compare the overall performance of the model in the two

studied case scenarios: one where static analysis tools analyse the same program over time and the other where a static analysis tool analyses a new subject program.

1.2 Organization of the Document

In this document we specify all the details constituent of our solution as well as the research work that has been done to support it. Some important topics that are discussed include:

- The importance of false positive filtering;
- An overview on the current source code analysis techniques;
- How and why Machine Learning techniques can be effective in this domain;
- Our solution to the proposed problem;
- The experimental results achieved;

This thesis is organized as follows. In Chapter 2, we present the most relevant research done in the scope of this project. In Chapter 3, we explain our solution and the reasoning behind it. In Chapter 4, we detail the experimental results obtained and their respective analysis. Finally, in Chapter 5, we draw conclusions from the work that has been developed and point out some areas of improvement.

2

Background and Related Work

Contents

This chapter presents the concepts and techniques necessary to understand the foundations of this work and its motivations. The main topics include Static Analysis, False Positives, Language Modeling and Machine Learning. In addition, the tool under study, Infer, is thoroughly analysed and finally, similar works and studies are examined and discussed.

2.1 Static Analysis

Static Analysis (SA) is the process of analyzing a program's source code to find flaws without executing it. Usually it is performed by automated tools that assist programmers and developers in carrying out static analysis. These tools are programs that examine source code, executables, or even documentation, to find problems before they happen (i.e, without actually running the code [5]). The process provides an understanding of the code structure and helps ensuring that the code complies to modern-day industry standards. The software will analyse all code in a project checking for vulnerabilities while validating the code. These tools can be applied in different contexts and can detect a variety of issues including potential security violations [6] (e.g., SQL injection) , runtime errors (e.g., dereferencing a null pointer) and logical inconsistencies (e.g., a conditional test that can't possibly be true).

SA is an active research topic and for this reason there are many tools being used and built that implement this technique, e.g., FindBugs [7], Checkstyle [8] and PMD [9]. However, in this chapter we will discuss Infer (2.3) in particular, since this is the tool that we aim to extend.

2.2 False Positives

> *"Static analyses should strike a balance between missed bugs (false negatives) and unactioned reports (false positives)."* [10]

A true positive is a report of a potential bug that can happen in a run of the program in question (whether or not it will happen in practice) and a false positive is one that is impossible to happen with the current state of the program. A common knowledge in static analysis is that it is important to keep control of the false positives because they can negatively impact engineers who use the tools, as they tend to lead to apathy towards reported alarms. This has been emphasized in previous Communications of the Association for Computing Machinery (ACM) articles on industrial static analysis [11] [12]. False negatives, on the other hand, are potentially harmful bugs that may remain undetected for a long time. An undetected bug affecting security or privacy can lead to undetected exploits. In practice, fewer false positives often implies more false negatives, and vice versa, fewer false negatives implies more false positives. For instance, one way to deal with false positives is to fail to report when you are less than sure that a bug will be real. Unfortunately by silencing an analysis in this way (say, by ignoring paths or

by heuristic filtering) has the effect of missing bugs. And, if you want to discover and report more bugs you might also add more spurious behaviours i.e false positives. The false positive rate is challenging to measure for a large and rapidly changing codebase since it would be extremely time consuming for humans to judge all reports as false or true as the code is changing [10].

With this in mind, here are some reasons why ranking of error reports based on their probability of being false positives should be done:

1. When a tool is first applied to code, the initial few error reports should be those most likely to be true positives so that the developers can easily see if the rest of the errors are worth inspecting. Developers tend to immediately discard a tool if the first two or three error reports are false positives, giving these first few slots an enormous importance [4].
2. Even if the initial reports are true positives, in many cases a run of more than 10 false positives will cause a developer to stop inspecting the tool's output. This means that it is crucial to rank as many true positives at the top.
3. It is important to put the true positives at the top rank since developers will look at these first as being more urgent, thus solving the real problems of the program. With this, they can afford the time to check the following errors to see if they are indeed false positives and if not, to solve them.

2.3 Infer Static Analysis Tool

Infer is a static analysis tool developed by Facebook that can be applied to Java, Objective C, and C++ source code [1]. It started as a specialized analysis based on Separation Logic [2] [3] that targeted memory issues, but has evolved into an analysis framework supporting a variety of sub-analyses. It reports errors related to memory safety [13], to concurrency [14], to security (information flow), and many more specialized errors.

Infer's main deployment model is based on fast incremental analysis of code changes [10]. Facebook practices continuous software development where a codebase is altered by thousands of programmers submitting code modifications, i.e. *'diffs'*. A programmer prepares a *diff* (a code change) and submits it to code review. Figure 2.1 shows a simplified picture of this process: *Diff time* corresponds to the time when a developer submits a *diff* to the system for review. Post land, or commit, is the period after a *diff* has been incorporated into the master build of the system. Continuous Integration (CI) refers to the continuous integration of *diffs* into the code base subject to code review. The developers share access to a single codebase and they commit a *diff* to the codebase after passing code review. Infer is run at *diff* time, before the commit occurs, while other longer-running tests are run post commit. When a *diff* is submitted an instance of Infer is run in Facebook's internal CI system (Sandcastle). Infer does not have the need of processing the entire code base in order to analyse a *diff*, and for this reason it is fast.

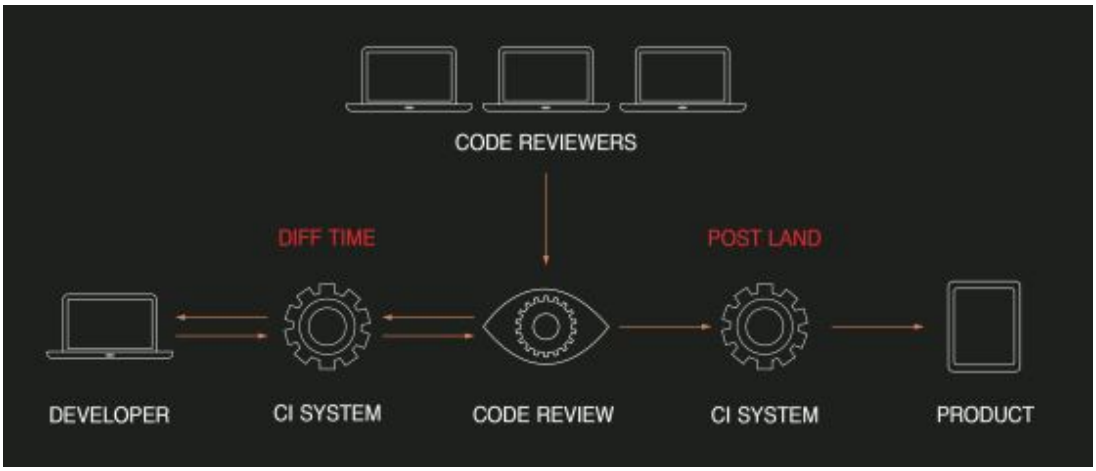

Figure 2.1: Continuous development and deployment.

Infer will take 10min-15min to run on a *diff* on average, this includes the time to check out the source repository, build the *diff*, and to run on base and parent commits. Infer then writes the output to the code review system. In the default mode used most often it reports only regressions: new issues introduced by a *diff*. In contrast, when Infer is run in whole-program mode it can take more than an hour (depending on the app being analysed).

2.3.1 Workflow

There are always two phases of an Infer run, regardless of the input language (Java, Objective-C, or C):

1. In the **Capture phase** the compilation commands are captured by Infer to translate the files that are to be analysed into Infer's own internal intermediate language. This translation is similar to compilation, Infer takes information from the compilation process to perform its own translation. This is why Infer is called with a compilation command:

   ```
   infer run -- javac File.java
   ```

   What happens is that the files get compiled as usual, and they also get translated by Infer to be analysed in the second phase. If no file gets compiled, also no file will be analysed. Infer stores the intermediate files in the results directory which by default is created in the folder where the Infer command is invoked, called `infer-out/`.
2. In the **Analysis phase** the files present in `infer-out/` are analysed by Infer. Infer analyses each function and method separately. If Infer encounters an error when analyzing a method or function, it stops there for that method or function, but will continue the analysis of other methods and functions. So, a possible workflow would be to run Infer on the code, fix the errors generated, and run it again to find possibly more errors or to check that all the errors have been fixed. The errors will then be displayed in the standard output (Figure 3.5) and also in a file `infer-out/bugs.txt`.

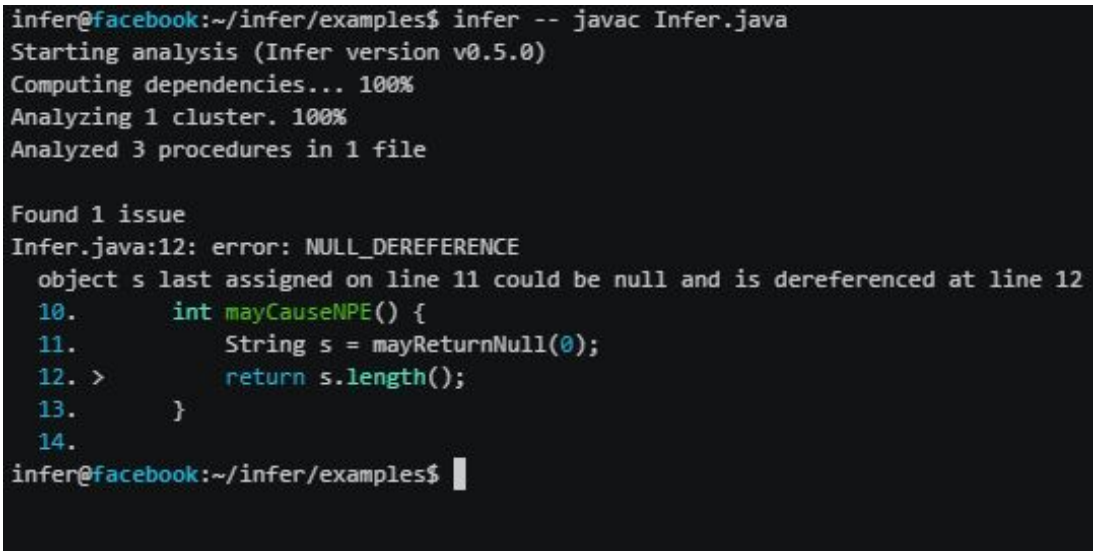

Figure 2.2: Infer warning example

2.3.2 Warning Prioritization

Facebook uses static analysis in the form of Infer to prevent bugs that would affect their products, and so they rely on the engineers judgement as well as the data from production to provide them with the information on which bugs matter the most.

It is crucial for a developer to be able understand that not all bugs are the same: different bugs can have different levels of importance or severity depending on the context and the nature. For instance, a memory leak on a rarely used service might not be as important as a vulnerability that would allow attackers to gain access to unauthorized information. Additionally, the frequency of a bug type can affect the decision of how important it is to attend to it. If a certain kind of crash, such as a null pointer error in Java, were happening hourly, then it might be more important to target than a bug of similar severity that occurs only once a year [10].

The problem with this type of evaluation is the fact that it can be subjective to the experience of each developer, and furthermore conditioned by the specific pieces of information that were available. For these reasons it is of the uttermost importance that the feedback that Infer gives to the engineers containing the errors found in the program it is already, in some way, prioritized.

2.3.3 False Positives

> *"We want to be clear that if you run Infer on your project you might get very good results, but it is also possible that you don't."* [1]

[1] https://fbinfer.com/docs/limitations.html

Infer developers do not make strong claims about rates of false alarms or similar when applied to arbitrary codebases. For example, the developers state that they had some success getting bugs fixed in the DuckDuckGo Android App using Infer, but that they found many false positives when running Infer on GNU coreutils.

We can then conclude that Infer does not guarantee that a large number of false positives warnings will not be present in the output. This situation presents a real problem that requires a solution. To further exemplify this problem we decided to run Infer on real life applications. We used Defects4J [15], a collection of reproducible bugs, which, among other things, contains a list of open-source projects written in Java and the number of bugs that were found on each one. We decided to run Infer on two of those projects: **Closure compiler** and **Apache commons-lang.**

Infer was executed with the following checkers:

- Biabduction (C/C++/ObjC, Java)
- Fragment retains view (Java)
- Inefficient keyset iterator (Java)
- RacerD (C/C++/ObjC, Java)
- Starvation analysis (C/C++/ObjC, Java)

These checkers are by default included in an Infer run and they can detect a number of bug types, including: *Deadlock*, *Dead store*, *Memory leak*, *Null dereference*, and others[2].

2.3.3.A Apache commons-lang project

Infer was executed on a version of this system that contained three reported bugs that correspond to four application tests failed. Infer found 7 issues:

- **Null Dereference:** 5
- **Resource Leak:** 1
- **Thread Safety Violation:** 1

None of the issues found by Infer corresponded to bugs in the application that were previously accounted for. We manually analysed each issue raised by Infer in order to check if they were indeed true positives and concluded than more than a half of the issues reported were False Positives: 3 True Positives and 4 False Positives. An example of a False Positive warning regards the second issue reported by Infer, that claims the existence of a *Resource Leak* in the file `SerializationUtils.java` at line 106. Infer reported that the resource acquired in line 88 is not released, however, by looking at the code in Figure 2.3 we can conclude that the resource will always be released in the *Finally* block at line 103. The remaining 3 False Positive warnings were all of the *Null Dereference* type.

[2] https://fbinfer.com/docs/checkers-bug-types.html

```java
public static <T extends Serializable> T clone(final T object) {
    if (object == null) {
        return null;
    }
    final byte[] objectData = serialize(object);
    final ByteArrayInputStream bais = new ByteArrayInputStream(objectData);

    ClassLoaderAwareObjectInputStream in = null;
    try {
        // stream closed in the finally
        in = new ClassLoaderAwareObjectInputStream(bais, object.getClass().getClassLoader());
        /*
         * when we serialize and deserialize an object,
         * it is reasonable to assume the deserialized object
         * is of the same type as the original serialized object
         */
        @SuppressWarnings("unchecked") // see above
        final
        T readObject = (T) in.readObject();
        return readObject;

    } catch (final ClassNotFoundException ex) {
        throw new SerializationException("ClassNotFoundException while reading cloned object data", ex);
    } catch (final IOException ex) {
        throw new SerializationException("IOException while reading cloned object data", ex);
    } finally {
        try {
            if (in != null) {
                in.close();
            }
        } catch (final IOException ex) {
            throw new SerializationException("IOException on closing cloned object data InputStream.", ex);
        }
    }
}

```

Figure 2.3: Code analysed by Infer that lead to a RESOURCE LEAK issue reported

2.3.3.B Closure Compiler

Infer was executed on a version of this system that contained 1 reported bug that correspond to 8 application tests failed. Infer found a total of 338 issues:

- **Null Dereference:** 160
- **Resource Leak:** 23
- **Thread Safety Violation:** 91
- **Deadlock:** 59
- **Inefficient Keyset Iterator:** 5

None of the issues found by Infer corresponded to bugs in the application that were previously accounted for. Since this is a larger application a lot more issues were found. This test demonstrates the large amount of data that is output by Infer which developers need to analyse and the need of employing bug prioritization techniques in this context.

Figure 2.4: Issues found by Infer when running on top of Apache Commons-lang

2.4 Language Modeling

Language Modeling (LM) is a function that captures the statistical characteristics of the distribution of sequences of words in a natural language, typically allowing one to make probabilistic predictions of the next word given preceding ones. In the Natural Language Processing (NLP) context these models have been used in several different tasks including machine translation, handwriting recognition, speech recognition, and information retrieval.

2.4.1 N-gram Language Models

The dominant methodology for probabilistic language modeling since the 1980's has been based on n-gram models [16, 17]. The probability of a token is computed based on the n previous tokens in the sequence. N-gram language models have shown extensive success in NLP applications. However, such models have two issues. First, they operate on small ranges (the n previous tokens), with usually low values of n. Increasing n does not scale well if the vocabulary is large: for a vocabulary of size m, there are m^n possible n-grams. Second, they suffer from a data sparsity problem: not all possible n-grams are present in the corpus.

2.4.2 Neural Language Models

The state-of-the-art in NLP is made of Neural Language Modeling (NLM) [18]. A NLM is a language model based on Neural Networks, and exploits their ability of learning distributed representations to reduce the impact of the curse of dimensionality. In the context of learning algorithms, the curse of dimensionality refers to the need for huge numbers of training examples when learning highly complex functions. When the number of input variables increases, the number of required examples can grow exponentially. The curse of dimensionality arises when a huge number of different combinations of values of the input variables must be differentiated from each other, and the learning algorithm needs at least one example per relevant combination of values. In the context of language models, the problem arises from the huge number of possible sequences of words.

A distributed representation of a word is a vector of features which characterize the meaning of the word and are not mutually exclusive. The basic idea is to learn to associate each word in the dictionary with a continuous-valued vector representation [19]. Each word corresponds to a point in a feature space, e.g., each dimension of that space corresponds to a semantic or grammatical characteristic of words. The intention is that functionally similar words get to be closer to each other in that space. A sequence of words can thus be transformed into a sequence of these learned feature vectors. The neural network learns to map that sequence of feature vectors to a prediction such as the probability distribution over the next word in the sequence.

The advantage of this distributed representation approach is that it allows the model to generalize well to sequences that are not in the set of training word sequences, but that are similar in terms of their distributed representation. Because neural networks tend to map nearby inputs to nearby outputs, the predictions corresponding to word sequences with similar features are mapped to similar predictions. Because many different combinations of feature values are possible, a very large set of possible meanings can be represented compactly, allowing a model with a comparatively small number of parameters to fit a large training set.

In addition, some neural architectures such as RNNs [20], LSTM networks [21, 22] or Transformers [23] are able to model relatively long range dependencies [24]. LSTM based language models can also be easily adapted to classification tasks by just replacing the last layers, that perform word predictions, with ones performing classification [25]. They are also highly effective in a variety of additional NLP tasks, including sequence to sequence models for translation or summarization tasks [26].

2.4.3 Neural Language Models in Software Engineering

Neural language models have also been used to analyse and model source code. In this section we review and discuss several different works in this area.

Babii et. al [27] started by making a list of important modeling choices for source code vocabulary and explores their impact on the resulting vocabulary on a large-scale corpus of almost 15.000 projects. Most importantly they show that a subset of modeling decisions have a major impact when modeling a language, thus allowing to train both accurately and quickly Neural Language Models on a large project corpus. Another study [28] contrasts programming languages against natural languages and discuss how these similarities and differences drive the design of probabilistic models. *Allamanis et. al* [29] introduced the use of graphs to both represent the syntactic and semantic structure of source code. They use graph-based deep learning methods for learning how to reason over program structures by constructing graphs from source code and scaling Gated Graph Neural Networks training to such large graphs. *Zhang et. al* [30] proposes a Neural Network for source code representation using Abstract Syntax Trees (AST). They split each large AST into a sequence of small statement trees and encode the statement trees to vectors by capturing the lexical and syntactical knowledge of statements. Based on the sequence of statement vectors, a bidirectional Recurrent Neural Network (RNN) model is used to leverage the naturalness of statements and finally produce the vector representation of a code fragment. *Alon et al.* presents CODE2SEQ [31]: an approach that leverages the syntactic structure of programming languages to better encode source code. The model they propose is able to represent a code snippet as the set of compositional paths in its AST and uses attention to select the relevant paths while decoding. Later in 2019 a work by the same authors made a presentation of a neural model for representing snippets of code as continuous distributed vectors (”code embeddings”). The main idea is to represent

a code snippet as a single fixed-length code vector, which can be used to predict semantic properties of the snippet. To this end, code is first decomposed to a collection of paths in its AST. Then, the network learns the atomic representation of each path while simultaneously learning how to aggregate a set of them [32].

2.5 Long Short-term Memory Networks

For text classification RNNs [33, 34] have emerged as a strong alternative approach that views text as a sequence of words, with arbitrary-length, and automatically learns vector representations for each word in the sequence [35]. Even though RNNs can be successfully used to model sequences, they are not quite efficient when applied to relatively long sequences. This can be explained by the vanishing gradient problem [36] that prevents standard RNNs from learning long-term dependencies.

LSTM networks were first proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber [21], and are among the most widely used models in Deep Learning for NLP today. The main advantage of LSTM networks relies on the fact that they do not suffer from problems such as gradient vanishing or exploding, thus making these type of networks capable of modeling long sequences as well as making the training phase much easier.

2.5.1 Network

An LSTM network memory unit consists of four attributes, namely a forget gate, a cell state, an input gate, and an output gate. The cell state remembers the information of the entire sequence and the three gates control the input and output of the cell, as shown in Figure 2.5.

At the start of the process, the forget gate checks which information to throw away and which information to keep in the cell state. Equation 2.1 is used for the forget gate. It is calculated at cell state c_{t-1} using hidden state h_{t-1} and input x_t. The output of the forget gate, between 0 and 1, is produced by the *sigmoid* function. An output value of 1 means keep all information in the cell state.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.1}$$

To store a new piece of information in the cell state, the input gate decides which value will be updated using the *sigmoid* function. The *tanh* function creates a new candidate value $\widetilde{c_t}$ for the cell state.

$$i_t = \sigma(w_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.2}$$

$$\widetilde{c_t} = \tanh(w_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.3}$$

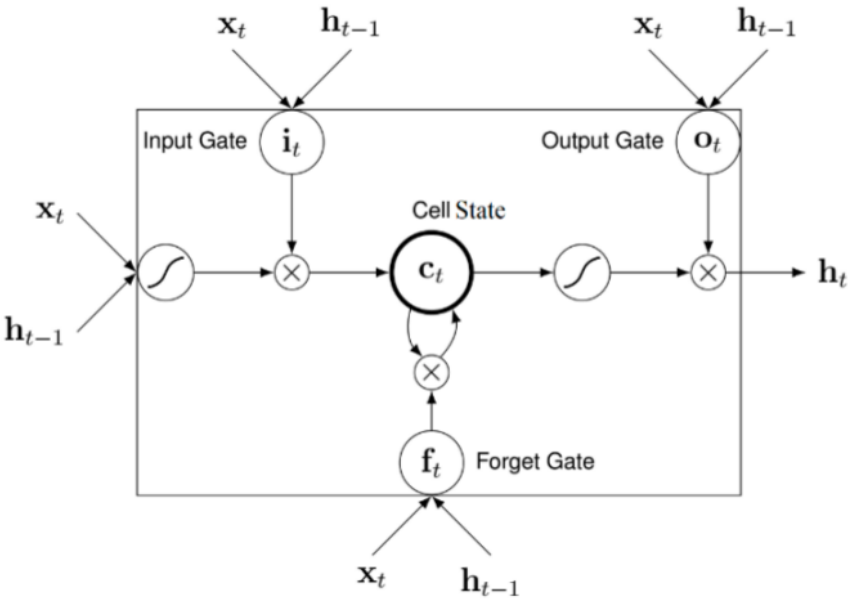

Figure 2.5: Internal structure of a simple long short-term memory (LSTM) unit. (Source: [37])

Then, the old cell state c_{t-1} is used to update c_t.

$$c_t = f_t * c_{t-1} + i_t * \widetilde{c_t} \tag{2.4}$$

We can now calculate the output of LSTM, which is based on a filtered version of the cell state.The sigmoid function decides which part of the cell state is going to the output and then updates the weight accordingly.

$$o_t = \sigma(w_o.[h_{t-1}, x_t] + b_o) \tag{2.5}$$

$$h_t = o_t * \tanh(c_t) \tag{2.6}$$

The gating mechanism is what allows LSTMs to explicitly model long-term dependencies. By learning the parameters for its gates, the network learns how its memory should behave. There are many more variations of this model. *LSTM: A Search Space Odyssey* empirically evaluates different LSTM architectures [38].

2.5.2 Language Modeling

Neural networks have become increasingly popular in the development of language models, to the point that it may now be the preferred approach. In source code, a single line may have reliance on the

preceding lines and this can lead up to the need of remembering very long sequences, making it difficult to evaluate complex source code by any conventional language model.

This can be solved by using LSTM Neural Networks for Language Modeling [22]. The fact that LSTMs are able to model very long sequences [39] is key for being able to successfully model source code and its dependencies. Because of this many researchers have used this type of models to detect source code errors in software engineering and programming education. In this section we review some of these works.

Pu et al. proposed a source code correction method based on LSTM using code segment similarities. The study leveraged the sequence-to-sequence (seq2seq) neural network model with natural language processing tasks for the code correction process [40]. Another study [41] proposed an LSTM-based model for programming education where the model predicts the next word by analyzing incomplete source code. To help novice programmers the model predicts the next word to complete a program. The model achieved a high degree of prediction accuracy. *Reyes et al.* [42] classified archived source code by type of programming language using an LSTM network. Empirical results showed that the proposed network outperformed both the Naive Bayes classifier and linguist classifier. In another study [43], a source code bug detection technique that uses LSTM was proposed. The hyper parameters of the network were adjusted to determine the optimal perplexity and training time. The LSTM network produces a plausible outcome for source code bug detection. *Rahman et al.* [37] presented a sequential language model that uses an attention-mechanism-based LSTM to assess and classify source code based on the estimated error probability. The proposed model was trained using correct source code and then evaluated its performance. The experimental results show that the proposed model has logic and syntax error detection accuracies of 92.2% and 94.8%, respectively, outperforming state-of-the-art models. Their proposed model can be effective various settings including programming education and software engineering by improving code writing, debugging and error-correction. *Yu et al.* [44] proposed a classification model for classifying a bug report as either helpful or unhelpful using an LSTM. By filtering unhelpful reports before running an Information Retrieval (IR) based bug locating system, their approach helps reduce false positives and improve the ranking performance. The model was tested on over 9,000 bug reports from three software projects. The evaluation results show that the model improves a state-of-the-art IR based system's ranking performance under a trade-off between the precision and the recall. They also concluded that their LSTM-network achieves the best trade-off between precision and recall than other classification models including convolutional neural networks and multilayer perceptrons. *Dam et al.* proposes a novel approach to build a language model for software code for being able to capture a long context where dependent code elements scatter far apart. Their language model is built upon a deep learning based Long Short Term Memory architecture that is capable of learning long-term dependencies which happen frequently in software code. Results from

the evaluation of the model on a corpus of Java projects have demonstrated the effectiveness of this language model [45].

LSTM networks have also been used in sentiment analysis and classification in several works, being able to successfully model the intricate aspects of the natural language. In this work [46] researchers propose an Attention-based LSTM for aspect-level sentiment classification. The attention mechanism can concentrate on different parts of a sentence when different aspects are taken as input. Their results show that the model achieves state-of-the-art performance on aspect-level sentiment classification. *Chen et al.* proposes a hierarchical neural network to incorporate global user and product information into sentiment classification. They first build a hierarchical LSTM model to generate sentence and document representations. Afterwards, user and product information is considered via attentions over different semantic levels due to its ability of capturing crucial semantic components. The experimental results show that this model achieves significant and consistent improvements compared to all state-of-the-art methods [47].

2.6 False Positive filtering using Machine Learning

There are several existing techniques that aim for a prioritization of bugs having in mind the false positives problem in the context of automated bug reporting tools. Research efforts have successfully applied machine learning techniques to filter out the false positive error reports from other SA tools. In this section we present some of these works.

For example, *Kremenek and Engler* [4] proposed z-ranking, a technique to rank error reports emitted by static program checking analysis tools. It employs a simple statistical model to rank those error messages most likely to be true errors over those that are least likely. It was also demonstrated that z-ranking applies to a range of program checking problems and that it performs up to an order of magnitude better than randomized ranking. *Yüksel and Sözer* [48] evaluated the application of machine learning techniques to classify alerts based on a set of artifact characteristics. The study was made in the context of an industrial case study to classify the alerts generated for a digital TV software. It was created a benchmark based on this code base by manually analyzing thousands of alerts. Then, evaluated 34 machine learning algorithms using 10 different artifact characteristics and identified characteristics that have a significant impact. *Tripp et al.* [49] efforts focused on the false positive problems within static security checkers and address this problem by introducing a general technique to refine their output. The main idea was to apply statistical learning to the warnings output by the analysis based on user feedback on a small set of warnings. This leads to an interactive solution, whereby the user classifies a small fragment of the issues reported by the analysis, and the learning algorithm then classifies the remaining warnings automatically. An important aspect of this solution is that the user can

express different classification policies, ranging from strong bias toward elimination of false warnings to strong bias toward preservation of true warnings, which the filtering system then executes. *Ruthruff, Joseph R., et al.* [50] reports automated support using logistic regression models that predicts the foregoing types of warnings from signals in the warnings and implicated code. Because examining many potential signaling factors in large software development settings can be expensive, they use a screening methodology to quickly discard factors with low predictive power and cost-effectively build predictive models. The empirical evaluation indicates that these models can achieve high accuracy in predicting accurate and actionable static analysis warnings, and suggests that the models are competitive with alternative models built without screening. It was proposed by *Koc et al.* [51] , a process whose goal is to discover program structures that cause a given static code analysis tool to emit false error reports, and then to use this information to predict whether a new error report is likely to be a false positive as well. To do this, the code is first pre-processed to isolate the locations that are related to the error report. Then, machine learning techniques are applied to the pre-processed code to discover correlations and to learn a classifier. Later in 2019 a study by the same authors [52] was conducted with the objective of empirically assessing machine learning approaches for triaging reports of a java static analysis tool. They describe a systematic, comparative study of multiple machine learning approaches for classifying static analysis results. Their experimental results provide significant insights into the performance and applicability of the ML algorithms and data preparation techniques. It was observed in this study that the recurrent neural networks perform better compared to the other approaches. And that with more precise data preparation, large performance improvements over the state of the art could be achieved.

3

Infer Warning Prioritization

Contents

The following section describes in detail the implemented solution. The solution consists in prioritizing Infer Static Analysis tool warnings by identifying False Positive reports. To achieve this objective we take advantage of Machine Learning techniques, more specifically LSTM networks.

We describe our solution using a top-down approach. We start in Section 3.1 by making an overall description of our approach and explain the configuration of each part of the solution. In Section 3.2 we analyse Infer in further detail and we build a benchmark consistent of real world programs that will allow us to train and evaluate the model. Then, we talk about the data processing phase in Section 3.3, presenting a set of transformations used to extract meaningful information from Infer's symbolic execution by normalizing and transforming the language into a sequence of tokens. As well as that we also explain the translation of the sequence of tokens into a representation that can be processed by the machine learning model. Finally, in Section 3.4, we talk about the machine learning model we chose to tackle this problem and the reasons behind it and as well as explain in detail the choice and configuration of each layer present in the model.

3.1 Solution Description

To prioritize Infer warnings based on false positive identification, we will need to correctly analyse and assess the symbolic execution where Infer bases its analysis. For this task, we will use LSTM networks applied to language modeling to extract and learn the features necessary to successfully identify false positive reports. This solution will be able to analyse each of the warnings present in Infer's output individually and assign to each one their probability of being a false positive.

The main advantage of this approach is that saves Infer users valuable time and effort when analysing the output reports since they are already filtered and ranked accordingly. Besides that, it adds value to the tool, since the output is more reliable and consistent. Also, since the analysis made by Infer is based on their own symbolic execution, this model could possibly be applied to different kinds of analysis as well as different programming languages.

The workflow of the proposed model is separated in three different phases as described in Figure 3.1: **(a)** data processing **(b)** supervised training of the LSTM network **(c)** false positive probability prediction and warning classification.

In the first phase, the output of the symbolic execution which Infer generates when analysing source code is processed. The processing is done by applying a set of transformations in order to normalize the code. These transformations are explained in detail in Section 3.3. The preprocessed data is also tokenized and encoded into integer vectors, allowing this information to be fed into the network.

In the second phase, an LSTM network is used to model data that is already normalized and encoded. The implementation details regarding the model construction and parameters are explained in

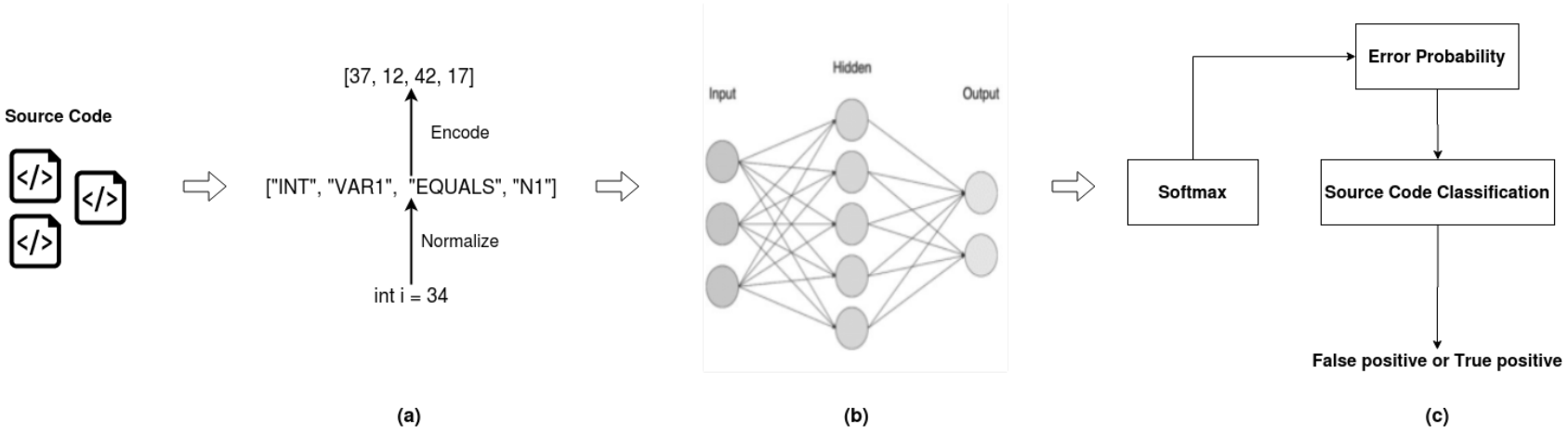

Figure 3.1: Workflow of the proposed model

Section 3.4.

Finally, in the third and final phase, the already trained LSTM network is used to classify each warning present in the output of Infer, these warnings are then sorted accordingly and reported to the user. The classification process is carried out by the softmax layer of the model.

3.2 Infer and Benchmarks

In this work, we study Infer version 0.17.0 and focus on Java written applications. Furthermore, we only evaluate reports that do not correspond to test files. When an Infer run occurs several checkers which identify different kinds of errors are active. These are the default checkers for the Java programming language:

(1) Biabduction (C/C++/ObjC, Java)
(2) Fragment retains view (Java)
(3) Inefficient keyset iterator (Java)
(4) Starvation analysis (C/C++/ObjC, Java)
(5) RacerD (C/C++/ObjC, Java)

For this work we disabled **RacerD (C/C++/ObjC, Java)** checker. The reason behind this decision is mentioned by *Blackshearer et al.* [14]. It states that RacerD performs a different kind of analysis than typical checkers, where the importance relies in keeping the false positive rates low even though it can cause the existence of false negatives. This is why this checker has a very low false positive rate compared to other types of analysis. We also disabled **Fragment retains view (Java)** checker since it is only focused on Android specific errors.

With the checkers that remain active, Infer is expected to detect the following type of errors [1]:

- Checkers Immutable Cast
- Deadlock
- Field should be nullable
- Null dereference
- Resource leak
- Retain cycle
- Inefficient keyset iterator
- UnsafeGuardedByAccess

[1] https://fbinfer.com/docs/checkers-bug-types

3.2.1 Infer intermediate language

In this work we decided not to analyse the source code present in the input program's, or bytecode as similar works do in order to track false positive warnings. This is motivated by the fact that Infer bases its analysis in the self-generated intermediate language. It was mentioned before while explaining Infer's workflow in Section 2.3.1, that in the capture phase Infer translates the files that are to be analysed into Infer's own internal intermediate language. This translation is similar to compilation, Infer takes information from the compilation process to perform its own translation. Infer, therefore, does not directly analyse input source code or even bytecode, it analyses the files translated to its own intermediate language. Therefore in this work we analyse and model Infer's intermediate language and subsequent symbolic execution. An example of this language and its respective symbolic execution, can be seen in Figure 3.2.

```
Instruction n$33=*n$32.edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location.lma_addr:edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location [line 236]
.... Rearrangement Start ....
Exp: null.edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location.lma_addr
Prop:
n$29 = old_this; n$31 = 0; n$32 = null ;
(0  <  @f$5); @f$13 != 0; @f$12 != @f$0; @f$11 != @f$0; @f$0 != 0; @f$0 != ""; UND < boolean Iterator.hasNext() > :231(@f$13); UND < Iterator List.iterator() > :231($bcvar2);
str = 0nullify:java.lang.String*; cntr = 0update:234:int; $irvar3 = n$1113initial:void; $bcvar2 = @f$1update:231:void; #GB<>$edu.ucla.cs.compilers.avrora.cck.text.CharUtil =
@f$10|->{java.util.HashMap.lastKey1:@f$11formal(z), java.util.HashMap.lastKey2:@f$12formal}formal(z):java.util.HashMap( sub ); @f$9|->{edu.ucla.cs.compilers.avrora.avrora.mon
[footprint
   (0  <  @f$5); @f$13 != 0; @f$12 != @f$0; @f$11 != @f$0; @f$0 != 0; @f$0 != "" *
   $RET_java.util.Iterator.next():java.lang.Object|abducedRetvar = @f$0none:java.lang.Object*( sub ); #GB<>$edu.ucla.cs.compilers.avrora.cck.text.CharUtil = {edu.ucla.cs.comp
   @f$10|->{java.util.HashMap.lastKey1:@f$11rearrange:(z)81, java.util.HashMap.lastKey2:@f$12rearrange:81}rearrange:(z)269:java.util.HashMap( sub ); @f$9|->{edu.ucla.cs.compi

explain_dereference Binop.Leteref n$32.edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location.lma_addr
Finding deref'd exp
find_outermost_dereference: Lfield n$32.edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location.lma_addr
find_outermost_dereference: normal var n$32
find_normal_variable_load defining &loc
exp_lv_dexp: program var &loc
lookup: found Dpvar
Failure of symbolic execution: NULL_DEREFERENCE [B1] object `loc` last assigned on line 233 could be null and is dereferenced at line 236 biabduction/Rearrange.ml:1623:55-62:
Can't find field edu.ucla.cs.compilers.avrora.cck.text.CharUtil.HEX_CHARS in StrexpMatch.find
Precondition:
(0  <  val$9); $RET_java.util.Iterator.next():java.lang.Object|abducedRetvar != val$5; $RET_java.util.Iterator.next():java.lang.Object|abducedRetvar != val$4; $RET_java.util.
$RET_java.util.Iterator.next():java.lang.Object|abducedRetvar = val$16none:java.lang.Object*( sub ); #GB<>$edu.ucla.cs.compilers.avrora.cck.text.CharUtil = {edu.ucla.cs.compi
val$14|->{edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMonitor.TO:val$10rearrange:(z)231}rearrange:(z)231:edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMonitor
SIL INSTR:
n$29=*&this:edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMonitor$PointToPointMon* [line 236];
_=*n$29:edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMonitor$PointToPointMon [line 236];
n$31=*&cntr:int [line 236];
n$32=*&loc:edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location* [line 236];
n$33=*n$32.edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location.lma_addr:edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$Location [line 236];
n$34=_fun_void TripTimeMonitor$PointToPointMon.addPair(int,int)(n$29:edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMonitor$PointToPointMon*,n$31:int,n$33:int) virtual
EXIT_SCOPE(_,n$29,n$31,n$32,n$33,n$34); [line 236];

.... After Symbolic Execution ....
```

Figure 3.2: Infer intermediate language

This decision can have a major impact in the accuracy score of the model, since the code analysed does not depend on different programmers and styles of coding. As all the information that is generated by Infer it has the same style independent of the program it is analyzing, only differing in some program specific words. This property makes it easier for the machine learning model to pick up false positive patterns and to more accurately model the language. This also has an impact in the way the information is preprocessed as reported in Section 3.3.

3.2.2 Benchmarks

The benchmark in this evaluation consists of 5 real-world programs that cover a wide range of issues. We analyse these programs using Infer and then manually classify the reports as true or false positives.

We selected these programs using the following criteria:

- The programs have to be written in Java.
- The programs must be open source since we need to access the source code to analyse each one with Infer.
- The programs are under active development and are highly used in order to increase relevancy.

Table 4.1. shows the programs we chose. All the entries of are from the Dacapo Benchmark [53, 54], except for JODA-TIME that was used by *Koc et al. 2019* [52] as part of their proposed real-world benchmark. The number of Lines of Code (LOC) was calculated by using Cloc (version 1.84) [55].

Table 3.1: Programs of the Benchmark

Program	**Description**	**LoC**
1. Apache Tomcat	Implements Java Servlet [56]	435438
2. Apache Xalan Java	Transforms XML docs into HTML [57]	205644
3. Avrora	Simulation and Analysis Tool [58]	92041
4. Joda-Time	Date and time framework [59]	94973
5. Jython	Python for the Java Platform [60]	945500

3.2.3 Infer report evaluation

Running Infer on top of each program of the benchmark resulted in more than 500 reports. We then labelled the reports by manually reviewing the code, resulting in 313 true positives and 230 false positives as can be seen in Table 3.2. To label a SA report, we first analyse the method containing the line where the warning originated from and the call tree originated from that reported error line, if existent. Then we inspect all the code present both in the call tree and in the method under analysis until either we find the error reported by the tool — indicating a true positive — or we exhaust the call tree without identifying any issue — indicating a false positive. False positives represented more than **40%** of all the reports output by Infer when analysing our benchmark.

By looking at each false positive report we concluded that 89% of them were NULL DEREFERENCES and the remaining 11% were RESOURCE LEAKS. We therefore decided to focus on these two types of bugs.

Table 3.2: Infer report manual classification

Program	Warnings	True Positives	False Positives
1. Apache Tomcat	265	153	112
2. Apache Xalan Java	50	27	23
3. Avrora	51	39	12
4. Joda-Time	12	11	1
5. Jython	165	83	82
TOTAL	543	313	230

3.3 Data Processing

Differently from many other SA tools that directly analyse the source code or bytecode that is generated in compilation time, Infer analysis falls on their own intermediate language.

In this section we present and discuss a set of modeling choices for source code vocabulary that we implemented in this work. All these transformations were studied in the past and applied in the context of false positive report classification [52]. In this work we decided to apply this set of transformations to our dataset with some changes. The first one is that we only abstract program-specific words after extracting english words from identifiers and not the other way around. We do this for the simple reason that when we extract english words from the identifiers we are reducing the vocabulary. If we abstracted program-specific words before applying this transformation to the dataset many identifiers would be eliminated and the information present on those identifiers would be lost. The second change we applied, was to split literals in order to reduce vocabulary. This transformation was studied in a recent work [27] and works particularly well in our case since the intermediate language generated by Infer contains literals on almost all of its identifiers. Lastly, we do not apply the same program slicing techniques used in the work mentioned above [52], which include computing backward slices and the program dependency graph. Instead we extracted the necessary information directly from the node responsible for the line where the error originated and the symbolic execution session when the error was reported.

The set of transformations used is described below. We list the transformations in order of complexity, and a transformation is applied only after applying all of the other less complex transformations.

1) Data Cleansing (T_{cln}): this transformation consists in performing basic data cleansing. First, whitespacing is fixed by placing a single space character between words and eliminating tabs, relevant special characters are replaced with tokens and irrelevant special characters are deleted. Second, tokens from paths of classes and methods are split by replacing the delimiters '.' or '/' for whitespaces.

2) Abstracting Numbers and Strings both in Literals and Identifiers (T_{ans}) : This transformation replaces numbers and string literals that are present in the code with abstract values.

This improves the learning phase of the model by reducing the size of the vocabulary of the dataset and helps us in the training of more generalizable models. First regarding literals, two digit numbers are replaced with N2, three digit numbers are with N3, and numbers with four or more digit are with N4+. We applied similar transformations for negative numbers as well. Next, we extract the list of string literals and replace each of them with the token STR followed by a unique number. For example, the first string literal in the list are replaced with STR1. To number literals that are also identifiers we decided to split them in their constituent digits in order to reduce the size of the vocabulary without losing information. To string literals that are also identifiers we attributed each one an unique token, e.g., VAR1.

3) EXTRACTING ENGLISH WORDS FROM IDENTIFIERS (T_{ext}) : Many identifiers are composed of multiple English words. For example, the GETFILEPATH method from the Java standard library consists of three English words: get, File, and Path. To make our models more generalizable and to reduce the vocabulary size, we split any camelCase or snake_case identifiers into their constituent words.

4) ABSTRACTING PROGRAM-SPECIFIC WORDS (T_{aps}) : In Infer intermediate language and respective symbolic execution one can find, as in source code, program specific words. Learning these program specific words and identifiers may not be useful for classifying SA reports in other different programs. Therefore, this transformation focus on abstracting certain words from the dataset that occur less than a certain amount of time, or that only occur in a single program. We do this by replacing the least common N words with the token UNK. Similar to the two previous transformations, it is expected to improve the effectiveness of the model by reducing the vocabulary size and generalizability via abstractions.

```
PROCESSING PROP 4 INSTRUCTION N 1 3 EQUALS PTR ADDR IR VAR 1 EDU UCLA CS COMPILERS
AVRORA AVRORA SIM MCU ADC PTR IN V PATH FIND EXP N 2 7 1 V PATH FIND DO SEXP NO MATCH
ON EEXP ADDR IR VAR 2 V PATH FIND CANNOT FIND N 2 7 1 REARRANGE MENT START EXP ADDR
IR VAR 1 PROP F 6 DIFFERENT ADC F 5 DIFFERENT ADC MEMNE SMALLER NEW BIGGER LBRACKET
V PATH RBRACKET LPARENTHESES IR VAR 2 RPARENTHESES RET SMALLER SENSOR BIGGER LPARENTHESES
N 2 7 3 RPARENTHESES M EQUALS NULLIFY EDU UCLA CS COMPILERS AVRORA AVRORA SIM MCU ATMEL
MICROCONTROLLER PTR SUB OLD M EQUALS F 3 FORMAL EDU UCLA CS COMPILERS AVRORA AVRORA SIM
MCU ATMEL MICROCONTROLLER PTR SUB IR VAR 0 EQUALS NULLIFY VOID ADC CHANNEL EQUALS F 2
FORMAL INT SUB OLD ADC CHANNEL EQUALS F 2 FORMAL INT SUB RETURN EQUALS N 2 7 2 INITIAL
VOID IR VAR 1 EQUALS NULL UPDATE VOID IR VAR 2 EQUALS N 2 7 1 UPDATE VOID THIS EQUALS F
1 FORMAL EDU UCLA CS COMPILERS AVRORA AVRORA SIM PLATFORM SENSORS ACCEL SENSOR PTR SUB
OLD THIS EQUALS F 1 FORMAL EDU UCLA CS COMPILERS AVRORA AVRORA SIM PLATFORM SENSORS ACCEL
SENSOR PTR SUB ASP EQUALS F 0 FORMAL EDU UCLA CS COMPILERS AVRORA AVRORA SIM PLATFORM SENSORS
ACCEL SENSOR POWER PTR SUB OLD ASP EQUALS F 0 FORMAL EDU UCLA CS COMPILERS AVRORA AVRORA SIM
PLATFORM SENSORS ACCEL SENSOR POWER PTR SUB IR VAR 2 POINTS LBRACKET THIS 0 OLD THIS UPDATE Z
RBRACKET FORMAL Z EDU UCLA CS COMPILERS AVRORA AVRORA SIM PLATFORM SENSORS ACCEL SENSOR ADC INPUT
```

Figure 3.3: Example resultant of applying transformations 1 to 3 to the original code.

3.3.1 Tokenizer

Before feeding the processed text to the machine learning model, first, it must be tokenized, that is, transformed into a sequence of words. Then, we transform each word into a numerical index that

represents it. For this we use Keras[2] implementation of a Tokenizer [3] for preprocessing the text, turning each text into a sequence of integers (each integer being the index of a token in a dictionary).

We also use this tokenizer implementation to perform the fourth transformation discussed in this section. This is done by setting the NUM_WORDS parameter when creating the tokenizer. This allows it to only consider the most common words in the training set, by only keeping the most common NUM_WORDS-1 words and replacing the rest with the token UNK.

3.4 Machine Learning Model

To efficiently find and locate false positive patterns in source code several works have successfully used machine learning approaches in the past. In a recent work several machine learning techniques were compared when used to identify false positives in the static analysis context [52]. In this work, the authors concluded that RNNs obtained the best results, in particular LSTM networks. These networks, with their proven ability to model long sequences [39], achieved the best accuracy, precision and recall when tested in a dataset of 400 static analysis reports.

For these reasons, we use LSTM networks to automatically learn features from token vectors extracted from source code, and then utilize those features to build false positive probability prediction and warning classification models. In this section we detail how the model is built, explaining all the decisions made on the choice of layers and parameters, and the measures chosen to evaluate it.

3.4.1 LSTM Network Layers

Our solution follows the architecture specified in Figure 3.4. Each of the layers present in the LSTM network have the following purpose:

- **Embedding layer:**
  - This layer turns positive integers (indexes) into dense vectors of fixed size.
  - A word embedding is an approach to representing words and documents. This effectively diminishes the complexity of the data and eliminates sparse vectors.
  - In sum, it transforms the sparse index-based representation of words into a dense representation.
- **LSTM layer:**
  - Learns long-term dependencies between time steps in time series and sequence data.
  - This layer performs additive interactions, which can help improve gradient flow over long sequences during training.

[2] https://www.tensorflow.org/api_docs/python/tf/keras
[3] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer

- **Fully Connected Layer:** (Dense layer)
  - Applies an activation function to an output.
  - Applies softmax probability prediction and enables classification.

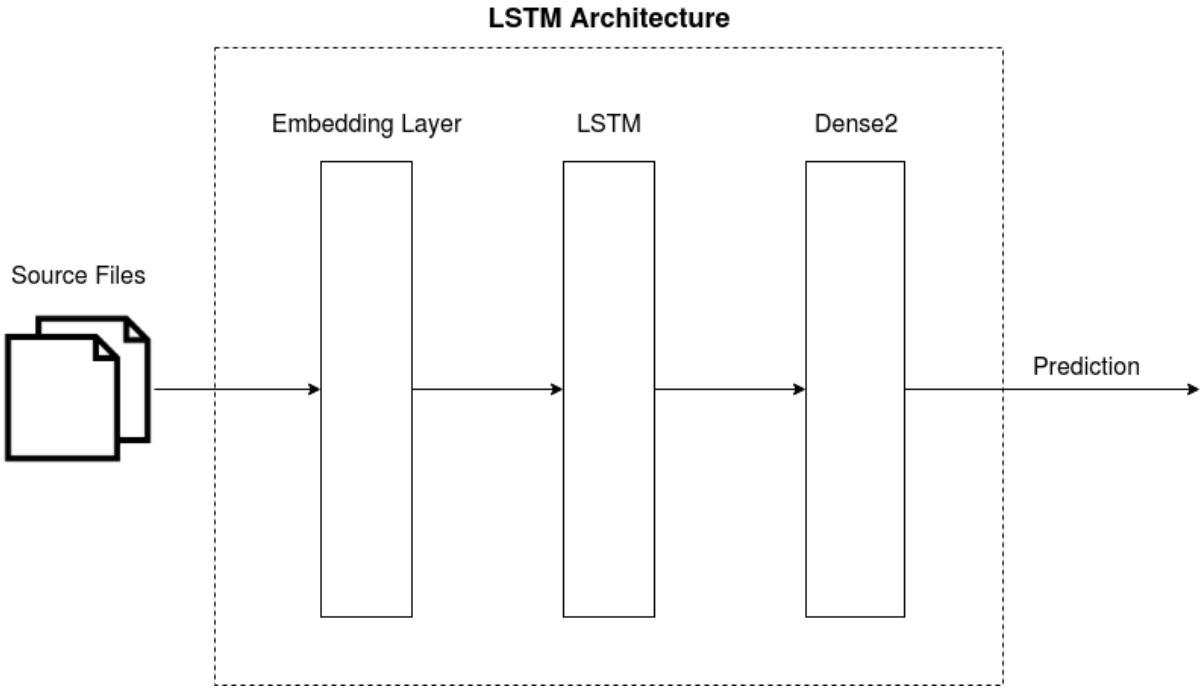

Figure 3.4: Architecture of the model

3.4.2 LSTM Network Parameters

Parameters constituent of each layer are of key importance when building a machine learning model, and make the difference between efficient or unreliable models. In this section we discuss the relevant parameters in an LSTM network.

- **Embedding layer:**
  - **Number of features:**
    This corresponds to the number of unique tokens per dataset. It encapsulates the vocabulary of a given dataset.
  - **Embedding size:**
    The output dimension of the layer. It determines the size of the dense vector.
  - **Maximum sequence length:**
    Maximum size of the input data.

$$output_{size} = \#vocabulary_{size} * \#Embedding_{size} \tag{3.1}$$

- **LSTM layer:**
  - **Number of Units:**

    Positive integer, dimensionality of the output space.
  - **Activation:**

    Activation function to use. Default: hyperbolic tangent (tanh).

$$output_{size} = 4 * (\#Embedding_{size} + \#units) * \#units \tag{3.2}$$

- **Fully Connected Layer:** (Dense layer)
  - **Number of units:**

    Number of hidden units in a layer
  - **Activation:**

    Activation function to use. In our case we will use softmax activation function, which is why the output shape is 2.

$$output_{size} = \#units * \#prev\_units + 2 \tag{3.3}$$

Here is a example of the model summary used during false positive report classification:

```
Model: "data_rnn"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 128)         256000
_________________________________________________________________
recurrent (LSTM)             (None, 128)               131584
_________________________________________________________________
output (Dense)               (None, 2)                 258
=================================================================
Total params: 387,842
Trainable params: 387,842
Non-trainable params: 0
```

Figure 3.5: Summary of the model

3.4.3 Machine Learning Model Evaluation

In this section we define the metrics to evaluate our LSTM network approach in correctly classifying Infer Static Analysis tool warnings based on their probability of being false positives. These metrics allow us to compare our proposed model and approaches to state-of-the-art static analysis report classification techniques.

The metrics used to evaluate our model are the common standards used to evaluate machine learning models. Therefore, we consider:

- Accuracy
- Precision
- Recall

Accuracy is represented as the number of correctly classified samples divided by the set of all samples (test set). It is a good indicator of effectiveness for our study since we have an even distribution of samples for each class. More formally:

$$Accuracy = \frac{tp + tn}{tp + fp + tn + fn} \tag{3.4}$$

Precision is the ratio of correctly classified faulty samples among all samples classified as faulty. It is particularly useful when the cost of reviewing false positive reports is unacceptable. More formally:

$$Precision = \frac{tp}{tp + fp} \tag{3.5}$$

Recall is the number of faulty samples correctly classified by the evaluated model divided by the total number of faulty samples. It is important when missing a true positive report is unacceptable (e.g., when analyzing safety-critical systems). More formally:

$$Recall = \frac{tp}{tp + fn} \tag{3.6}$$

All these metrics are composed of three parameters. These parameters correspond to the number of elements that are regarded as:

- **True Positives (tp) :** Total number of reports that were predicted to be defective and are actually defective.
- **True Negatives (tn) :** Total number of reports that were predicted to not be defective and are actually not defective
- **False Positives (fp) :** Total number of reports that were predicted to be defective but are actually not defective.
- **False Negatives (fn) :** Total number of reports that were predicted to not be defective but are actually defective.

We comment on the obtained experimental results based on these metrics and draw conclusions about the proposed model. We compare the developed models under the same conditions i.e, to the same dataset and parameters. All three metrics are computed using the test portion of the datasets.

3.5 Infer integration

We develop an application that complements the analysis performed by Infer Static Analysis tool. We do this by utilizing the Neural Language Model discussed in this section and pair it with two other programs.

The idea consists in retrieving the original output given by Infer and trace the warnings to the line they originated from. From there, the information present in the node and session responsible for raising the warning is captured and preprocessed. The data is then fed into the neural language model which returns the probability of the input warning to be a false positive.

Finally, the output is ranked accordingly placing true positives over false positives and given to the user.

3.6 Solution Requirements

In order to implement the solution we presented in this chapter, we took advantage of several different technologies. To reproduce the results the following tools are required:

- Java Programming Language version 1.8
- Infer version 0.17.0
- Python Programming Language version 3.5.2
- Maven version 3.3.8
- Ant version 1.10.7
- Keras Library with TensorFlow version 2.2.0 backend

The dataset with all the programs of the benchmark and their respective analysis can be found at https://github.com/joaofranciscomartins/infer_warning_classification.

The developed application's code and the datasets used to train the model can be found at https://github.com/joaofranciscomartins/infer_output_prioritization .

4

Experimental Results

Contents

In this chapter we detail the experimental evaluation and discussion of the results of the proposed solution concerning Static Analysis tool warnings classification. We begin by laying out the test environment, then we describe the different LSTM approaches to be studied as well as two different application scenarios and the creation of two different datasets to address each scenario, then we cover the experiments made to achieve an optimal model parameter configuration and finally, we present the results for each model variation for the two different scenarios. These results are then discussed and analysed and the research questions are answered.

4.1 Experimental Environment

All the experiments were carried out in a Debian 10 server composed by a 32-core Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz and 64GB of RAM, and in a Debian 10 server composed by a 24-core Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz and 64GB of RAM.

The machine learning model implementation is done with the Keras Library using a Tensorflow backend. All the datasets are obtained from the previously mentioned repositories in Section 3.6.

4.1.1 LSTM approaches

In this work, we study how much each transformation applied to the dataset in Section 3.3 impacts the performance of the model. Each approach consists in the set of transformations mentioned in the table below.

Table 4.1: LSTM approaches

Applied Transformations	Approach Name
T_{cln}	*LSTM-CLN*
T_{cln} + T_{ans}	*LSTM-ANS*
T_{cln} + T_{ans} + T_{ext}	*LSTM-EXT*
T_{cln} + T_{ans} + T_{ext} + T_{aps}	*LSTM-APS*

4.1.2 Application Scenarios

Most program defect predicting studies have been conducted under two different scenarios [52, 61]. We evaluate our approach considering these two types of scenarios:

1. Within-project false positive classification
2. Cross-project false positive classification

The first scenario considers the case where developers might continuously run static analysis tools on the same set of programs as those programs evolve over time i.e they use Infer for 'diff' time analysis as explained in Section 2.3. In this scenario, the models might learn signals that specifically appear in those programs, certain identifiers, API usage, etc. To mimic this scenario, we divided our benchmark randomly into training and test sets. Doing so, both training and test sets will have samples from each program in the dataset. We refer to the random-wise split benchmark as *B-Rand* for short.

The second scenario concerns the case where developers might want to deploy static analysis on a new subject program. In this scenario, the training would be performed on one set of programs, and the learned model would be applied to another. To emulate this scenario, we divided the programs randomly so that a collection of programs form the training set and the remaining ones form the test set. We refer to the program-wise split benchmark as *B-Prog* for short.

4.2 Parameter Tuning

In this section we introduce the choices that lead to the optimal configuration of our machine learning model. The parameters which had the biggest impact in the performance of the model are: the size of the embedding, the number of LSTM layer units and the batch size. We chose the optimal values for each parameter where the model attained the highest Accuracy measure. We trained every model for 20 epochs with a patience of 5 epochs, using the B-Rand dataset and the LSTM-APS approach.

The following Figures 4.1, 4.2 and 4.3 show the variance of the Accuracy metric by each of the three parameters:

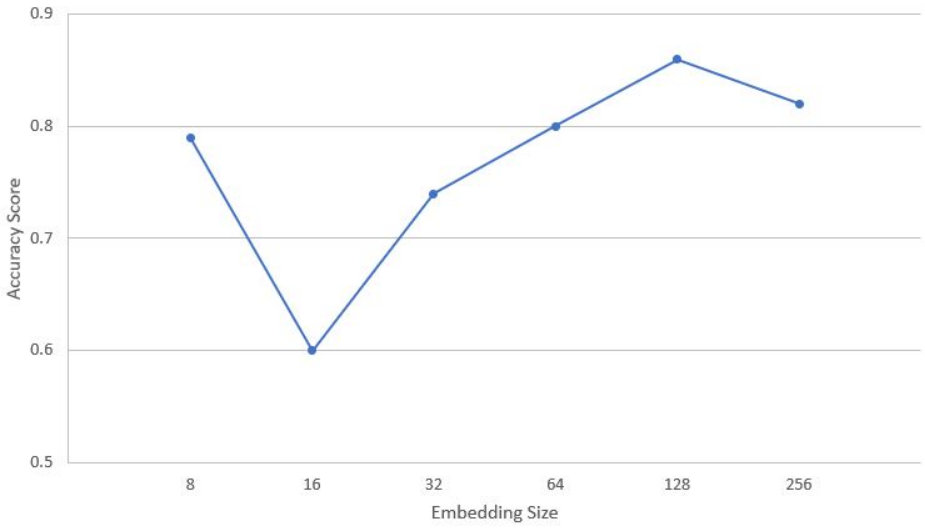

Figure 4.1: Variation of the Accuracy score, considering the word embedding size

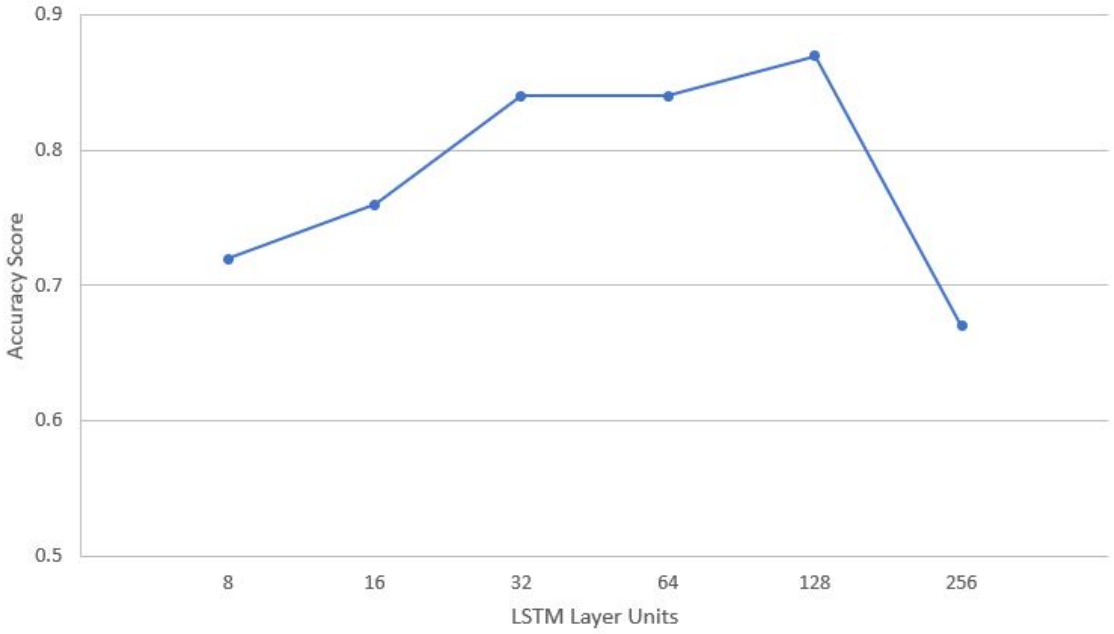

Figure 4.2: Variation of the Accuracy score, considering the size of the LSTM layer

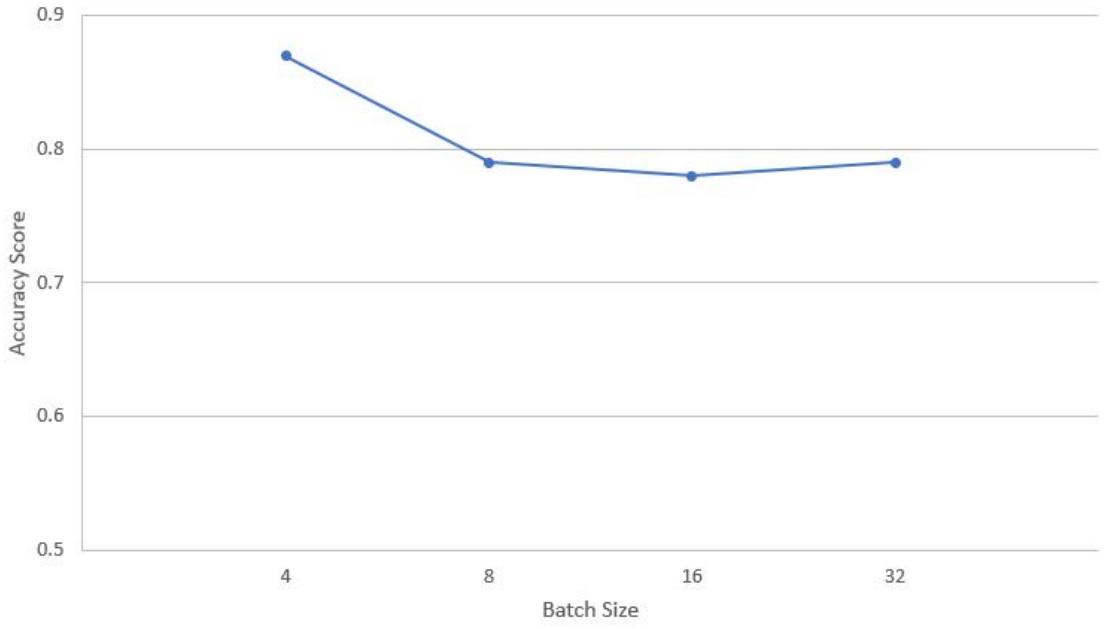

Figure 4.3: Variation of the Accuracy score, considering the batch size

First, let us look at the variance of the Accuracy measure depending on the word embedding size. The analysis of the Figure 4.1 is clear. The accuracy value first decreases to 0.60 at size 16, but after that it steadily increases until embedding size 128 when it starts to decrease again. Thus, considering this scenario the optimal choice is an embedding word size of 128.

Second, we analyse the variance of the Accuracy measure depending on the number of LSTM layer units. The analysis of the Figure 4.2 is straightforward. The accuracy value increases until 128 units, from there is starts decreasing again. Thus considering this scenario, the optimal choice for the number of LSTM layer units is 128.

Finally, we analyse the variance of the Accuracy measure depending on the batch size. As can be seen in the chart, the accuracy value steadily decreased from 4 until 32. We therefore concluded that the optimal choice is a batch size equal to 4. We also tested with batch size 2, but the training time was too high comparing to the other approaches and therefore it was not considered.

In summary, the optimal configuration used for testing corresponds the following parameter settings listed in Table 4.2.

Table 4.2: Optimal configuration of the LSTM network

Parameters	Optimal Value
Embedding Size	128
Number of LSTM units	128
Batch Size	4

4.3 Results

4.3.1 Training Configuration

Evaluating machine learning algorithms requires separation of the data points into a training set, used to estimate model parameters and a test set, used to evaluate the classifier's performance. The training method utilized was k-fold cross validation. As illustrated in Figure 4.4, the total data set is split in k sets. One by one, a set is selected as test set and the k-1 other sets are combined into the corresponding training set. This is repeated for each of the k sets. A set of data is stored as the validation set where the model is tested to measure the accuracy of the obtained solution.

For both application scenarios mentioned in the previous Subsection 4.1.2 we perform 5-fold cross validation. Furthermore, we repeat each execution 5 times with different random seeds. The purpose of these many repetitions (5-fold cross-validation $\times$ 5 random seeds = 25 runs) is to evaluate whether the results are consistent.

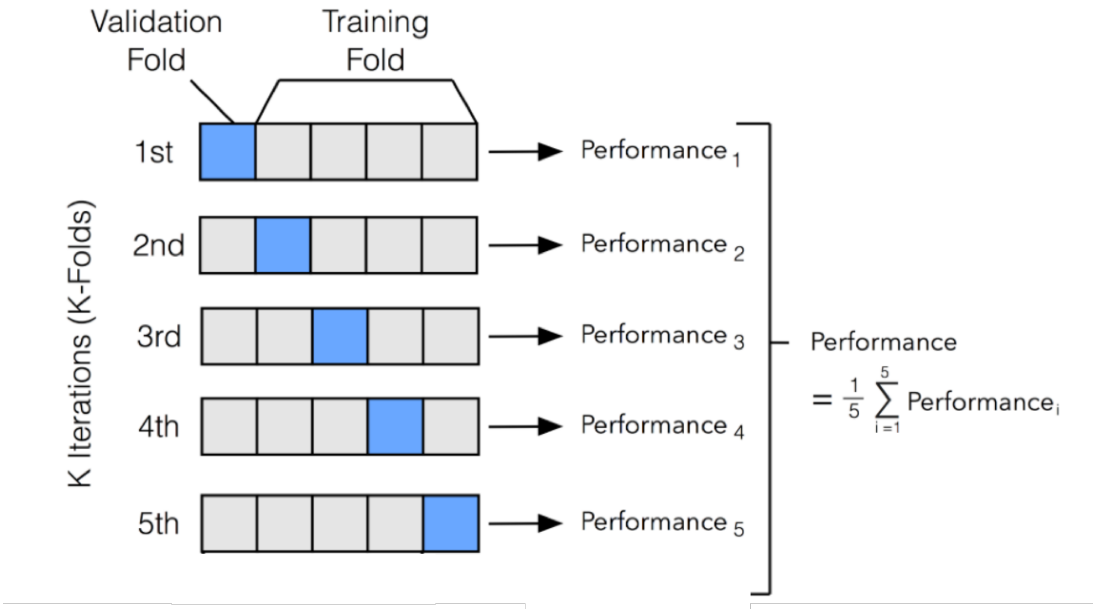

Figure 4.4: K-fold cross validation technique. (Source: [62])

4.3.2 Analysis of Results

Dataset	Approach	Accuracy	Recall	Precision
B-Rand	*LSTM-APS*	85.80 0.40	85.20 1.05	82.05 0.50
	LSTM-EXT	85.65 1.40	84.21 5.65	84.88 1.70
	LSTM-ANS	72.51 5.50	66.89 9.50	74.52 3.50
	LSTM-CLN	65.29 2.50	52.91 8.50	61.15 3.00
B-Prog	*LSTM-APS*	66.00 4.82	39.24 3.96	35.00 6.96
	LSTM-ANS	60.39 7.50	40.00 9.50	36.27 5.00
	LSTM-EXT	57.60 7.50	26.95 5.50	29.39 5.00
	LSTM-CLN	51.59 5.00	38.26 8.50	37.40 9.00

Table 4.3: Measure results sorted by Accuracy.

In total we trained 200 Infer warning classification models: 2 different datasets $\times$ 5 splits $\times$ 5 random seeds $\times$ 4 data preparation routines for the LSTM network. The summary of the results can be found in Table 4.3, as the median and Semi-interquartile Range (SIQR) of 25 runs. Numbers in bigger font are the median, and numbers in smaller font are the SIQR. We report median and SIQR as we do not make any assumptions on the underlying distribution of the data and also, we want to be able to directly compare our results with state-of-the-art SA tools report classification models.

We now answer each research question.

RQ1: What is the overall performance of the model in classifying Infer Static Analysis Tool warnings?

In this section we analyse the overall performance of our model using the Accuracy score and make a comparison with state-of-the-art SA tools report classification models.

The different LSTM approaches used by Koc et. al [52], when analysing the FindBugsSec SA tool warnings on a real-world random-wise split dataset, achieved a maximum Accuracy score of 89.33% and a maximum of 80.00% Accuracy score when analyzing a real-world program-wise split dataset.

By using identical LSTM approaches, our model obtained 85.80% and 66.00% Accuracy score on the B-Rand and B-Prog datasets respectively, as stated in Table 4.3. These values indicate that our model is effective when modeling long sequences as can be seen in Table 4.4, as well as in accurately modeling the language, thus allowing the finding of false positives patterns in the symbolic execution performed by Infer. These results also validate the capability of this model of correctly classifying Infer output reports.

However, our model did not match the results achieved by the work mentioned above [52], as our model's maximum Accuracy score was lower for both random and program wise split datasets. This could be due to a few reasons:

1. Only 11% of the false positives warning samples are of the RESOURCE LEAK type. This makes it harder for the model to correctly identify false positives patterns correspondent to such errors in the symbolic execution since there are not that many spurious samples. This causes for the overall Accuracy score of the model to drop.
2. As can be seen in Table 4.4, we are dealing with relatively long sequences, with a maximum of up to 17 700 words. Since the sequences are so long and we do not have a large number of samples, it is harder for the neural language model to correctly capture all the features of the language and its dependencies. This ultimately takes a toll in the Accuracy score of the model, since its harder to identify patterns in the language.

RQ2: What is the effect of different data preparations on performance?

We now analyse the effect of the different data preparations on the performance of the machine learning model. The data preparation process is done with the objective of providing the best use for the information present in Infer's symbolic execution. *We found that LSTM-APS data preparation provided the best accuracy results for both variations of the dataset.* The main reason for this result is the fact that on top of all the previous preparations that were applied, which include basic cleansing, abstracting variable identifiers and splitting class paths, it removes program specific words increasing furthermore the level of abstraction. The difference between the *LSTM-APS* and *LSTM-EXT* is not as clear when

analyzing the B-Rand dataset, due to the fact that both train and test sets contain warning samples from every program present in the dataset. However, this difference becomes noticeable when analyzing the B-prog dataset results, with *LSTM-APS* outperforming *LSTM-EXT* by 8.40% in Accuracy. The reason for this difference is that the program under evaluation is not present in the training set, which makes vocabulary abstraction more relevant to deal with program-specific words.

The *LSTM-CLN* approach got the lowest Accuracy score in both datasets. This is due to the simple fact that the level of abstraction is to low to correctly identify false positive patterns even between samples of the same program.

It is also worth mentioning that *LSTM-ANS* approach achieved a better score in the B-program dataset than the *LSTM-EXT*. This happens because the sequence length increases substantially from the *LSTM-ANS* approach to the *LSTM-EXT* as seen in Table 4.4 and it is aggravated by the fact that the programs under evaluation are not present in the training set, meaning that they have a bigger number of unknown words. This makes it harder for the model to identify patterns, and ultimately resulting in an Accuracy score drop.

RQ3: What is the variability in the results?

By using the SIQR values present in Table 4.3, we analyse the variance in the Accuracy, Recall and Precision measures.

On the B-Rand dataset, SIQR values are relatively low for all approaches, except for the recall values in the *LSTM-ANS* and *LSTM-CLN* approaches. The *LSTM-APS* approach obtained the minimum variance for accuracy, recall and Precision, followed by the *LSTM-EXT* approach.

On the B-Prog dataset, the variance is in general bigger than in the previous dataset. With a maximum SIQR value obtained of 7.50, 9.50 and 9.00 for Accuracy, Recall and Precision respectively. Again the *LSTM-APS* approach obtained the minimum variance across all metrics followed by *LSTM-EXT*.

Overall, we conclude that applying more data preparations to the datasets usually leads to a smaller variance for all the three metrics in both case scenarios.

RQ4: How do different application scenarios impact the performance of the model??

The maximum Accuracy score achieved is clearly higher for the B-Rand dataset scenario, reaching to a score of almost 86% against the 66% achieved in the B-Prog dataset scenario.

The almost 20% Accuracy difference between case scenarios is caused by the very nature of the programs under analysis which are complex and considerably large. This complexity and size means it is hard to capture false positives patterns in a program never seen before, with new vocabulary and dependencies. Although this effect is mitigated by the fact that we are analyzing Infer's symbolic execution which is somewhat regular, the results still show the consequences of analyzing a new subject program.

Our model does not demonstrate significant effectiveness when dealing with cross-project warning

classification, regarding Infer output warnings.

Table 4.4: Dataset size statistics for each LSTM approach

Approach	Dictionary Size	Minimum Sequence Length	Maximum Sequence Length
LSTM-CLN	10805	418	13960
LSTM-ANS	3807	404	12955
LSTM-EXT	3189	501	17610
LSTM-APS	2000	491	17599

4.3.3 Output Prioritization

In this section we present the output of our application as described in Section 3.5 and compare it with the original output of the Infer when analysing the same program. As can be seen in Figure 4.5, Infer warnings are prioritized according to their probability of being false positives, thus reducing the number of spurious warnings in the first lines of the output, having 0 False positives in the first 10 warnings. Infer in the first 10 output entries, Figure 4.6, reports a total of 5 false positive warnings. This example shows the effect and importance of warning prioritization. It also allows the programmer to choose with more precision which warnings to review since the probabilities are discriminated.

```
src/org/apache/xalan/xsltc/util/JavaCupRedirect.java:63: error: RESOURCE_LEAK
Probability of being a False Positive: 0.06495786
src/org/apache/xalan/xslt/EnvironmentCheck.java:134: error: RESOURCE_LEAK
Probability of being a False Positive: 0.09818842
src/org/apache/xalan/xsltc/trax/TransformerFactoryImpl.java:1305: error: RESOURCE_LEAK
Probability of being a False Positive: 0.10622824
src/org/apache/xalan/xsltc/trax/TransformerFactoryImpl.java:1312: error: RESOURCE_LEAK
Probability of being a False Positive: 0.10622824
src/org/apache/xalan/xsltc/trax/TransformerFactoryImpl.java:1209: error: RESOURCE_LEAK
Probability of being a False Positive: 0.11100773
src/org/apache/xalan/xsltc/trax/TransformerFactoryImpl.java:1164: error: RESOURCE_LEAK
Probability of being a False Positive: 0.11100773
src/org/apache/xalan/xsltc/runtime/AbstractTranslet.java:561: error: RESOURCE_LEAK
Probability of being a False Positive: 0.16833092
src/org/apache/xalan/xsltc/compiler/Key.java:90: error: NULL_DEREFERENCE
Probability of being a False Positive: 0.2923071
src/org/apache/xalan/xsltc/dom/DOMAdapter.java:184: error: NULL_DEREFERENCE
Probability of being a False Positive: 0.2968096
src/org/apache/xalan/xsltc/dom/DOMAdapter.java:249: error: NULL_DEREFERENCE
Probability of being a False Positive: 0.3001589
```

Figure 4.5: First 10 warnings of the prioritized output.

```
src/org/apache/xalan/xsltc/runtime/output/WriterOutputBuffer.java:38: error: NULL_DEREFERENCE
src/org/apache/xalan/extensions/XPathFunctionResolverImpl.java:61: error: NULL_DEREFERENCE
src/org/apache/xalan/xsltc/util/JavaCupRedirect.java:63: error: RESOURCE_LEAK
src/org/apache/xalan/xsltc/compiler/FormatNumberCall.java:59: error: NULL_DEREFERENCE
src/org/apache/xalan/xsltc/compiler/ApplyImports.java:65: error: NULL_DEREFERENCE
src/org/apache/xml/serializer/SerializerBase.java:71: error: NULL_DEREFERENCE
src/org/apache/xalan/xsltc/compiler/ApplyImports.java:79: error: NULL_DEREFERENCE
src/org/apache/xalan/xsltc/compiler/ApplyImports.java:83: error: NULL_DEREFERENCE
src/org/apache/xalan/xsltc/compiler/Key.java:90: error: NULL_DEREFERENCE
src/org/apache/xalan/xsltc/trax/TrAXFilter.java:116: error: NULL_DEREFERENCE
```

Figure 4.6: First 10 warnings of Infer's original output.

4.4 Summary

This chapter described the testing environment, presented the different model approaches tested in different application scenarios, described how the optimal model configuration was obtained, discussed the experimental results and compared them with state-of-the-art approaches. The LSTM network neural language model achieved good results when dealing with within-project false positive classification, which is the most common use case for the Infer Static Analysis Tool. Our approach was not able to match the results obtained by previous works when using similar techniques. This is mainly due to the lack of spurious samples regarding a specific type of error as well as dealing with long-sized text sequences without an abundance of samples. All the decisions made were able to produce an effective model, capable of correctly identifying false positive Infer warnings, however, the results achieved do not represent a significant improvement over current techniques.

5

Conclusions

Contents

Infer is one of the most relevant Static Analysis tools of today. In this work we contributed to improve the usability and relevancy of this tool by identifying and filtering false positive warnings from its output. We presented a Neural Language Model, based on the use of LSTM networks, which allows to successfully model the symbolic execution performed by Infer while analysing a program, enabling the identification of false positive patterns in the analysis. We enumerated state-of-the-art static analysis tools warnings false positives identification techniques. We also introduced a dataset of more than 500 Infer Static Analysis Tool warnings labeled as false/true positive that can serve as a reference dataset to the research community. In our experiments, we studied four different data preparations as input for the language model in two different contexts with the end purpose of prioritizing Infer output by filtering false positives from true positives. We discussed the results and got important remarks and insights.

Using our model, we were able to successfully classify Infer's output, with a percentage of 86% correctly classified warnings when dealing with within-project warning classification, which is the most common use case for Infer. As well as that, we created an application that combines the proposed model with Infer Static Analysis tool which sorts the output by prioritizing true positives over false positives. We also observed in both application scenarios that more detailed data preparation with abstraction and word extraction leads to significant increases in accuracy. Finally, we observed that the second application scenario, with cross-project false positive classification, is more challenging since it is required to learn patterns of true/false positive reports that can hold across different programs.

5.1 Future Work

In future work, we believe that there are some areas of improvement regarding this subject that should be explored.

Firstly, the construction of a larger dataset with more samples and with balanced data, i.e., with a significant number of samples of false positive warnings for every type of error identified by the tool, so the neural network is more efficiently trained and also with higher confidence. This should allow the model to be more generalizable focusing specifically on Infer's symbolic execution false positive patterns and less on program specific words and also for the model to have enough spurious warnings examples, so it can confidently report the nature of any kind of error detected by the tool

Secondly, Infer supports the analysis of a variety of languages and would be interesting to find out whether if the model here developed is generalizable enough to be capable of correctly modeling the symbolic execution that Infer performs for analyzing programs written in other languages than Java.

Lastly, having a different strategy towards the parsing phase, where less data is extracted from Infer's symbolic execution. This increasing of abstraction and decrease in specificity may result in a model that is highly capable of making prediction for a given dataset, especially when performing cross-project false

positive classification where the programs under evaluation are different from the ones present in the training set.

Bibliography

[1] C. Calcagno, D. Distefano, and P. O'Hearn, "Open-sourcing facebook infer: Identify bugs before you ship," *code. facebook. com blog post*, vol. 11, 2015.

[2] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2002, pp. 55–74.

[3] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *International Workshop on Computer Science Logic*. Springer, 2001, pp. 1–19.

[4] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *International Static Analysis Symposium*. Springer, 2003, pp. 295–315.

[5] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[6] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[7] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[8] "Checkstyle - static code analysis tool used in software development for checking if Java source code complies with coding rules." https://checkstyle.sourceforge.io/.

[9] "PMD - A static ruleset based source code analyzer that identifies potential problems." https://pmd.github.io/.

[10] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Communications of the ACM*, vol. 62, no. 8, pp. 62–70, 2019.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[12] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” 2018.

[13] C. Calcagno and D. Distefano, “Infer: An automatic program verifier for memory safety of c programs,” in *NASA Formal Methods Symposium*. Springer, 2011, pp. 459–465.

[14] S. Blackshear, N. Gorogiannis, P. W. O’Hearn, and I. Sergey, “Racerd: compositional static race detection,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 144, 2018.

[15] “Defects4J - collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research.” https://github.com/rjust/defects4j.

[16] F. Jelinek and R. L. Mercer, “Interpolated estimation of Markov source parameters from sparse data,” in *Proceedings, Workshop on Pattern Recognition in Practice*, E. S. Gelsema and L. N. Kanal, Eds. Amsterdam: North Holland, 1980, pp. 381–397.

[17] S. Katz, “Estimation of probabilities from sparse data for the language model component of a speech recognizer,” *IEEE transactions on acoustics, speech, and signal processing*, vol. 35, no. 3, pp. 400–401, 1987.

[18] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, “A neural probabilistic language model,” *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[19] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[20] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, “Recurrent neural network based language model,” in *INTERSPEECH*, 2010.

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

[22] M. Sundermeyer, R. Schlüter, and H. Ney, “Lstm neural networks for language modeling,” in *Thirteenth annual conference of the international speech communication association*, 2012.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[24] U. Khandelwal, H. He, P. Qi, and D. Jurafsky, “Sharp nearby, fuzzy far away: How neural language models use context,” *arXiv preprint arXiv:1805.04623*, 2018.

[25] J. Howard and S. Ruder, “Universal language model fine-tuning for text classification,” *arXiv preprint arXiv:1801.06146*, 2018.

[26] P. Ramachandran, P. J. Liu, and Q. V. Le, “Unsupervised pretraining for sequence to sequence learning,” *arXiv preprint arXiv:1611.02683*, 2016.

[27] H. Babii, A. Janes, and R. Robbes, “Modeling vocabulary for big code machine learning,” *arXiv preprint arXiv:1904.01873*, 2019.

[28] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[29] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” *arXiv preprint arXiv:1711.00740*, 2017.

[30] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[31] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” *arXiv preprint arXiv:1808.01400*, 2018.

[32] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[33] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with lstm,” 1999.

[34] D. P. Mandic and J. Chambers, *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. John Wiley & Sons, Inc., 2001.

[35] Y. Goldberg, “Neural network methods for natural language processing,” *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017.

[36] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[37] M. Rahman, Y. Watanobe, K. Nakamura *et al.*, “Source code assessment and classification based on estimated error probability using attentive lstm language model and its application in programming education,” *Applied Sciences*, vol. 10, no. 8, p. 2973, 2020.

[38] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[39] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," 2014.

[40] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 39–40.

[41] K. Terada and Y. Watanobe, "Code completion for programming education based on recurrent neural network," in *2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)*. IEEE, 2019, pp. 109–114.

[42] J. Reyes, D. Ramírez, and J. Paciello, "Automatic classification of source code archives by programming language: A deep learning approach," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2016, pp. 514–519.

[43] Y. Teshima and Y. Watanobe, "Bug detection based on lstm networks and solution codes," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2018, pp. 3541–3546.

[44] X. Ye, F. Fang, J. Wu, R. Bunescu, and C. Liu, "Bug report classification using lstm architecture for more accurate software defect locating," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 1438–1445.

[45] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *arXiv preprint arXiv:1608.02715*, 2016.

[46] Y. Wang, M. Huang, X. Zhu, and L. Zhao, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 606–615.

[47] H. Chen, M. Sun, C. Tu, Y. Lin, and Z. Liu, "Neural sentiment classification with user and product attention," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 1650–1659.

[48] U. Yüksel and H. Sözer, "Automated classification of static code analysis alerts: a case study," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 532–535.

[49] 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*. ACM, 2014, pp. 762–774.

[50] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, “Predicting accurate and actionable static analysis warnings: an experimental approach,” in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 341–350.

[51] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, “Learning a classifier for false positive error reports emitted by static code analysis tools,” in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 2017, pp. 35–42.

[52] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, “An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 288–299.

[53] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, “The DaCapo benchmarks: Java benchmarking development and analysis,” in *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[54] ——, “The DaCapo Benchmarks: Java benchmarking development and analysis (extended version),” Tech. Rep. TR-CS-06-01, 2006, http://www.dacapobench.org.

[55] “Cloc (Count Lines Of Code) - counts blank lines, comment lines, and physical lines of source code,” https://github.com/AlDanial/cloc.

[56] “Apache Tomcat - an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies.” https://tomcat.apache.org.

[57] “Apache Xalan - develops and maintains libraries and programs that transform XML documents using XSLT standard stylesheets.” https://xalan.apache.org/.

[58] “Avrora - The AVR Simulation and Analysis Tool.” http://compilers.cs.ucla.edu/avrora/.

[59] “Joda-Time - a quality replacement for Java date and time classes.” http://www.joda.org/joda-time/.

[60] “Jython - Python for the Java Platform.” https://www.jython.org.

[61] 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 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 91–100.

[62] J. Ashfaque and A. Iqbal, “Introduction to support vector machines and kernel methods,” 04 2019.