# Prioritizing Facebook's Infer Static Analysis Tool Warnings

João Francisco Roberto Martins*
*Instituto Superior Técnico, Universidade de Lisboa*
joaofrmartins@tecnico.ulisboa.pt

*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.**

*Index Terms*—**Infer; Static Analysis; False Positive Classification; Output Prioritization; Machine Learning.**

## I. Introduction

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 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, 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.

### A. 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 Long Short-term Memory (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 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, 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 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.

## II. Background

### A. *Static Analysis and False Positives*

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.

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 ACM articles on industrial static analysis [6], [7]. However, 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 [8].

### B. *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 [9], to concurrency [10], to security (information flow), and many more specialized errors.

Infer's main deployment model is based on fast incremental analysis of code changes [8]. 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. When a *diff* is submitted an instance of Infer is run in Facebook's internal Continuous Integration 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. Infer will take 10min-15min to run on a *diff* on average. In contrast, when Infer is run in whole-program mode it can take more than an hour (depending on the app being analysed).

**Workflow**: There are always two phases of an Infer run, regardless of the input language (Java, Objective-C, or C). 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. 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/`. 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 and also in a file `infer-out/bugs.txt`.

### C. Neural Language Models

The state-of-the-art in Natural Language Processing is made of Neural Language Models [11]. A Neural Language Model 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 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.

### D. Long Short-term Memory Networks

For text classification Recurrent Neural Networks (RNNs) [12], [13] 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 [14]. 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 [15] that prevents standard RNNs from learning long-term dependencies.

Long Short-term Memory (LSTM) networks were first proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber [16], and are among the most widely used models in Deep Learning for Natural Language Processing 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.

Is for this reason that in this work use LSTM networks to build our Language Model.

### E. False Positive filtering using Machine Learning

Research efforts have successfully applied machine learning techniques to filter out the false positive error reports from other Static Analysis tools. In this section we present some of these works.

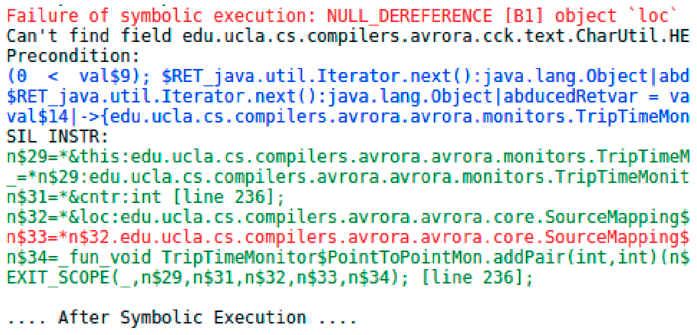
*Kremenek and Engler* [4] proposed z-ranking, a technique to rank error reports emitted by static program checking analysis tools. It employs a simple statistical model to rank those error messages most likely to be true errors over those that are least likely. *Yüksel and Sözer* [17] evaluated the application 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. *Tripp et al.* [18] 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. *Ruthruff, Joseph R., et al.* [19] reports automated support using logistic regression models that predicts the foregoing types of warnings from signals in the warnings and implicated code. 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.* [20] , 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. Later in 2019 a study by the same authors [21] 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.

In this work we try to apply some of these techniques to Infer's output.

## III. Infer Warning Priorization

### *A. 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 II-B, 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 then 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 1.



```
Failure of symbolic execution: NULL_DEREFERENCE [B1] object `loc`
Can't find field edu.ucla.cs.compilers.avrora.cck.text.CharUtil.HE
Precondition:
(0  <  val$9); $RET_java.util.Iterator.next():java.lang.Object|abd
$RET_java.util.Iterator.next():java.lang.Object|abducedRetvar = va
val$14|->{edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMon
SIL INSTR:
n$29=*&this:edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeM
_=*n$29:edu.ucla.cs.compilers.avrora.avrora.monitors.TripTimeMonit
n$31=*&cntr:int [line 236];
n$32=*&loc:edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$
n$33=*n$32.edu.ucla.cs.compilers.avrora.avrora.core.SourceMapping$
n$34=_fun_void TripTimeMonitor$PointToPointMon.addPair(int,int)(n$
EXIT_SCOPE(_,n$29,n$31,n$32,n$33,n$34); [line 236];

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

Fig. 1. Infer intermediate language example

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 III-B.

### *B. Data Processing*

Differently from many other Static Analysis 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 [21]. 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 [22] 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 [21], 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 Static Analysis 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
AVRORA AVRORA SIM MCU ADC PTR IN V PATH FIND
ON EEXP ADDR IR VAR 2 V PATH FIND CANNOT FIND
IR VAR 1 PROP F 6 DIFFERENT ADC F 5 DIFFERENT
V PATH RBRACKET LPARENTHESES IR VAR 2 RPARENTHESES
N 2 7 3 RPARENTHESES M EQUALS NULLIFY EDU UCLA
MICROCONTROLLER PTR SUB OLD M EQUALS F 3 FORMAL
MCU ATMEL MICROCONTROLLER PTR SUB IR VAR 0 EQUALS
FORMAL INT SUB OLD ADC CHANNEL EQUALS F 2
VOID IR VAR 1 EQUALS NULL UPDATE VOID IR VAR
1 FORMAL EDU UCLA CS COMPILERS AVRORA AVRORA
OLD THIS EQUALS F 1 FORMAL EDU UCLA CS COMPILERS
SENSOR PTR SUB ASP EQUALS F 0 FORMAL EDU UCLA
ACCEL SENSOR POWER PTR SUB OLD ASP EQUALS F 0
PLATFORM SENSORS ACCEL SENSOR POWER PTR SUB IR
RBRACKET FORMAL Z EDU UCLA CS COMPILERS AVRORA
```

Fig. 2. Example resultant of applying transformations 1 to 3 to the original code.

## C. 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 [21]. In this work, the authors concluded that Recurrent Neural Networks obtained the best results, in particular LSTM networks. These networks, with their proven ability to model long sequences [23], 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.

## D. 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.

The code for the developed application can be found here[1]

## IV. Tool 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.* [10]. 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.

The benchmark in this evaluation is constituent of 5 real-world programs, that cover a wide range of issues. We then analyze 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 analyze each one with Infer.
- The programs are under active development and are highly used in order to increase relevancy.

Table III. shows the programs we chose. All the entries of are from the Dacapo Benchmark [24], [25], except for Joda-Time that was used by *Koc et al. 2019* [21] as part of their proposed real-world benchmark. The number of Lines of Code (LoC) was calculated by using Cloc (version 1.84) [26].

TABLE I
Programs of the Benchmark

| **Program** | **Description** | **LoC** |
|---|---|---|
| **1.** Apache Tomcat | Implements Java Servlet [27] | 435438 |
| **2.** Apache Xalan Java | Transforms XML docs into HTML [28] | 205644 |
| **3.** Avrora | Simulation and Analysis Tool [29] | 92041 |
| **4.** Joda-Time | Date and time framework [30] | 94973 |
| **5.** Jython | Python for the Java Platform [31] | 945500 |

[1]https://github.com/joaofranciscomartins/infer_output_prioritization

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 II. To label a Static Analysis report, we first analyze 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.

TABLE II
INFER REPORT 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 |

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. All of the programs and respective analysis are located here[2].

## V. EXPERIMENTAL SETUP

In this section we discuss our experimental setup. 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.

### A. Test 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 used to trained the model are located here[3].

[2]https://github.com/joaofranciscomartins/infer_warning_classification
[3]https://github.com/joaofranciscomartins/infer_output_prioritization

### B. LSTM approaches

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

TABLE III
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* |

### C. Application Scenarios

Most studies that use machine learning techniques to classify false positives envision two different scenarios for the use of these types of models. We evaluate our approach considering these two 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 II-B. 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, looks on 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 forms the training set and the remaining ones form the test set. We refer to the program-wise split benchmark as *B-Prog* for short.

### D. Parameter Tuning

In this section we introduce the choices that lead to the optimal configuration of our machine learning model. The model parameters that we find relevant 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.

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

TABLE IV
OPTIMAL CONFIGURATION OF THE LSTM NETWORK

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

### *E. 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 classifier performance. The training method utilized was k-fold cross validation. First 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 V-C 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 × 5 random seeds = 25 runs) is to evaluate whether the results are consistent.

### *F. Metrics*

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

**Accuracy** (1) 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.

$$Accuracy = \frac{tp + tn}{tp + fp + tn + fn} \tag{1}$$

**Precision** (2) 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.

$$Precision = \frac{tp}{tp + fp} \tag{2}$$

**Recall** (3) 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).

$$Recall = \frac{tp}{tp + fn} \tag{3}$$

All these metrics composed of three parameters. These parameters are the counts 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.

## VI. ANALYSIS OF RESULTS

TABLE V
MEASURE RESULTS SORTED BY ACCURACY.

| 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 |

In total we trained 200 Infer warning classification models: 2 different datasets × 5 splits × 5 random seeds × 4 data preparation routines for the LSTM network. The summary of the results can be found in Table V, as the median and the 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 Static Analysis 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 [21], when analysing the FindBugsSec Static Analysis 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 V.* These values indicate that our model is effective when modeling long sequences as can be seen in Table VI, 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 [21], 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 VI, 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 VI 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 V, 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.

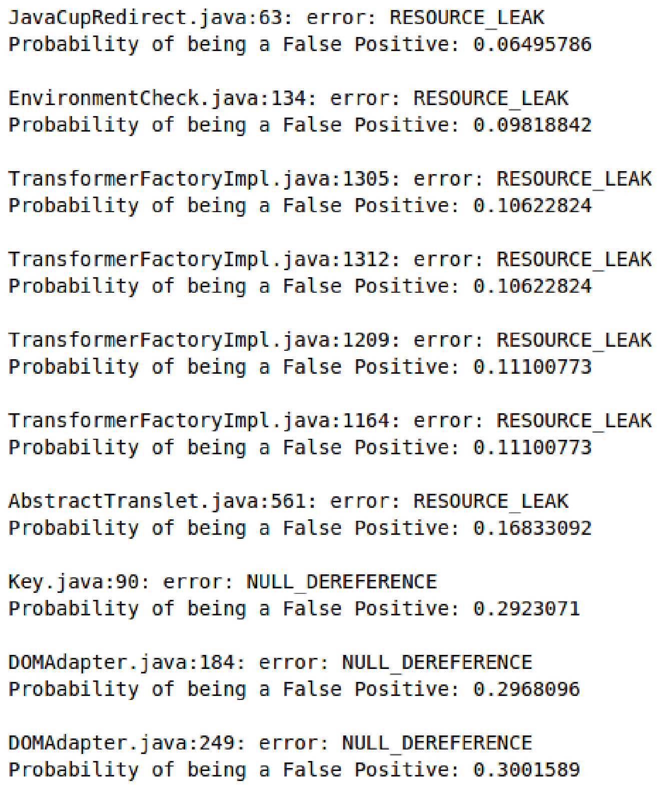
Our model does not demonstrate significant effectiveness when dealing with cross-project warning classification, regarding Infer output warnings.

TABLE VI
DATASET SIZE STATISTICS FOR EACH LSTM APPROACH

| **Approach** | **Dictionary Size** | **Min Seq Length** | **Max Seq Length** |
|---|---|---|---|
| *LSTM-CLN* | 10805 | 418 | 13960 |
| *LSTM-ANS* | 3807 | 404 | 12955 |
| *LSTM-EXT* | 3189 | 501 | 17610 |
| *LSTM-APS* | 2000 | 491 | 17599 |

## *A. Output Prioritization*

In this section we present the output of our application as described in Section III-D and compare it with the original output of Infer when analysing the same program. As can be seen in Figure 3, Infer warnings are prioritized according to



```
JavaCupRedirect.java:63: error: RESOURCE_LEAK
Probability of being a False Positive: 0.06495786

EnvironmentCheck.java:134: error: RESOURCE_LEAK
Probability of being a False Positive: 0.09818842

TransformerFactoryImpl.java:1305: error: RESOURCE_LEAK
Probability of being a False Positive: 0.10622824

TransformerFactoryImpl.java:1312: error: RESOURCE_LEAK
Probability of being a False Positive: 0.10622824

TransformerFactoryImpl.java:1209: error: RESOURCE_LEAK
Probability of being a False Positive: 0.11100773

TransformerFactoryImpl.java:1164: error: RESOURCE_LEAK
Probability of being a False Positive: 0.11100773

AbstractTranslet.java:561: error: RESOURCE_LEAK
Probability of being a False Positive: 0.16833092

Key.java:90: error: NULL_DEREFERENCE
Probability of being a False Positive: 0.2923071

DOMAdapter.java:184: error: NULL_DEREFERENCE
Probability of being a False Positive: 0.2968096

DOMAdapter.java:249: error: NULL_DEREFERENCE
Probability of being a False Positive: 0.3001589
```

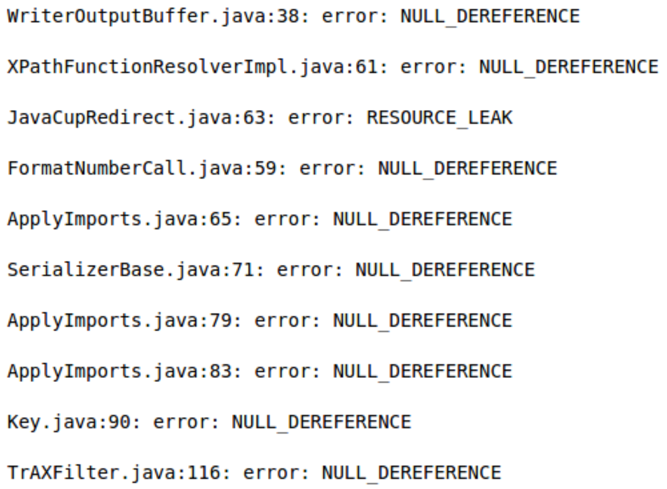
Fig. 3. First 10 warnings of the prioritized output.

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, 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.

## VII. Conclusion and Future Work

In this work we explored how Neural Language Models can be used to prioritize bug reports produced by Facebook's Infer. 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. Our proposed model is 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



```
WriterOutputBuffer.java:38: error: NULL_DEREFERENCE

XPathFunctionResolverImpl.java:61: error: NULL_DEREFERENCE

JavaCupRedirect.java:63: error: RESOURCE_LEAK

FormatNumberCall.java:59: error: NULL_DEREFERENCE

ApplyImports.java:65: error: NULL_DEREFERENCE

SerializerBase.java:71: error: NULL_DEREFERENCE

ApplyImports.java:79: error: NULL_DEREFERENCE

ApplyImports.java:83: error: NULL_DEREFERENCE

Key.java:90: error: NULL_DEREFERENCE

TrAXFilter.java:116: error: NULL_DEREFERENCE
```

Fig. 4. First 10 warnings of Infer's output.

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.

In future work, we plan to explore other types of Infer analyses with different programming languages. In addition, we plan to utilize a different strategy in the parsing phase, where fewer data is extracted from Infer's symbolic execution, making the model more abstract and less specific.

## References

[1] Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Open-sourcing facebook infer: Identify bugs before you ship. *code. facebook. com blog post*, 11, 2015.

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

[3] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.

[4] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*, pages 295–315. Springer, 2003.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[7] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. 2018.

[8] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[9] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[10] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):144, 2018.

[11] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

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

[14] Yoav Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017.

[15] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] Ulas Yüksel and Hasan Sözer. Automated classification of static code analysis alerts: a case study. In *2013 IEEE International Conference on Software Maintenance*, pages 532–535. IEEE, 2013.

[18] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 762–774. ACM, 2014.

[19] Joseph R Ruthruff, John Penix, J David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350. ACM, 2008.

[20] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam 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*, pages 35–42. ACM, 2017.

[21] Ugur Koc, Shiyi Wei, Jeffrey S Foster, Marine Carpuat, and Adam 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)*, pages 288–299. IEEE, 2019.

[22] Hlib Babii, Andrea Janes, and Romain Robbes. Modeling vocabulary for big code machine learning. *arXiv preprint arXiv:1904.01873*, 2019.

[23] Hasim Sak, Andrew W Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. 2014.

[24] 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*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[25] 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 (extended version). Technical Report TR-CS-06-01, 2006. http://www.dacapobench.org.

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

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

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

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

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

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