Ranking the output of static analysis

Kleidi Ismailaj

kleidi.ismailaj@student.uantwerpen.be

July 3, 2020

Contents

1 Abstract

Previous research has estimated the number of false positive static analysis alerts to be as high as 70% (**TO DO: CITE**). Research has also shown that developers loose trust and ignore such tools when the number of false positive (or unimportant) alerts is high. Given that static analysis is still helpful in promoting cleaner code, different aspects from the history of the codebase can be exploited to improve the relevance of the output. If the output is matched to what the developer finds important or what has been proven to be helpful in the past, this will lead to a better acceptance and wider usage of such tools.

2 Introduction

The amount of code that is being produced is increasing with time. That brings multiple challenges in the software engineering landscape, one of which is assuring adequate code quality. In this context, automatic approaches such as static analysis can be really useful and time saving in detecting and preventing potential bugs. Its usefulness has been acknowledged also by big tech companies, such as Google with its Tricorder architecture ([1]) or Facebook with its Infer static analyzer. Another reason to adopt SA techniques, is that the cost of repairing a defect is much lower when that defect is found early in the development cycle (generally accepted principle).

Even though it is a promising technique, in practice static SA results are far from perfect. Since the software under analysis is not executed, static analysis tools must speculate on what the actual program behavior will be, thus mistakes are inevitable. Heuristics or approximations are commonly used to determine properties for a given code construct, which may lead to inaccurate assumptions. A tool can also deliberately introduce approximations for scalability or speed, potentially resulting in an outbreak of errors.

Given the ever increasing amount of code and the tendency of SA tools to over-estimate possible faulty program behaviours, there is a need to improve the output of these tools. The above problem can be addressed either by increasing the precision of the analysis (and thus decrease the efficiency) or by post-processing the alarms effectively after they are generated, according to specific criteria. The classic approach most tools use for prioritizing and filtering results is to statically classify the results based on severity levels. They are oblivious of the actual code that is being analyzed and of the location or frequency of a given defect. Furthermore, it has been shown that if developers lose trust in the beginning, they then tend to ignore the output of these tools altogether (**TO DO: CITE**).

Different approaches have been proposed to improve the output of SA tools. Optimally, the initial error reports should be those most likely to be real errors. Alerts can be strategically prioritized for examination, by tracking warnings through a series of software versions, revealing which SA rules are more important and which parts of the software are more problematic. An understanding of how developers react on these alerts can help improve the utility of these tools. Alerts can be divided into two categories: actionable alerts (AA) to define a SA alert that the programmer would act on to resolve and unactionable alerts (UA) to define a SA alert that the programmer would not act on to resolve (**TO DO: CITE**). An unactionable alert may be: a trivial concern to fix, less likely to manifest at runtime, or incorrectly identified due to the limitations of the tool. We want thus to prioritize the AAs and hide the UAs.

Another approach is to prioritize alerts that in the past have pointed at bugs (**TO DO: CITE**). By tracking back the bugs up to a certain revision in the past, we can collect code lines that were changed during bug(**TO DO: CITE**) fixes. Subsequently, we can pinpoint which alerts warned about those specific parts of code and prioritize accordingly.

Given that for a relatively large project, the number of SA alerts can be prohibitive, the ultimate goal is to maximize the utility of time spent analyzing these alerts. Ranking schemes do not reduce alert investigation burdens if the aim is to check them all. Instead, they solve the problem by showing alerts that are most likely to be real/useful, so that developers can spend time by inspecting the most important ones.

Different approaches have been proposed in the literature but few have tried to do a comparison in terms of the utility of each method. Comparison is done among open source Java software and has the following problems... (**TO DO: FILL IN**). Also, these methods can be combined with the aim of achieving better results. It is important to do so because different techniques can be better suited to different types of alerts or can compensate for each others weaknesses.

Given an industrial code base, where SA tools have been abandoned because of the large amount of false positives, the goal of this thesis is to test the feasibility of successfully applying these ranking approaches even in a context with limited amount of data.

The rest of this thesis will be structured in the following sections... (**TODO**)

3 Literature review

***** HIDDEN FOR NOW TO KEEP THE VIEW CLEAN *****

This section will consist of research papers focused on three main topics:

- *Ranking static analysis alerts*: some of the most known (cited) approaches to rank alerts.
  - Using a single SA tool
  - Combining multiple tools
  - Looking at survey papers that describe the state of the research and state of the art techniques.
- *Bug prediction*: Rank the alerts in problematic parts of code higher.
- Information on *real world usage of SA tools*, problems and suggested solutions.

TODO: Insert the 2 papers about comparative studies

4 Problem statement

Improving the output of static analysis is dependent on a particular codebase and the people who produce it. Different organizations can have different priorities and expectations on code quality. Also developers or teams might be more interested on a particular subset of warnings (or the context on which the warnings appear).

Given an industrial codebase, the goal is to explore if these automatic techniques are useful, which produces the best results, and if an ensemble technique provides extra benefits. The starting point is the version control history of the project. By extracting information about the past versions, an attempt can be made to learn which SA alerts are more important and can be prioritized in the future. In contrast to open source project where you can test the approaches only on those project that have a sufficient amount of data, in an industrial codebase you have to make the most of the data you can extract. This thesis examines which ML techniques can be used to deal with highly imbalanced or noisy data.

Two main approaches are explored: detecting actionable alerts (alerts deemed useful by the developers) and alerts that aid in detecting bugs. These approaches are complementary because the sets of alerts are not necessarily equal, thus they are a good candidate for a combination.

The research questions can be formulated as follows:

- R0: Can we apply SA ranking techniques in an industrial environment with limited amount of data (abandoned because of high false positives)?
- R1: Given a highly imbalanced and noisy dataset, what are the best performing dataset balancing techniques?
- R2: Can we combine SA ranking techniques to achieve better results?

This research is important because it quantitatively examines if the version history of a project combined with machine learning techniques can be used to effectively improve the output of SA tools. By doing so, we can examine the real utility of this approach in practice and identify which techniques are more effective. It is also relevant to see if these approaches produce meaningful results in the case of limited amount of data. We can observe the impact in performance by testing different ML techniques to reduce noise and balance the dataset (like under and oversampling).

Based on the literature review, few papers make direct comparisons between different methods on a common experiment baseline (**TO DO: cite papers**). Also, they focus on open source Java systems with an adequate amount of data. One paper researches the impact of noisy data and proposes a solution (**TO DO: cite clni**). Regarding ensemble techniques, there have been approaches where multiple SA tools are combined, or where each alert types is handled by its own classifier. In contrast, we focus on C++ code and compare different preprocessing techniques. Also, we try an ensemble approach with two different methods of ranking SA alerts.

5 Ranking the output of Static Analysis

Given the company context, a single tool approach was chosen as to not introduce extra dependencies/complexity (in contrast to combining different SA tools).

The different techniques that were tested are:

- Using a Bayes Network for prioritizing alerts depending on location information.

- Predicting actionable alerts using ML algorithms based on different code/change metrics.
- Prioritizing alerts that point at bugs: analyze the code history to see which alerts pointed to fixed bugs and try to extract patterns from that.
- Detecting bug-prone methods and prioritizing alerts to those parts of code.
- Combining the three aforementioned methods, where each one focuses on a particular part of the dataset/alert types.

5.1 Collecting data

Clang AST ([2]) and Clang Tidy ([3]) was used to analyze the code. It was chosen is because it's a reliable open source tool, extensible and most importantly supported by the codebase of the company.

5.1.1 Clang AST

Since some of the algorithms need metric data from the code, the Clang AST was chosen to extract that information. Assuming that the project can be successfully compiled, Clang can provide an API on top of the parsed AST.

All information about the AST for a translation unit is bundled up in the class *ASTContext*, which allows traversal of the whole translation unit.

Clang's AST nodes are modeled on a class hierarchy that does not have a common ancestor and that consists of three main node types: Declarations, Statements (including Expressions) and Types (each with its own large inheritance hierarchy).

In order to traverse the AST, Clang offers two approaches:

- **RecursiveASTVisitor**: A recursive visitor based approach, which allows you to run custom actions based on the node that is visited.
- **AST Matchers**: An approach that allows you to define what nodes you want to match by using a domain specific language.

5.1.2 Clang Tidy

Clang Tidy is a modular C++ linter tool which provides an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis ([3]).

In addition to its Static Analyzer checks, Clang Tidy contains a large list of other checks ranging from those that target bugprone code constructs, to CERT Secure Coding Guidelines, C++ Core Guidelines etc...

Clang Tidy can be configured by selecting the type of checks to be executed or by restricting the parts of code where to carry out such checks.

5.1.3 Workflow for collecting information

By exploiting the extensibility of Clang Tidy (ability to define new checks) and the power of the Clang AST (collecting information from code), the metrics needed by the ML algorithms can themselves be implemented as checks and thus be extracted automatically when analyzing the codebase. This is a flexible approach which allows to build automatic processing of alerts by integrating alert as well as metric collection within a single toolchain.

In order to make processing alerts easier, the C++ interface code of Clang Tidy was slightly changed to output information in a more suited format (line number instead of bit offset from start of file), hide not useful information, and output alerts only from the file under analysis (not from imported headers).

Starting by the provided open source python scripts, a workflow can be built in python to automatically collect alerts and metrics. In order to crawl the code history of the project, different script were implemented in python to automatically guide the workflow and process the output of SVN (version control system used at the company).

The high level workflow for collecting data consists of the following steps:

- Start by selecting a base revision in the code history.
- Fully analyze that revision of the codebase (collect all alerts output by Clang Tidy).
- Repeat for desired number of iterations (revisions):

- Checkout next revision, detect changed parts of code, collect surrounding information (author, changed methods, etc...).
- Run Clang Tidy only on the changed parts of code.
- Collect any new alerts and metrics.

5.1.4 Assumptions made for collected data

Some algorithms use the notion of Actionable Alerts for processing the output of SA tools. Actionable Alerts (AA) are alerts that are deemed important by developers in the past (alerts that they acted on). In order to automatically label alerts as such (no existing information that can point to that), an important and risky assumption has to be made: alerts that disappear during code changes (revisions/commits) are considered as actionable, the rest is not.

This assumptions, though necessary, is risky because not all alerts disappear as a result of direct and targeted change by developers. Their disappearance can be caused by other unknown factors (those related to deleted files are not taken into account). Data generated using this approach can contain a lot of false positives and potentially damage the performance of the used ML algorithms.

Another algorithm uses alerts pointing at past bugs as a way to prioritize future alerts. Also in this case an assumption is needed: the alerts that pointed to past bugs, were directly related to that bug. This may not always be true and can cause the aforementioned problems as a result.

The nature of automatic data extraction, without a reliable oracle pointing at the right decisions, leads to impure data and penalizes the efficiency of ML algorithms, but unfortunately it is an indispensable trade-off to be made.

5.2 Data overview

TODO: Analyze of large portion of revisions and produce statistics TODO: List which alert types were considered TODO: Bug statistics for bug related lines

There are X packages, Y files, Z lines of code.
Analyzed X revisions, from Y date to Z date.
There are these alerts per type.
Open/closed alerts statistics.
Alert distribution per type/package (graphs).

5.3 Feedback Rank and Z-Score

Feedback Rank ([4]) combined with Z-Score ([5]) is a simple technique which ranks alerts on the probability of them being true/actionable. It consists of three basic features: package, file, function of the alert being analyzed (though may be extended as needed). By constructing a Bayesian Network based on these three features and the history of the project (which alerts have been proven valuable) it produces a probability ranking about which alerts are most useful.

As explained on the literature review (section 3), Feedback Rank is based on the assumption that bugs and false positives are clustered by code locality. The alerts are divided into two major regions, one that contains mostly true positive, and one that contains mostly false positives. A Bayesian Network (BN) is used to calculate the probabilities of an alert or cluster of alerts belonging to a certain region.

TODO: add graph of alert distribution for the project, similar to this paper

The initial configuration of the network can learned from historical data (extracted as explained on section 5.1.3). Feedback Rank is supposed to be an online ranking system, if we inspect a report and know its value, the probabilities of the parents are re-calculated.

To construct the BN, the Pomegranate library was used ([6]), trained with the extracted alert data from the version history.

5.3.1 Z-Score

To break ties when the probabilities provided by the BN are equal between alerts, the Z-Score metric is used based on the number of alerts on the same file. Z-Score is used in Z-Ranking ([5]), which makes use of the observation that the most reliable error reports are those that generated few failed checks and many successful checks, since the actual amount of bugs in code is relatively small.

The *z-test* statistic, which measures how far an observed value is from the real population, in this case produces a large positive *z-score* when there are few errors and many successes, and a large negative *z-score* when there are few successes and many errors.

To make use of the Z-Score, an approximation is made. The granularity used for calculating the scores of alerts is based on file level (how many actionable/unactionable alerts of a certain type in a file), instead of the original granularity of the alert (for example an alert that only works on *for loops*). This approximation is made because we do not know for each alert in which code construct it works on. Also, since it is only used as a tie-breaker, a high precision is not necessary.

5.4 Detecting Actionable Alerts via Machine Learning

Different research papers have focused on automatically classifying alerts in true/false positives or actionable/unactionable, by constructing classifiers based on code or change metrics ([7], [8]).

Instead of focusing on classifying alerts as true or false positives, we focus on Actionable Alerts instead: alerts that are deemed important by the developers (not restricted to the type of alerts, but also to the context on which it manifests itself). The later is a less restrictive definition and makes it easier to collect data. Classifying alerts as true or false would necessitate an oracle telling which is which or a large and representative dataset generated manually. In addition, from a developer's perspective, AAs can be more useful. An alert can be true but might be considered not important by developers and thus be equally useless as a false one (low criticality, no impact on user side etc...).

Research is conducted by following the example of [8], since it contains an agglomeration of alert characteristics (AC) collected from other research papers.

The workflow, as explained in section 5.1.3, consists of iterating through the version history, collecting alerts characteristics and keeping track which alerts disappear (considered as actionable). ACs are then later used as features in ML algorithms with actionability being the target to predict.

The scikit-learn library was used to perform ML experiments ([9]).

5.4.1 Alert Characteristics

The collected ACs can be classified in five main categories: alert information, source code metrics, version history, churn metrics, and aggregate characteristics.

Alert information: (a) package name (or folder), (b) file name, (c) file extension, (d) alert type, (e) alert category (security, core guidelines...), (f) method signature.

Source code metrics: (a) number of statements, (b) number of methods, (c) number of classes, (d) cyclomatic complexity.

Version history: (a) alert open revision, (b) developers who made changes from the open revision of an alert to revision under analysis, (c) file creation revision, (d) file deletion revision, (e) latest modification revision.

Churn metrics: (a) added lines, (b) deleted lines, (c) growth (added-deleted), (d) total modified (added+deleted), (e) percent modified.

Aggregate characteristics: (a) total alerts for revision, (b) total open alerts for revision, (c) alert lifetime, (d) file age, (e) alerts for artifact (method, file, package), (f) staleness (amount of time since last change of file, method, package).

5.4.2 Pre-processing data

***mostly categorical data -> tree based algorithms
*****do not use label encoding with nominal data (false order)**
*****try other encoding techniques**

Label encoding Some algorithms need data in numerical form. Since most of the features are categorical data, we need a way to convert them to numerical. A simple approach was chosen, label encoding, which assigns an integer to each value of a category. Other approaches such as One-Hot encoding would increase the amount of data a lot (ex. consider one-hot encoding the file where an alert a method is from, which is a feature with high cardinality).

Imbalanced dataset A quick look at the number of closed or not closed alerts shows that the dataset is heavily unbalanced. In order to train the ML algorithms we need first to balance the dataset. Two main techniques were tried: random undersampling and SMOTE (by using the library from [10]).

TO DO:
-cite paper where oversampling works better
-list other metrics
-explain balancing techniques?

Scaling While for tree based algorithms scaling is not needed, for other algorithms like Logistic Regression, scaling the inputs can make a difference.

Missing values In some ACs data may be missing, for example...

Feature selection The features needed to train the models vary from project to project. Since collecting features that have little to no impact on algorithm performance is a waste of resources, techniques can be applied to select the most representative subset of features. Two main techniques were used: PCA and Recursive Feature Elimination. **TODO: COMPARISON FULL FEATURES VS REDUCED**

5.5 Bug related lines

Another automatic way to determine which alerts are useful is to check if they pointed to lines that were changed during bug fixes. By doing so, we are regarding as valuable only those alerts that potentially signaled future bugs. The concept of bug related lines (BRL) is used in [11] and [12].

BRLs are calculated as follows:

- We start at a base revision and iterate backwards to a target revision.
- If revision under analysis is a bug-fixing revision (contains a bug ID) collect changed/deleted code lines from the version history.
- If those collected lines were present in the code at least since the target revision, we consider them bug related lines.
- Continue iterating backwards, collecting BRLs. If previous lines that were considered BRL were changed before reaching the target revision, we remove them from the set.

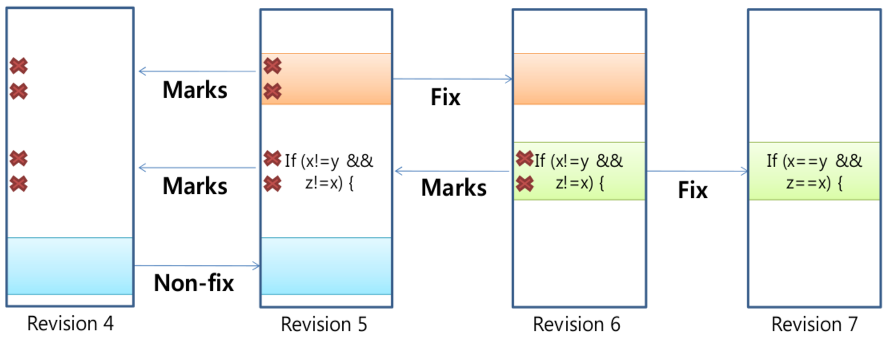

Figure 1: Calculating BRLs ([11])

Since the amount of collected lines and warnings can be rather limited, we extend the definition to Bug Related Methods (BRM). Namely, we trace in which methods the BRL belong to and also consider alerts inside those methods as valuable. That allows to extend the dataset, but also potentially weakens the data by introducing more noise.

5.6 Bug prediction

Alerts pointing at potential bugs are the ones that can be considered the most important. The cost of detecting and fixing a bug is much lower if detected early in the development cycle. If we can predict which components will be likely to cause failures in the future, we can prioritize alerts that point to those components.

The appropriate granularity to use for bug prediction is at method level. File level granularity is too broad since a lot of files contain many lines of code, which would result in prioritizing a lot of alerts, while anything lower than method level would be too hard to predict as bug prone.

We follow the example of the [13] method. By using a combination of source code and change metrics and by exploiting the version history of the codebase, classifiers can be built that predict bug prone methods.

TO DO: list used features

5.7 Combining the techniques

5.7.1 How to combine the strengths?

apply each technique to a subset of the warnings (which is more appropriate)

5.7.2 Better results?

Does it provide better results?

5.8 Evaluation methods

TODO: EVALUATE OVERSAMPLING METHODS

To evaluate and compare the approaches different techniques are used:

- K-Fold Validation
- Release based testing
- Alert ratio
- Average false positives to inspect ([5]).

K-Fold validation is often used in papers containing ML approaches to ranking alerts, though that may not necessarily be the best choice (add why). In K-Fold validation, the data is randomly divided into folds and one of those is used for testing while the rest for training. Release based testing consists in training the algorithms with data of a previous release and testing on the next one. Instead of releases it can be applied by splitting the data into two sets based on time. This is better suited to the domain than K-Fold since alerts are not totally independent from each other in time. Other ways to evaluate the results of ranking approaches are to compare the new output with a random order of alerts and check how many more alerts are better placed in that order. Another approach is to average the number of times an unactionable alert has to be inspected before reaching an actionable one (and compare it again to random order).

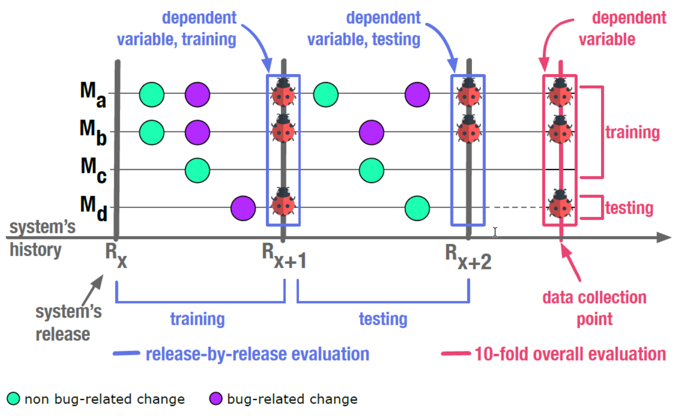

Figure 2: Release-based vs K-Fold ([14])

5.8.1 Preparing the dataset

Since the amount of actionable warnings is very low compared to the total amount of warnings, the dataset is balanced in two steps. First a subset of the packages and warnings is selected. By considering the ratio of actionable to the total amount of alerts, we can select the packages or types of warnings with the higher ratio (ex. the first 100).

After the first step, a test set is selected for evaluating the algorithms (equal to 20% of the dataset). That is especially needed to test the algorithms after the K-Fold evaluation.

The second step consists of balancing the amount of actionable and not actionable alerts, either by random subsampling or by artificially creating new samples (via SMOTE).

5.9 Results of individual tools

5.9.1 Random test set

5.9.2 Release based testing

5.9.3 Alert ratio

5.9.4 Average false positives to inspect

6 Conclusions

TO DO: FEASIBILITY STUDY BECAUSE OF LIMITED AMOUNT OF DATA

6.1 Conclusions

-got these results, because...
-importance on preprocessing
-initial results may be slightly better that normal, but the nature of data collection is limiting, need to be used in continuance

6.2 Summary of Contributions

This thesis provides the following contributions:

- Building a workflow to extract information by making use of the Clang toolset and version history (SVN).
- Evaluating SA ranking techniques on an industrial codebase.
- Evaluating preprocessing techniques to deal with highly imbalanced and noisy data.
- Detailed report of ML workflow (not a lot of information on most papers)
- Comparing different approaches on a common codebase.
- Exploring the utility of combining different methods.

6.3 Future Research

Future research can be focused on different aspects, the most important being reliable data collection. A classifier is as good as the data it was trained on, so new ways to collect actionable alerts in a more precise way are crucial to achieving better performance. Information Retrieval or Natural Language Processing techniques can be applied for example on bug messages/descriptions to have a clearer connection between a bug and an alert (did the alert really predict the bug?).

Given also the limited amount of data, new or improved approaches that can generalize easily are needed. In that regard, research can focus on the type of features extracted from the code or version history that are discriminative enough to make correct classification even with limited data. FEATURE ENGINEERING, high cardinality, categorical data

Furthermore, given the diverse nature of alerts current tools have, from finding bugprone construct, stylistic alerts, library-oriented alerts, to security or performace-oriented checks, different methods can be tailored that maximise performance within these subsets of alerts. For example, a simple method like Z-Ranking ([5]) can be more suited to predict stylistic alerts than others.

In addition, a way to continuously improve the ML algorithms needs to be put in place. Even though the initial performance may not be spectacular, if new resolved alerts are tracked consistently, performance will also rise accordingly. A rather naive implementation is to give warnings an identifier and include them in the commit messages if they were useful.

7 Background information

Possible concepts to explain/introduce in the thesis background section:

- Bayesian networks
- ML algorithms
- Precision, recall, auc

References

[1] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61:58–66, 03 2018.

[2] Clang AST.

[3] Clang Tidy.

[4] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. *ACM SIGSOFT Software Engineering Notes*, 29, 09 2004.

[5] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. volume 2694, pages 295–315, 06 2003.

[6] Jacob Schreiber. Pomegranate: fast and flexible probabilistic modeling in python. *Journal of Machine Learning Research*, 18(164):1–6, 2018.

[7] Joe Ruthruff, John Penix, J. Morgenthaler, S. Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. pages 341–350, 01 2008.

[8] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. pages 161–170, 04 2009.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[11] Sunghun Kim and Michael Ernst. Which warnings should i fix first? pages 45–54, 01 2007.

[12] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. pages 93–102, 01 2010.

[13] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald Gall. Method-level bug prediction. pages 171–180, 09 2012.

[14] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software*, 161:110493, 2020.

[15] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. volume 2006, pages 452–461, 01 2006.

[16] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. 08 2006.

[17] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. pages 297–308, 05 2016.

[18] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. Prioritizing alerts from multiple static analysis tools, using classification models. pages 13–20, 05 2018.

[19] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. pages 1–10, 08 2019.

[20] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53:363–387, 04 2011.

[21] T. Muske and A. Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, Oct 2016.

[22] Moritz Beller, Radjino Bholanath, Shane Mcintosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. 03 2016.

[23] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. 08 2019.

[24] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. pages 317–328, 09 2018.

[25] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 152–161, New York, NY, USA, 2014. ACM.

[26] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. pages 17–26, 08 2015.

[27] Radhika Venkatasubramanyam and Shrinath Gupta. An automated approach to detect violations with high confidence in incremental code using a learning system. *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*, 05 2014.