RANKING THE OUTPUT OF STATIC ANALYSIS

Kleidi Ismailaj

kleidi.ismailaj@student.uantwerpen.be

August 17, 2020

Contents

List of Figures

List of Tables

Abstract

Previous research has estimated the number of false positive static analysis alerts to be as high as 70%. 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.

1 Introduction

Software development and maintenance is a complex activity [3]. Because of this complexity, there are many challenges in the software engineering landscape, when developing software. In this environment, automated tools and techniques matter to help the development scale with consistency [4].

Static Analysis (SA) is an automated technique that can be useful for the development process. For example, it is possible to use SA to detect potential bugs in the source code. It is a generally accepted principle that resolving issues in earlier stages of the development cycle is less costly [5]. Therefore, the bugs detected by SA may reduce the maintenance and development costs.

Popular companies acknowledge the usefulness of Static Analysis techniques by adopting them in their process. For instance, Google with its Tricorder architecture [6] or Facebook with its Infer static analyzer [7].

Even though SA is useful and employed by real companies, there is still much room for improvement. Since the software under analysis is not executed, SA tools must infer what the actual program behavior will be. Therefore, SA tools are bound to make misclassifications or raise false alarms.

Given the tendency of SA tools to over-estimate possible faulty program behaviours, there is a need to fine-tune and improve the alarms and warnings raised by these tools. There are two ways that we can improve SA tools: (i) increase the precision of the analysis which usually decreases its recall and the overall number of raised alarms; or (ii) by post-processing the alarms after they are generated, according to specific criteria more appropriate for the codebase. We opted to focus on the second option, post-processing of alarms. The classic approach most tools use for prioritizing and filtering results is to classify the results based on severity levels. This approach is simple but oblivious to the actual analyzed code or the location and frequency of a given issue. Furthermore, it has been shown that if developers lose trust in the tool, they then tend to ignore the output of it altogether [6].

Different approaches have been proposed to improve the output of SA tools. Optimally, the initial warnings report should be those most likely to be real errors. Alerts can be strategically prioritized for examination, by tracking and analyzing them through a series of software versions, we can automatically determine which SA rules are more important and which parts of the software are more problematic. Moreover, an understanding of how developers react to these alerts can help improve the usability of these tools. Alerts can be divided into two categories: (i) actionable alerts (AA) which the programmer would act on to resolve; and (ii) unactionable alerts (UA) which the programmer would not act on [8, 9]. An unactionable alert may be of trivial concern to fix, less likely to manifest at runtime, or incorrectly identified due to the limitations of the tool. Thus, we want to prioritize the actionable alerts and hide from developers the unactionable ones.

Another approach is to prioritize alerts that, in the past, lead to the discovery of bugs [1, 10]. By tracking back the bugs up to a past version, we can collect code lines that changed during bug fixes. Subsequently, we can pinpoint which alerts warned about those specific parts of code and prioritize accordingly.

For a relatively large project, the number of SA alerts can be prohibitive, which is one of the reasons developers avoid these tools [11]. The ultimate goal is not to analyze all alerts, but to maximize the time-cost spent on them. 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 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. Among those few, the comparison is mainly done between open-source Java software where a good amount of data can be extracted. In our case, we assess the utility of different approaches inside an industrial C++ codebase, where static analysis has been abandoned due to poor performance. Also, it is interesting to explore if these methods can be combined to achieve better results. Different techniques can be better suited to different types of alerts or can compensate for each others weaknesses if combined.

In this thesis, our goal is to verify if it is possible to apply ranking approaches to static analysis alerts in an industrial code base with a limited amount of noisy data. We like to highlight that SA tools were initially employed and later abandoned by this company due to a large number of false positives. We claim that a ranking mechanism fined tuned to this company's code base can achieve good results and be useful to developers.

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

2 Problem statement

Improving the warnings provided by static analysis techniques dependents on a particular codebase and the people who produce it. Organizations can have different priorities and expectations on the static analysis alert report. Also, developers might be more interested in a particular subset of alerts, or the context on which the alerts appear.

OMP regards static analysis as a useful approach in improving software quality and preventing bugs early in the development cycle. Though, the usage of such tools was stopped because of the high amount of false positives.

Starting from the original alert set produced by a SA tool, automatic processing techniques can be applied to rank the alerts in such a way that the utility of inspecting the ranked alerts is higher than the default order (there are more actionable alerts in the post-processed output). 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 also examines which ML techniques can be used to deal with 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:

- RQ#1: Can we apply SA ranking techniques in an industrial environment with limited amount of data (abandoned because of high false positives)?
- RQ#2: Can we combine SA ranking techniques to achieve better results?
- RQ#3: Do pre-processing techniques provide a significant performance benefit?

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 also observe the impact in performance by testing different ML techniques to reduce noise and balance the dataset (under and oversampling).

Based on the literature review, few papers make direct comparisons between different methods on a common experiment baseline [8, 12]. Also, they mainly focus on open source Java systems with an adequate amount of data. Kim et al. [13] research the impact of noisy data and propose a solution . 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.

3 Literature review

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

- *Ranking static analysis alerts*: some of the most known (cited) approaches to rank alerts.
  - Using a single SA tool (section 3.1.1).
  - Combining multiple tools (section 3.1.2).
  - Looking at survey papers that describe the state of the research and state of the art techniques (section 3.1.3).
  - Comparative studies evaluating different methods (section 3.1.4).
- *Bug prediction*: Rank the alerts in problematic parts of code higher (section 3.2).
- Information on *real world usage of SA tools*, problems and suggested solutions (section 3.3)

3.1 Dealing with False Positives

3.1.1 Single Tool

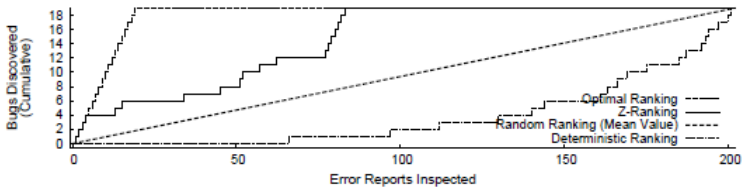
Kremenek and Engler [14] introduce *Z-ranking*, a statistical model to rank the error reports of SA tools. They make a distinction between successful and failed checks (those that satisfy a checked property and those that violate it). The underlying observation is that the most reliable error reports are those that generated few failed checks and many successful checks, since the actual amount of bugs in code is relatively small. An explosion of failed checks is a likely indicator that something is going wrong with the analysis. Reports are sorted based on the calculated *z-test* statistic (based on the relative frequency of successful and failed checks).

The problem can be formally defined as a classification task. Let P be the population of all reports, both successful checks and failed checks, emitted by a program checker analysis tool. *P* consists of two subpopulations: *S*, the subpopulation of successful checks and *E*, the subpopulation of failed checks (or error reports). The set of error reports *E* can be further broken down into two subpopulations: *B*, the population of true errors or bugs and *F*, the population of false positives. The classification problem can then be restated as follows: given an error report $x \in E$, decide which of the two populations *B* and *F* it belongs to. That is based on the fact that *B* and *F* have different statistical characteristics.

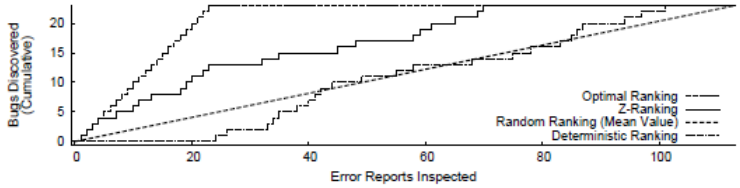
Given a grouping operator *G* that groups successful and failed checks together, we calculate the proportion of failed checks $G.\rho = \frac{G.successful}{G.failed}$. Populations are ranked both by the ρ value and by the degree of confidence in its estimation. By treating these checks inside the groups as a sequence of binary trials (coin tosses). The probability p_i of success will have to be approximated using the standard error. By using the *z-test* statistic, which measures how far an observed value is from the real population, a value can be specified that 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.

Given an estimated p_i and a calculated *SE*, we can chose p_0 to produce the effect mentioned above: $z = \frac{observed-expected}{SE} = \frac{p_i-p_0}{SE} = \frac{p_i-p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}}$. The average population success rate can be chosen as a starting point for the value of p_0.

According to their tests, *Z-ranking* performed better than randomized ranking 98.5% of the time. Moreover, within the first 10% of reports inspected, *Z-ranking* found 3-7 times more real bugs on average than found by randomized ranking.



(a) Intra-procedural check on Linux spin (lock)



(b) Inter-procedural check on Linux (free calls)

Figure 1: Z-Ranking evaluation

Kremenek et al. [15] introduce *Feedback-Rank*, a dynamic ranking scheme that adapts as reports are inspected. By analyzing historical data, they observed that both bugs and false positives cluster by code locality. They present a probabilistic technique that exploits this correlation and also incorporates user feedback by reordering reports after each inspection. Since reports are correlated within a population (cluster), inspecting one of them yields information about the others. The ranking works by using a *Bayesian Network* and exploiting two features, the number of populations (error messages grouped together) and the strength of correlation in each population. Furthermore, the strategy also continues improving with time, by taking into account the history of inspections.

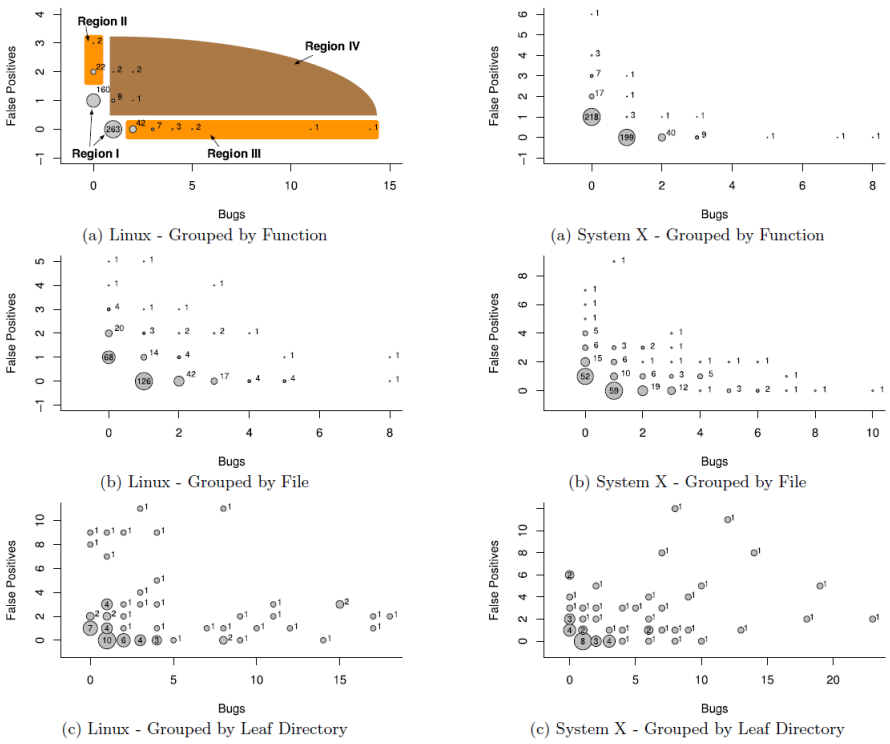
An intuitive explanation for the reason that reports cluster is given for both real and false positives. Regarding true positives, when developers do not know a rule, they will repeatedly violate it, so errors of the same type will correlate together. As for the false positives, there are three main causes: analysis mistakes of the tool (explosion of errors), rare coding idioms used by developers (which trigger the tools), incomplete rule specifications (a rule holds in most cases, but can be safely violated in others).

To cluster the reports, code locality in different granularities is chosen: function, file and directory level. From the results in fig. 2a it can be seen that very few populations contain a mix of bugs and false positives. *Applicability* is defined as the ratio of non singleton clusters (which are bad for online ranking) to the total amount of clusters. The coarser the granularity the greater the applicability but also the smaller the correlation. *Skew* is defined as the ratio of homogeneous clusters (all bugs or all false positives) to the total number of clusters. In this case, the more refined the granularity (function level), the higher the skew. Thus, a trade-off needs to be made between applicability and skew.

To apply the algorithm a model is needed that produces the correlations among the reports. The reports are divided into two major regions, one that contains mostly true positive (g_B), and one that contains mostly false positives (g_{FP}) (see fig. 2b). A *Bayesian Network* is used to calculate the probabilities of a cluster belonging to a certain region (regions are different for different granularities). The initial configuration can either be chosen by the user or learned from historical data. A simple model can be seen on **??**, where 3 reports depend on the probabilities of the parent function, file and directory clusters they belong to. Influence though, flows across both directions: if we inspect a report and know its value, the probabilities of the parents are re-calculated. Gives a training set, the conditional probability distributions of the network (along with the probabilities for the regions) can be learned using *Expectation Maximization*. *Belief Propagation* is used to update probabilities after each inspection and *Information Gain* is used as a secondary factor to rank the reports.

Feedback-Rank represents a complementary approach to static ranking schemes (it can be combined with Z-Ranking for example) and can be trained with other forms of correlation instead of code locality.

According to their tests, *Feedback-Rank* performed 2-8 times better than randomized ranking.



(a) Linux - Grouped by Function

(a) System X - Grouped by Function

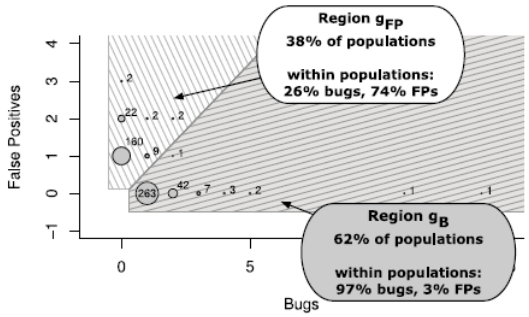
(b) Linux - Grouped by File

(b) System X - Grouped by File

(c) Linux - Grouped by Leaf Directory

(c) System X - Grouped by Leaf Directory

(a) "L" shape of clustering



(b) Populations divided into regions, mostly true or false positives

Figure 2: Feedback-Rank clustering of alerts

Boogerd and Moonen [16] present a technique, *ELAN*, that prioritizes SA warnings by using the (predicted) likelihood that the execution reaches the location for which the warnings are reported. The execution likelihood is defined as the probability that a program point will be executed at least one in an arbitrary program run and is calculated statically. This computation is demand-driven, thus it is only performed for the locations associated with warning reports.

The workflow (fig. 3) consists of normalizing the results of SA tools (to a specific format), creating system dependency graphs, calculating for every warning the likelihood of execution, ordering the results using the execution probability and possibly other external techniques (like Z-Ranking).

Likelihood analysis is based on system dependency graphs, which tie all program dependency graphs (function level) together by modelling the inter-procedural control dependencies. Their approach only considers control flow and ignore dataflow information. In order to avoid traversing all the SDG, *program slicing* is used on control points. Other than the basic algorithm, they also introduce branch prediction heuristics, which do not excessively impact performance.

Experiments show that predicted execution likelihoods correlate with data extracted from dynamic profiling. One problem though, is that when for example 30% of all code is always executed, then the ranking of those warnings that belong to that piece of code, cannot be distinguished.

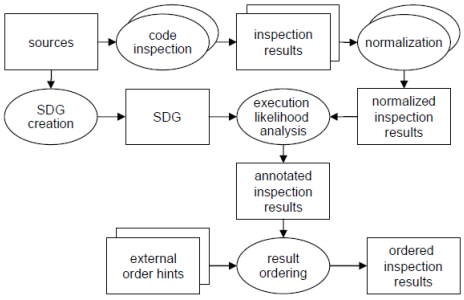


Figure 3: Workflow for the ELAN tool

Kim and Ernst [1] propose a history-based warning prioritization (HWP) algorithm which works by mining fix-changes in the VCS. It is based in the intuition that if a warning is removed by a fix, then probably that warning was important. On the other hand, if a warning instance is not removed for a long time, then warnings of that category may be neglectable, since the problem was not noticed or was not considered worth fixing. They measured the tool warning prioritization (TWP) on three different systems and found a precision of 3%, 12% and 8%.

They set a weight to each warning category to represents its importance. The weight will be proportional to the number of warnings eliminated by changes (where fix-changes have the biggest weight, fig. 4a). Selecting the top weighted warnings improves precision up to 17%, 25%, and 67% respectively. Precision is calculated as $precision = \frac{number\ of\ warnings\ on\ bug\ related\ lines}{total\ number\ of\ warnings}$. By looking at the fix-changes and corresponding affected lines, by starting at the last revision, they can mark the bug-related lines, up to the first revision when they appeared (fig. 4b). Ranking is category-based, so only the categories of warnings are considered and there is no distinction between the warnings inside each category. The algorithm works well if the categories are fine grained and internally homogeneous.

They measure precision by training the weights in the first half of the version history, and testing them on the other half. The HWP outperforms TWP for all three systems (fig. 4c).

// initialize weight w_c

$w_c = 0$

for each warning instance *i* in category *C*

// fix-change promotion step

if *i* is removed in a fix change

then $w_c = w_c + \alpha$

// non-fix change promotion step

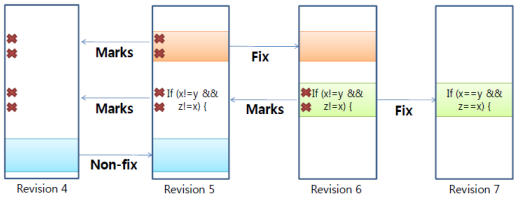
if *i* is removed in a non-fix change

then $w_c = w_c + \beta$

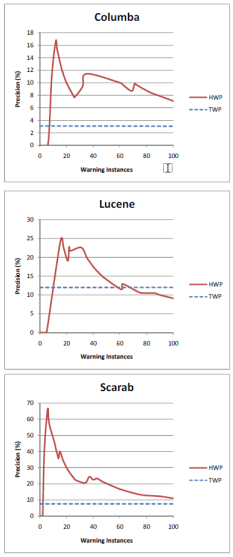
// weight normalization step

$w_c = w_c / |C|$ where |C| is the number of warning instances in category *C*

(a) Precision results at line-level



(b) Line marking approach



(c) Line marking approach

Figure 4: HWP algorithm and results

Ruthruff et al. [9] use *logistic regression* models to not only reduce the number of false positives in the output of SA tools, but also to predict actionable warnings. Warnings are not always acted on by developers even if they reveal true defects. The reason may be that the defects may have little impact and require significant effort for little perceived benefit. Furthermore, they introduce a statistical methodology for discarding features with low predictive power and thus avoiding the capture of expensive data. Information to build the models is mainly drawn from: (a) light-weight code complexity metrics for post-release bug prediction, (b) file (history) information to predict fault counts within individual files. The features include the history of warnings, source code characteristics, churn factors, and warnings descriptors (fig. 5a).

Their screening methodology, for selecting an independent subset of predictor features, consists of up to four stages, and attempts to identify at least six predictive features. The stages respectively consider 5%, 25%, 50%, 100% of the warnings, continuously removing features with low predictive power. One of the reasons to consider this cost-effective approach is that it may be desirable to rebuild the models at different points in time, either because a significant number of new warnings have been reported, or the codebase has undergone substantial change.

By considering a sample of around 1600 warnings (inspected by two engineers), and by using different models (with resulting different features) for classifying true positives and actionable warnings, they achieved an accuracy of 85% for the former, and 70% for the later (fig. 5b).

Factor	Description
FindBugs warning descriptors	
Pattern	Bug pattern of warning
Category	Category of warning
Priority	FindBugs warning priority
Google warning descriptors	
BugRank	Google metric of warning's priority
BugRank Range	Category (range) of warning's BugRank
File characteristics	
File age	Number of days that file has existed
File extension	Extension of Java file
History of warnings in code	
File warnings	Number of warnings reported for file
File staleness	Days since warning report for file
Package staleness	Days since warning report for package
Project warnings	Number of warnings reported for project
Project staleness	Days since warning report for project
Source code factors	
Depth	How far down (%) in file is warning
File length	Number of lines of code in file
Indentation	Spaces indenting warned line
Churn factors: files, packages, and projects (6 × 3 factors)	
Added	Number of lines added
Changed	Number of lines changed
Deleted	Number of lines deleted
Growth	Number of lines of growth
Total	Total number of lines changed
Percentage	Percentage of lines changed

(a) Some of the features considered for building the models

Model Type	Resubstitution	Holdout Data		
		70/30	*80/20*	*90/10*
Screening	85.29%	87.48%	87.09%	86.54%
All-Data	85.71%	83.48%	84.74%	85.47%
BOW	76.51%	77.96%	78.73%	79.32%
BOW+	84.62%	82.24%	83.81%	83.06%

Table 7: Predicting false positive warnings. Holdout data shows the average precision of the models from the three observations.

Model Type	Resubstitution	Holdout Data		
		70/30	*80/20*	*90/10*
True Defects				
Screening	77.32%	71.82%	71.68%	71.95%
All-Data	71.37%	71.42%	69.95%	70.97%
BOW	60.19%	61.30%	63.47%	62.74%
BOW+	70.90%	67.04%	67.41%	69.79%
All Warnings				
Screening	77.42%	72.02%	71.36%	71.94%
All-Data	73.73%	72.90%	75.26%	75.68%
BOW	62.23%	59.35%	60.77%	61.12%
BOW+	73.91%	67.76%	69.50%	69.26%

Table 8: Predicting actionable warnings.

(b) Results for predicting true positives and actionable warnings

Figure 5: Actionable Alerts with Logistic Regression

Heckman and Williams [17] present a generic approach for building actionable alert machine learning models and provide a comparative study of different algorithms tested of two Java systems. Their initial feature set consists of 51 alert characteristics originating from alert type and history, software metric, software history and source code churn.

To collect data, they check out and build the program for each chosen revision (in practice they did that once in every 25 revisions) and collect alerts and their characteristics. Starting from the first revision, the sets of alerts between two revisions are compared, collecting information when alerts are opened and closed (and thus classifying them as actionable or unactionable).

By trying different feature reduction strategies and different machine learning algorithms, they provide results for two systems. The number of selected alert characteristics ranged from 3/4 to 13/14 and both projects had 5 distinctive sets. That shows that the set of AC needs to be tailored for each project. The average metrics for all models (fig. 6) show very good results. The difference between selected ACs and the best models between projects suggests that false positive mitigation models should be project-specific.

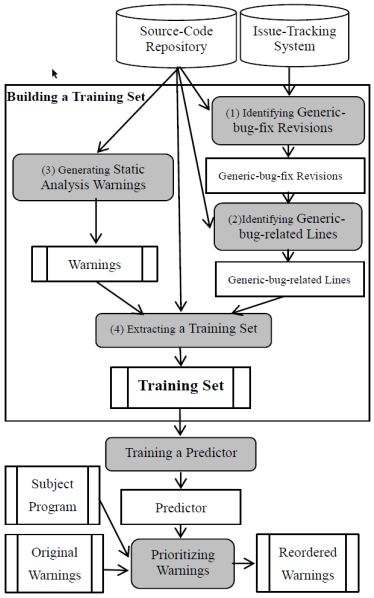
Project	Average Precision	Average Recall	Average Accuracy
jdom	89.0%	83.0%	87.8%
runtime	98.0%	99.0%	96.8%

Figure 6: Average metrics for all models

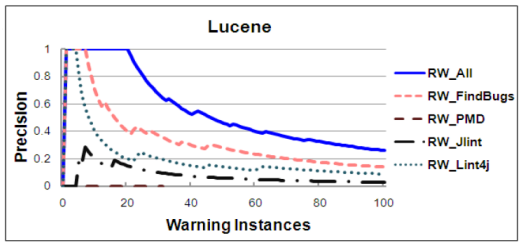
Liang et al. [10] propose an automatic approach (fig. 7a) for constructing an effective training set for warning prioritization algorithms. They introduce the notion of "generic-bug-fix revisions" vs. "project-specific-bug revisions", which differentiate bug-fixing lines depending on the sort of bug that they deal with. SA tools are designed to catch generic bugs that are applicable to all projects, while most of the bugs are domain (project) specific. By restricting the training set to only those set of bugs that can be caught by the tools, models can be trained better and a higher accuracy can be reached.

To identify generic-bug-fix revisions, they first limit the revision size to an empirically derived value (max 4 files changed). Then, they analyze the revision messages and using a natural language processing approach compare them against generic bug descriptions (by SA tools). If the similarity of these messages is above a certain threshold and the number of changed files is under the predefined limit, the revision is marked as a generic-bug-fix. To identify the generic-bug-related lines of a specific revision X, they start by analyzing all older revision than X, and backwardly calculate lines that were already present at X, when they were later changed by a generic-bug-fix revision.

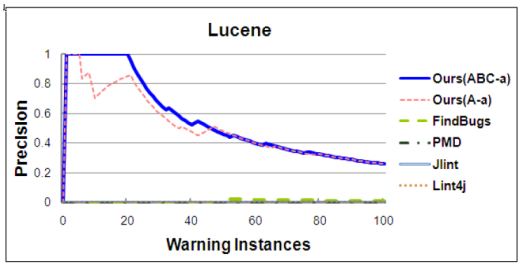
Using a *K-Nearest Neighbor* classifier (trained on the first half of the revisions) paired with a feature selection algorithm, they achieve significantly better result than the tool output, especially in the first 20 warnings range. They found that using multiple SA tools gives a better result than single tool models (fig. 7b). The type training set has also an effect in the final results, models trained only with the project under analysis performed worse that models trained with extra projects (fig. 7c). That adding inter project data (at least in the case of open source projects) has a positive effect in the model predictions can be explained with the choice of focusing only on generic-bug-fix revisions.



(a) Workflow for constructing a training set and a model



(b) Multiple vs. single tool results using training set



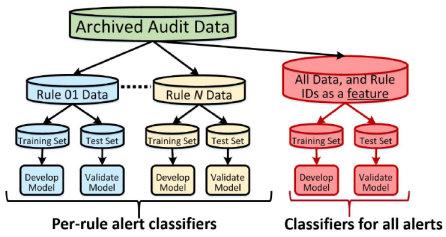
(c) Intra (A-a) vs inter (ABC-a) project training

Figure 7: Automatic construction of an effective training set

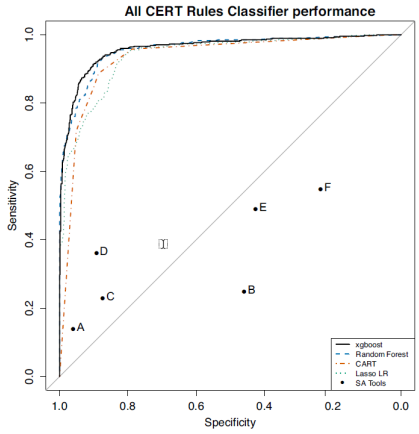
3.1.2 Multiple Tools

Flynn et al. [18] use *Alert Fusion* (unifying alert information from different tools) and different classifiers to classify alerts as expected true positive (e-TP), expected true negative (e-TN) and indeterminate (I). The e-TP alerts are separately prioritized for code repair and the I-alerts are automatically ranked based on classifier confidence and a cost metric to fix the code flaw.

The authors used a total of 354 manually audited SA alerts, which there then mapped to standardized coding rule violations (CERT). Different types of classifiers were used, using different portions of data: trained to detect a single rule violation, trained for a single programming language, and all rule classifiers. The results vary from around 80 to 90% accuracy, depending on the classifier type. The reliability of some of the results is doubtful since one of the major problems of the study was a lack of data.



(a) Workflow of building classifiers



(b) Results of all-rules classifiers

Figure 8: Prioritizing alerts from multiple SA

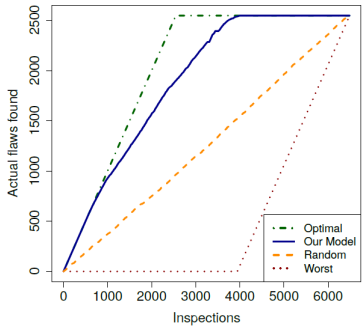
Ribeiro et al. [19] aim to reduce the false positive rate of an ensemble of static analyzers by using *Decision Trees* and *AdaBoost*. The goal is to make possible to combine the strengths of different analyzers without suffering too much from false positives. Their approach ignores source code characteristics, making it possible to be applied without any pre-processing step on the codebase.

They use *Juliet*, a synthetic C/C++ test suite which contains specific flaws with links to program code, to train and test the classifiers (see fig. 9a for the results of different tools on the set of selected test cases). The features to train the model include the tool name, number of warnings per file, warning category, number of neighboring warnings, number of warnings per file, and a boolean feature for each of the static analyzers.

By combining weak decision tree classifiers with AdaBoost, they can reach a mean acccuracy of 80% with a hundred trees, with precision and recall around 68% and 96% respectively. The ranking is done by sorting the warnings according to the probability assigned by the model, achieving a five time improvement over random ordering. The most important features in the classifier were the number of warnings per file and the tool name.

Tool	Warnings	TP	FP	FP Rate	Precision
Clang Analyzer	6207	984	5223	0.84	0.16
Cppcheck	4035	314	3721	0.92	0.08
Frama-C	15717	8892	6825	0.43	0.57
Aggregated tools	25959	10190	15769	**0.61**	**0.39**

(a) Labeled warnings per tool (from the extracted list of Juliet)



(b) Results of the classifier

Figure 9: Ranking warnings from mulitple SA via ensemble learning

3.1.3 Literature review papers

Heckman and Williams [20] perform a systematic review of *Actionable Alert Identification Techniques* (AAIT). The goal is to make an informed decision which AAIT to pair to an SA tool, in order to present relevant warnings to the tool

users. An actionable alert is defined as an important, fixable anomaly. Different studies have estimated the amount of unactionable alerts ranging from 35% to 91%. The authors divide the tools into different categories, based on input type, approach used, and evaluation method.

The categories of artifacts used by AAIT's are divided into five main categories: (a) alert characteristics (type, location), (b) code characteristics (metrics), (c) source code repository metrics (code churn), (d) bud database metrics, (e) dynamic analysis metrics (extracted during code execution). Most AAIT's combine more than one of these input categories.

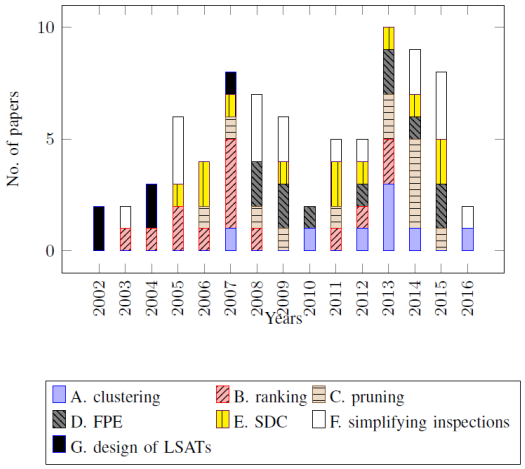
Approaches followed by AAIT's fall into seven main categories: (a) alert type selection (selecting altert types that are the most relevant for a codebase), (b) contextual information (limiting SA tools only to parts of code where that they can analyze well), (c) data fusion (combining multiple SA tools), (d) graph theory (system dependency graphs or repository history of changes), (e) machine learning, (f) mathematical and statistical models, (g) test case failures (generate test cases that demonstrate faults in the warning location).

Evaluation methodologies are divided into six categories: (a) baseline comparison (use a standard baseline), (b) benchmarks, (c) comparison to other AAIT's, (d) random and optimal order comparison, (e) train and test, (f) other. Classification AAIT's are evaluated using typical metrics as precision, recall, accuracy, and false positive rate, while Prioritization AAIT's are evaluated using correlation coefficients, statistical tests (chi-square), improvements over random, AUC etc..

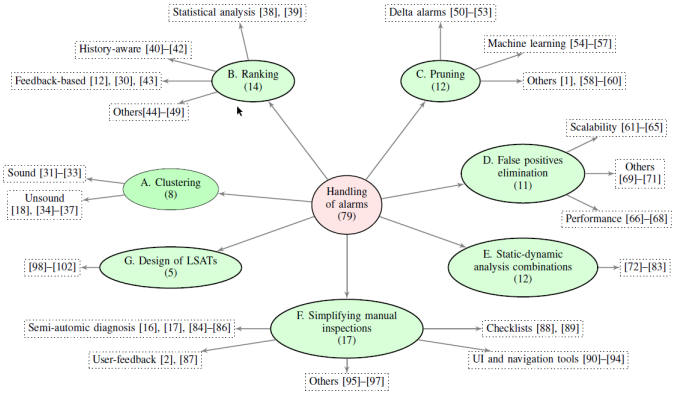
Muske and Serebrenik [21] perform a systematic review of SA alarm handling techniques (fig. 10a). They define *handling of alarms* as: (a) post-processing to reducing the manual inspection effort (using correlation, clustering, ranking...), and (b) supporting manual inspection of alarms.

Seven categories for identifying alarms are defined (fig. 10b): (a) clustering, (b) ranking, (c) pruning, (d) false positive elimination, (e) combination with dynamic analysis, (f) simplifying inspections, (g) design of light-weight SA tools (*LSATs*).

In *clustering*, alarms are partitioned into several groups based on similarity/correlation. There are two sub-categories: sound clustering, where there is a guarantee of certain dependencies among clustered alarms, and unsound clustering, where there are no guarantees on dependencies/relationships. In *ranking*, alarms are prioritized and those more likely to be errors are output at the top of the list. Different techniques can be used to support ranking, such as statistical models, history of alarm fixes, user feedback etc... In *pruning*, alarms are classified as actionable or non-actionable. Machine learning techniques can be used to classify the alarms (using patterns from surrounding code and syntactic/semantic differences), or alarm delta identification can be used identify the alarms that are newly generated (useful for legacy code). In *false positive elimination*, more precise techniques like model checking and symbolic execution are used to eliminate false positives. This approach is more precise and automatic but faces the issues of non scalability and poor performance. In *combining dynamic and static analysis*, SA alarms are checked if they are true errors. SA has been combined with test-case generation or slicing to find errors or extract more precise information. In *simplifying manual inspection*, approaches are used to help the user in alarm inspection by making inspections more automatic/systematic. Different techniques are used, from rule and checklist based approaches, to improved visualisation, to automatically deriving possible alarm causes. In *designing LSATs*, light-weight, scalable and shallow analysis tools are built to avoid generation of a large number of alarms. However there are no guarantees that all defects of a type will be uncovered.



(a) Number of relevant papers per year and category



(b) Summary of the approaches

Figure 10: Survey of approaches for handling SA alerts

3.1.4 Comparative studies

Heckman and Williams [8] perform a comparative study of six alert ranking techniques on the *Faultbench* dataset: (a) Actionable Prioritization models that are based on the assumption that alerts sharing a type/location are like to be all actionable or non-actionable, (b) Alert Type Lifetime models that prioritize alert types by their average lifetime (important alerts are fixed quickly), (c) Check 'n Crash that automatically generates unit test cases and checks if the test fails (alert is then considered actionable), (d) History-Based Warning Prioritization models that uses commit messages and code changes in the source code repository to prioritize alert types, (e) Logistic Regression models that are trained on thirty-three alert characteristics and predict the probability of an alert being actionable, (f) Systematic Actionable Alert Identification that collects a number of alert characteristics and tries to find the best subset of these characteristics and the best machine learning models that optimizes accuracy and precision.

On each of the three test projects of the benchmark, there is a different winner (based on accuracy), with Systematic Actionable Alert Identification and Logistic Regression models that generally perform better. There is also a trend where precision and recall decrease with the amount of analyzed revisions (70, 80 or 90%). That can be explained by the fact that the balance between actionable and unactionable alerts is heavily shifted to the later. This trend can also be explained by the fact that these techniques can be better at identifying unactionable alerts than actionable alerts.

Allier et al. [12] perform a comparison of different ranking algorithms based on their effort metric: average number of alerts to inspect to find an actionable one. They also focus on two other research questions, whether its better to rank alerts individually or alert types, and if there is a performance difference between statistical ranking methods and ad-hoc ones. They test six raking approaches (six Java and Smalltalk systems): Aware, FeedbackRank and Z-Ranking which mainly use alert type and location, RPM that uses logistic regression with thirty-three alert characteristics, AlertLifeTime that prioritizes alerts on type and lifetime, and EFindBugs which prioritizes alert types based on their defect likelihood.

They found out that Aware and FeedbackRank perform significantly better than the other ranking approaches. In addition individual alert raking algorithms performed better than those that rank alert types. Also they did not find a clear distinction in performance between statistical and ad-hoc approaches.

3.2 Bug Prediction

Nagappan et al. [22] use code complexity metrics to predict the likelihood of *post-release* defects for new entities (failures that occurred in the field six months after the release). Although, according to their findings, these metrics are correlated to failure prone entities, there is no universal set of metrics that produces the best results. As a consequence, principal component analysis is used to choose the optimal set of features for a particular project. Information from bug databases and historical data is used to select the appropriate metrics.

By analyzing a set of five large scale projects, they discovered the following results: (a) for each project a set of metrics can be found that correlates with post-release defects, (b) there is no single et of metrics that fits all projects, (c)

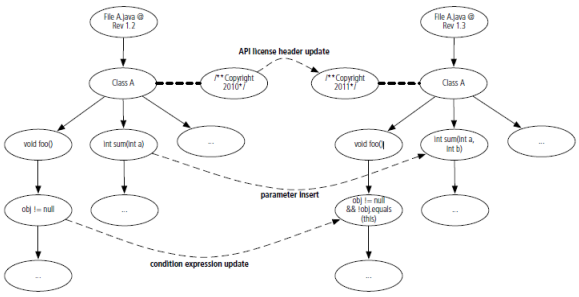
predictors build using *PCA* are useful for building regression models that predict post-release defects, (d) predictors are only accurate when obtain from the same or similar projects.

This approach can be generalized to predict arbitrary measures of quality, as long as we can extract the right information from the project's history. The general workflow is the following: decompose system in entities, build a function that assigns a quality measure to an entity, have a set of metrics and a metric functions that assigns a value to each entity, determine correlation of metrics to the quality measure and use PCA to select the most relevant set, use the principal components to predict quality of new entities.

Giger et al. [23] present bug prediction models at method level. In comparison to previous file or module level techniques, this increases the granularity of the prediction and thus reduces manual inspection (developers don't have to inspect a whole file).

The models are based on source code metrics that are applicable on method level (fig. 11b) while change metrics are based on fine-grained operations extracted from AST comparisons (tree edit operations needed to transform one AST into the other, combined with semantic information from the source code, fig. 11a).

By using an extensive test set of multiple open source Java projects and by labeling each method as bug-prone or not bug-prone (using historical VCS data) they were able to measure the efficacy of different classifiers. The classifiers were trained with both source and change metrics and each separately. The source metrics alone performed significantly worse than the other two and suffer from low precision values (around %50). The change metrics (combined or not with the source ones) perform significantly better (> 80 % precision) and the type of classifier does not significantly affect the results (fig. 11c).



(a) Fine grained code changes extracted from AST comparisons

Metric Name	Description (applies to method level)
fanIN	Number of methods that reference a given method
fanOUT	Number of methods referenced by a given method
localVar	Number of local variables in the body of a method
parameters	Number of parameters in the declaration
commentTo CodeRatio	Ratio of comments to source code (line based)
countPath	Number of possible paths in the body of a method
complexity	McCabe Cyclomatic complexity of a method
execStmt	Number of executable source code statements
maxNesting	Maximum nested depth of all control structures

(b) Method level metrics used for prediction

	CM			SCM			CM&SCM		
	AUC	P	R	AUC	P	R	AUC	P	R
RndFor	.95	.84	.88	.72	.5	.64	.95	.85	.95
SVM	.96	.83	.86	.7	.48	.63	.95	.8	**.96**
BN	.96	.82	.86	.73	.46	.73	.96	.81	**.96**
J48	.95	.84	.82	.69	.56	.58	.91	.83	.89

(c) Precision, recall and AUC results for the metric sets

Figure 11: Method level bug prediction

Wang et al. [24] leverage deep learning to automatically learn semantic features from source code. The aim is to apply this knowledge into defect prediction, which traditionally uses syntactic features to build the models. In order to make accurate prediction, the features need to be discriminative, but traditional features cannot distinguish code regions with different semantics (see for example fig. 12a).

A Deep Belief Neural Network is used to learn the semantic features from input vectors that contain tokens extracted from the AST's (code is parsed into tokens, tokens are then mapped into integers, which then form the vectors). Three main categories of AST nodes are extracted: a) nodes of method invocations and class instance creations, b) declaration nodes, c) and control flow nodes (see fig. 12b for the general workflow).

To handle noise in data, the edit distance between the token sequences along with the *Closest List Noise Identification* approach is used (compare instance label against its k-nearest neighbors). Additionally, infrequent tokens are filtered out of the training process.

The DBN is tested against models trained with traditional features and models trained with the AST nodes. The DL approach outperforms the traditional methods, with an average improvement in precision and recall of 14% and 11% respectively.

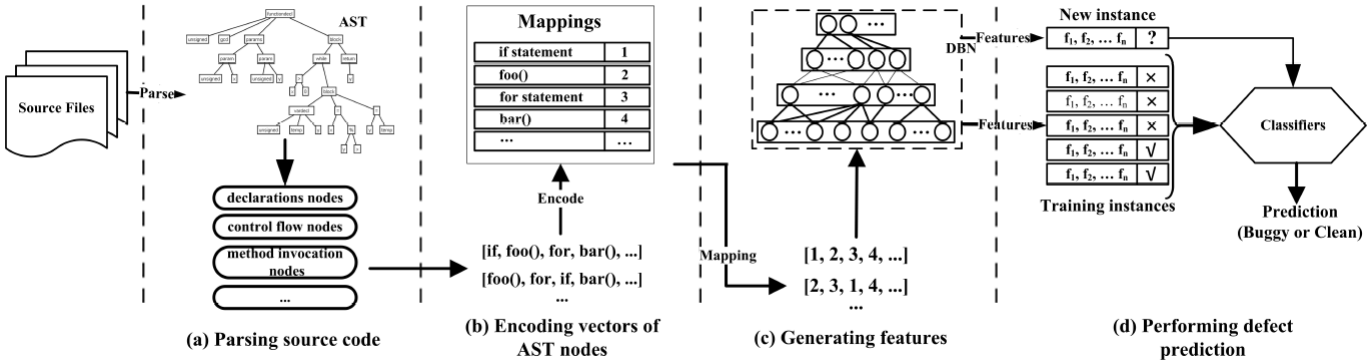
```
1 int i = 9;
2 if (i == 9) {
3   foo();
4   for (i = 0; i < 10;
        i++) {
5     bar();
6   }
7 }
```

File1.java

```
1 int i = 9;
2 foo();
3 for (i = 0; i < 10; i
      ++) {
4   if (i == 9) {
5     bar();
6   }
7 }
```

File2.java

(a) Example of programs with same syntax (tokens) but different semantics



(b) Workflow for the semantic learning process

Yang et al. [25] propose a deep learning technique to detect defect-prone changes (just-in-time defect prediction, i.e. inside commits). The advantage of this granularity is that there is a smaller amount of code to check and that it is easy to decide which developer should fix a bug (the one who committed the code). They use a two phase approach: a feature selection phase and a machine learning phase.

The feature selection phase is to decide the best set of features to use to train the model. The data is pre-processed to in two steps: data is first normalized and then random under-sampling is used to balance the categories of buggy or not buggy changes. Since in logistic regression each feature is calculated independently, new features cannot be created by combining existing ones. For that reason, they leverage *Deep Belief Networks* to generate a more expressive feature set.

The logistic regression model is trained with the new feature set and evaluated with a cost effectiveness measure defined as the percentage of bugs that can be discovered by inspecting the top (most relevant) 20% lines of code. On average 50% of bugs can be found in the top 20% LOC, and the feature processing step helps logistic regression achieve better results than previous approaches.

3.3 Static Analysis tools in practice

Beller et al. [26] conduct a large scale evaluation of how SA tools are used in practice in open-source systems. They study the prevalence of SA tools, their configurations and how they evolve.

By performing a survey on a 36 open-source projects, they found out that most of them used SA tools, with a relevant subset using more that one (fig. 13a). Although, most of them run the tools sporadically and without enforcing them.

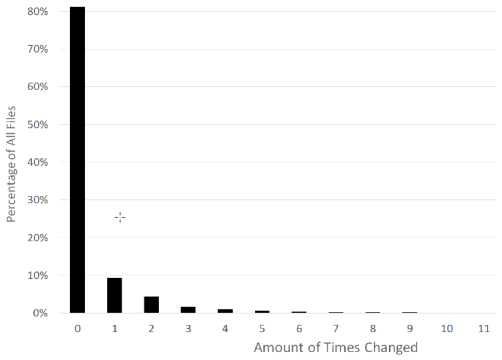
A configuration file of a SA tool, shows what rules developers deem important (enable), and what they do not deem important (disabled, perhaps because of a high false positives rate). The contents of a configuration file are hence an important indicator of how developers use SAs and how well the tool's default settings reflect its use.

In the analyzed projects, most enabled rules belong to the maintainability category, and only 35% of the enabled rules belong to a functional category. Both the majority of actively enabled and disabled rules are maintainability-related.

Most configurations change or reconfigure rules from the default configuration, but typically only one rule. Most changes are small, and a third of them happen in the first week of creation of the configuration. Also, most configuration files never change (fig. 13b).

Source	Projects	Use 1 ASAT	Use > 1 ASATs	Enforce Use
GitHub	19	36%	32%	42%
OpenHub	1	0%	0%	0%
SourceForge	3	34%	66%	0%
Gitorious	10	30%	40%	30%
Other*	3	100%	66%	33%
Total	36	41%	36%	36%

(a) Survey results of 36 open-source systems using SA tools



(b) Changes in configuration of SA tools

Figure 13: Analyzing the state of static analysis

Imtiaz et al. [27] analyze the SA usage of five large open-source systems (the tool is *Coverity*). They study the amount of actionable alerts, time for fixing alerts and the size of fixes.

They discover that 80% of alerts belong to 20% of the alert types and that the actionability rate varies from 27% to 49% depending on the project (fig. 14a). Also in the case of actionable alerts, 20% of the types causes 80% of the actionable alerts.

The median lifespan of actionable alerts varies between projects, ranging from 36 to 245 days, and the complexity of code changes is generally low. This means that developers generally take a long time to fix the alerts despite the fixes being low in complexity.

To increase the developer interaction with SA tools they suggest two solutions: (a) prioritizing the most critical alerts and (b) providing an estimate for the fix effort.

Prject	**Total Alerts**	**Eliminated Alerts**	**Actionable Alerts**	**Triaged Bug**
Linux	17133	10336 (60.3%)	6047 (36.7%)	624 (3.6%)
Firefox	12945	9522 (73.6%)	6193 (48.4%)	1062 (8.2%)
Samba	4186	3055 (73.0%)	1148 (27.4%)	102 (2.4%)
Kodi	2325	1538 (66.2%)	1146 (49.5%)	369 (15.9%)
Ovirt-engine	2906	1302 (44.8%)	905 (31.3%)	75 (2.6%)

(a) Actionability results for total alerts

Alert Type	**Im-pact**	**Occurr-ence**	**Action-ability**	**Lifespan (days)**
Resource leak	H	844.0	49.9%	121.5
Unchecked return value	M	469.0	38.7%	109.5
Logically dead code	M	385.0	44.3%	89.5
Explicit null dereferenced	M	304.0	38.4%	83.2
Dereference after null check	M	273.5	47.2%	178.0
Dereference before null check	M	254.0	62.3%	51.0
Various (a type by Coverity)	M	214.5	33.4%	660.5
Dereference null return value	M	212.0	48.1%	65.0
Uninitialized scalar variable	H	170.0	57.0%	24.5
Missing break in switch	M	141.5	41.6%	173.8

(b) Top 10 alert occurrences for C/C++

Figure 14: How do developers act on SA alerts?

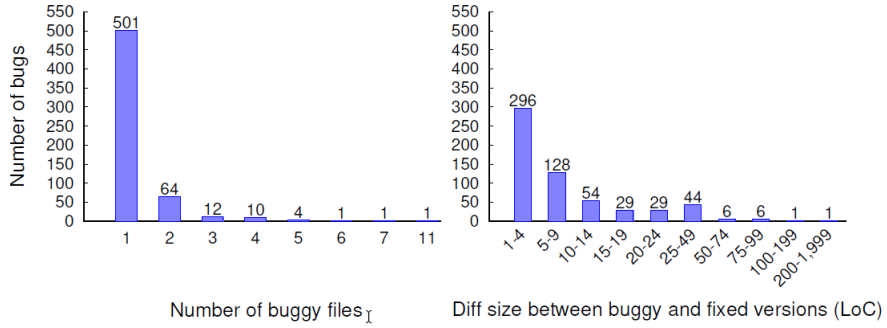
Habib and Pradel [28] study how many of all real-world bugs static bug detectors find. The results of their study show that: (a) static bug detectors find a non-negligible amount of all bugs, (b) different tools are mostly complementary to each other (see fig. 15a), and (c) current bug detectors miss the large majority of the studied bugs.

The three bug detectors together reveal 27 of the 594 studied bugs (4.5%). Some of the missed bugs could have been found by variants of the existing detectors, while most of them are domain-specific problems that do not match any existing bug pattern that the SA tool have. By manually analyzing a small subset of 20 bugs, 14 of them were domain-specific and not related to any pattern supported by the checkers, while 6 of them were near-misses that could have been detected with a more powerful variant of the tool.

They also found that the majority of bug fixes is limited in size and that most bugs are clustered on a small percentage of files (fig. 15b).

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
Total:	**31**
*Total of **27** unique bugs*	

(a) Distribution of found bugs per tool



(b) Bugs/file and lines/bugfix

Figure 15: How many of all bugs do we find?

Sadowski et al. [6] provide an overview of the process that Google underwent to increase the developer interaction with SA tools. They list a number of shortcomings that hinder large scale adoption of such tools and suggest solutions that proved effective in their company.

The main reasons developers ignore of lose faith in SA tools are: (a) **no integration in workflow** (most important reason), (b) non actionable warnings, (c) reported bugs do not manifest in practice, (d) suggested bug is too risky/expensive to fix, (e) warnings are not understood.

Google switched from the dashboard based *FindBugs* tool (whose warnings were mostly ignored for two main reasons: developers lost faith because of false positives or alerts that were not important, and because the warnings came to late in the development workflow), to another better integrated approach. According to their findings, reporting issues sooner is better: moving as many checks into the compiler is the way to go. When possible, fixes are suggested or carried out automatically. A second place to show alerts that relate to high impact bugs, is the code review platform (for alerts with no simple fix). Code review is also a good context for reporting relatively less-important issues like stylistic problems or opportunities to simplify code.

Another key point in making SA tools more valuable for the developers, is integrating their feedback, whether they accept or not the alerts proposed by the tool (for ex. adding a button for each alert *Useful/Not Useful*). An additional workflow integration point is *gating commits*: blocking a commit when a check fails (used for check with a low false positive rate).

4 Collecting and analyzing data

Given the company context, a single tool approach was chosen as to not introduce extra dependencies/complexity (in contrast to combining different SA tools).
We use Clang AST [29] and Clang Tidy [30] to analyze the code. We chose them because both are reliable and extensible open-source tools. Moreover, and most important, both are supported but the company's C++ codebase.

4.1 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.

4.2 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 [30].

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.

The following checks were regarded as relevant by the company and used during code analysis:

- Clang Static Analyzer checks
- Checks related to C++ Core Guidelines
- Checks related to CERT Secure Coding Guidelines
- Checks related to performance

4.3 Workflow for collecting data

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 (see also fig. 16):

- 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.

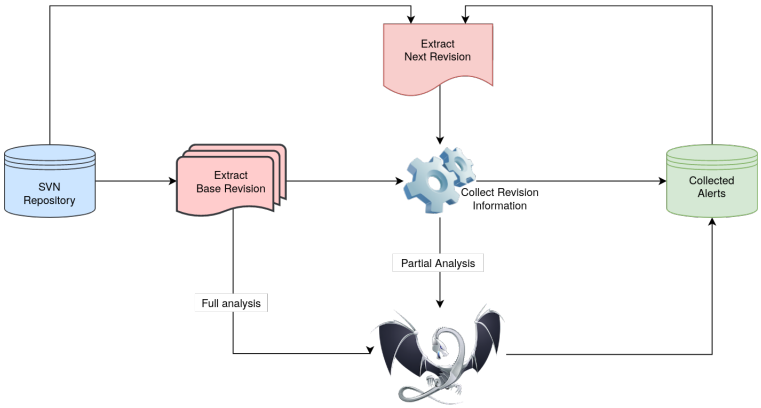


Figure 16: Collecting alerts and other information

4.4 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.

4.5 Data overview

A typical release in the *OMP's* codebase consists of around *27.000 .cpp* files containing over *4.000.000* lines and coded by more than 300 developers (in its entire history). The following section will provide a numerical overview of a portion of revisions analyzed from one of its releases

We analyzed 1868 revisions, dating from *09/2019* to *05/2020*. From those revision, 1313 contain bug fixes.

4.5.1 Revision data

We take a look at the main differences between normal and bug-fix commits.

Table 1: File/Method change metrics in collected revision

	Modified Files	Added Files	Deleted Files	Modified Methods
			MEAN	
Normal	7	1.4	0.98	6.8
Bug-Fix	4.3	0.52	0.21	4.64
			MEDIAN	
Normal	2	0	0	1
Bug-Fix	3	0	0	2

Table 2: Line change metrics in collected revision

	Added Lines	Deleted Lines	Modified Lines	Growth Lines
			MEAN	
Normal	99.8	40.2	140	59.7
Bug-Fix	47.8	24.5	72.3	23.3
			MEDIAN	
Normal	16	6	26	2
Bug-Fix	9	4	14	2

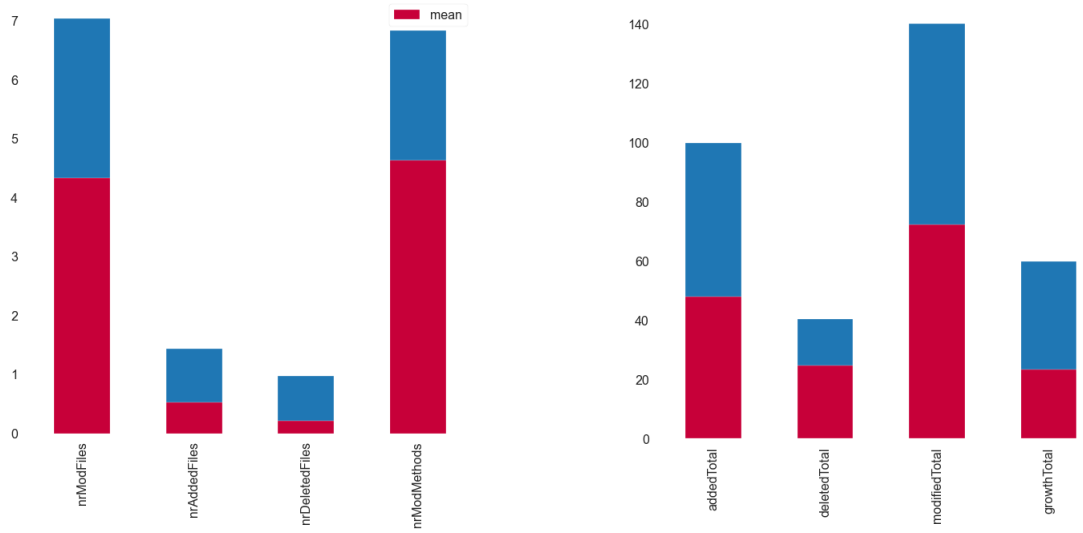
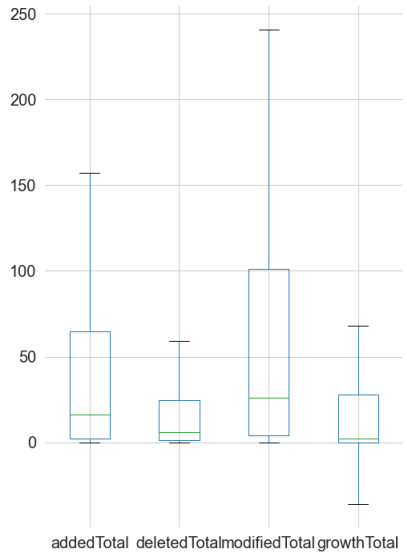
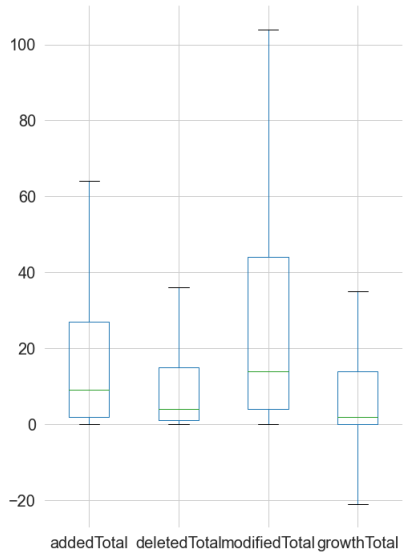


Figure 17: Average changes (*red=bug revision, blue=normal*)

As can be seen from the data (table 1, table 2, and fig. 17), there is significant difference between the amount of changes that happen during a normal and a bug-fix revision, with the later being almost half the size. That is important because the bug fix changes can be better located. There is also a big difference between mean and median values of the collected features, shifting the values of the former to be higher (can be also seen on the box plots on fig. 18 and fig. 19). That means that there are certain revision that are abnormally large compared to others and that negatively impact the quality of the data.

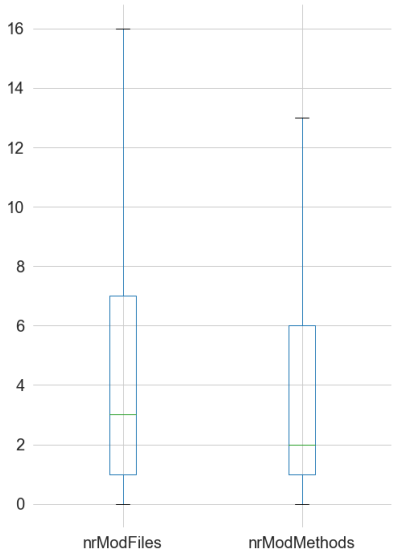


(a) Box plots for changed lines in normal revisions

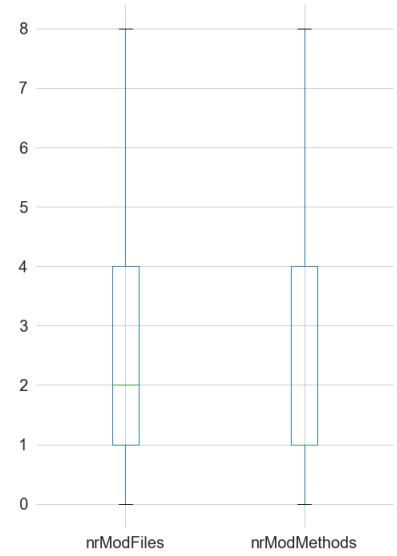


(b) Box plots for changed lines in buggy revisions

Figure 18: Box plots of changed lines



(a) Box plots for changed files/methods in normal revisions



(b) Box plots for changed files/methods in buggy revisions

Figure 19: Box plots of changed files/methods

4.5.2 Closed Alerts

	Number of alert	% of alerts	Number of closed alerts	% of closed alerts
cppcoreguidelines	177812	93.8%	78785	95.6%
performance	6010	3.2%	2297	2.8%
cert	4673	2.5%	948	1.2%
clang	1040	0.5%	346	0.4%

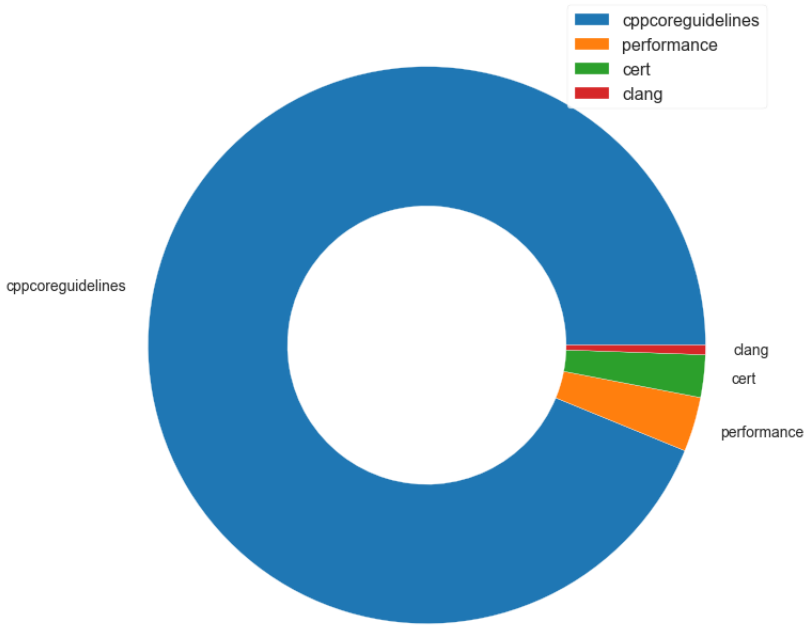


Figure 20: Distribution of collected alerts per alert category

From the distribution of collected alerts, we can see that it is dominated by the *cppcoreguidelines* checks, while the remaining three categories consist only of 4.4% of the total alerts.

	Number of closed alerts	% of closed alerts inside category
cppcoreguidelines	78785	44%
performance	2297	38%
cert	948	20%
clang	346	33%

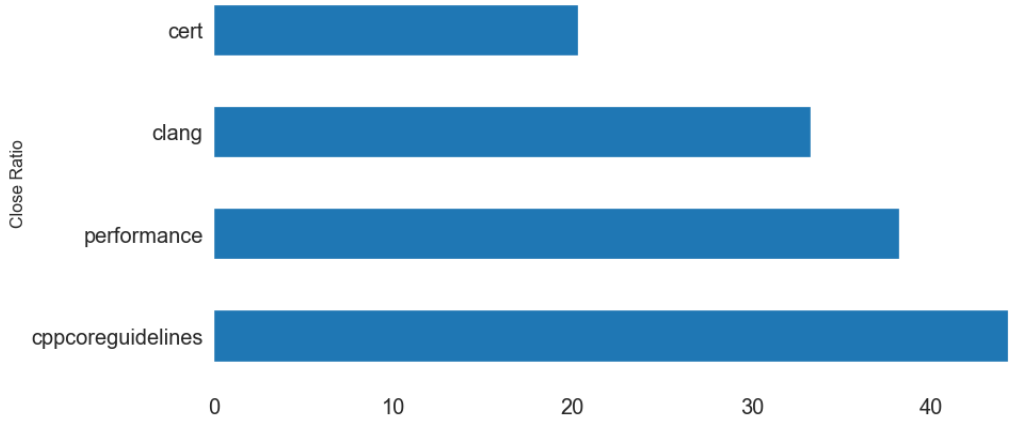
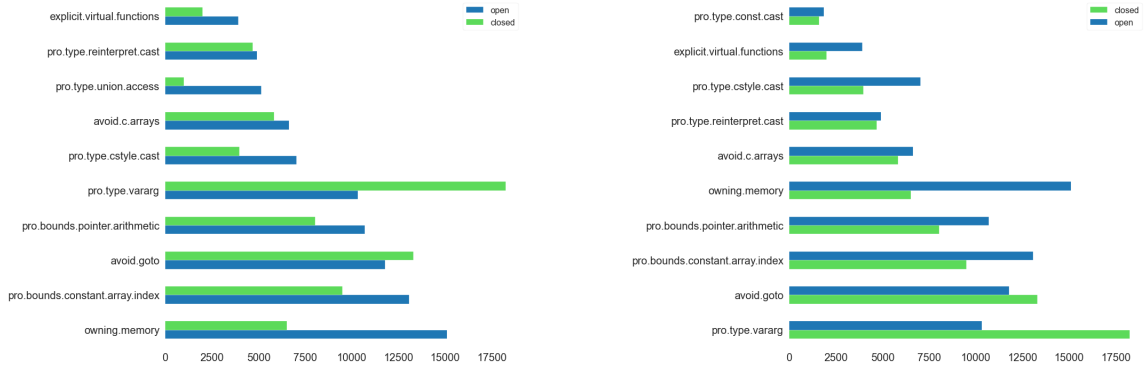


Figure 21: Ratio of closed alerts per category

The ratio of closed alerts inside each category also differs greatly, from 44% of *cppcoreguidelines* to 20% of the *cert* alerts.

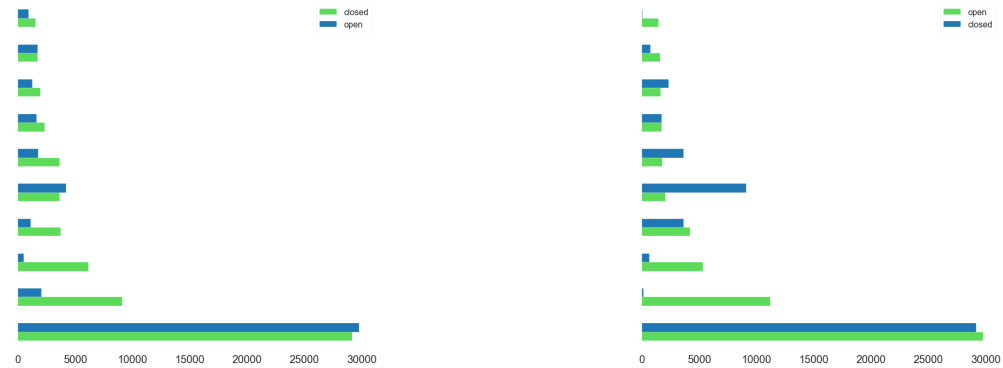
By analyzing the lifetime of closed alerts, in terms of number of revisions, we can see that alerts take before getting closed (see fig. 22). Around a quart of alerts gets closed within 200 first revisions after being opened. That is not a good indication because it may mean that they do not directly disappear from targeted action from developers.



(a) Bar plot for alert types with most open alerts

(b) Bar plot for alert types with most closed alerts

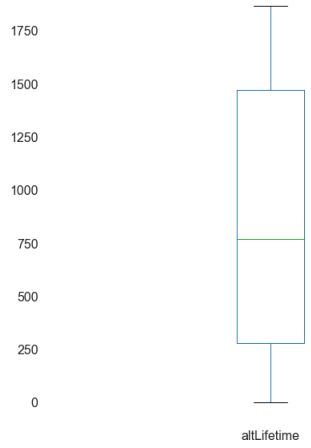
Figure 23: Most open/closed alert types



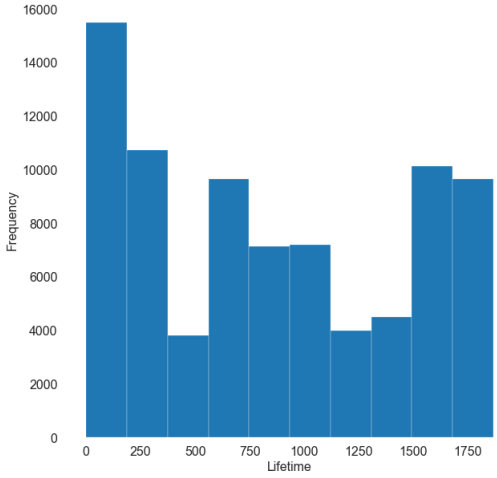
(a) Bar plot for packages with most open alerts

(b) Bar plot for packages with most closed alerts

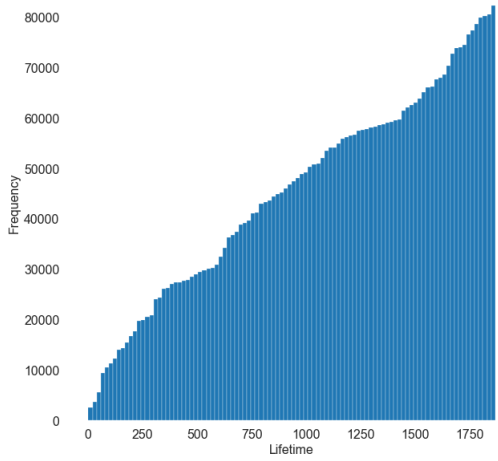
Figure 24: Most packages with open/closed alerts



(a) Box plot for closed alert lifetime



(b) Histogram for closed alert lifetime



(c) Cumulative plot for closed alert lifetime

Figure 22: Alert lifetime graphs

From the plots in fig. 23 and fig. 24, we can see that the distribution of open/closed alerts is dominated on an package level. The package *cc_corstationlinks* contains half of the closed alerts. There exists thus a heavy imbalance in terms of the location of closed alerts.

4.6 Dealing with unbalanced and noisy data

Our automatically collected data suffers from two main problems, imbalance and noise.

Kim et al. [13] propose a method for dealing with noisy data in the context of defect prediction algorithms. They also provide guidelines for acceptable noise levels and study the impact of different amounts of noise on the prediction performance. They found out that performance, in terms of f-measure, is affected more by the presence of false positives and that the tested algorithms were robust to the presence of false negatives. When noise levels reach 20-35% of both FP and FN, the performance decreases significantly, especially on small datasets. The proposed algorithm for cleaning noisy data by [13], calculates the ratio of the top N most similar instances of each item that have a different class label. If that ratio exceeds a certain threshold, then that item is considered as noisy.

Batista et al. [31], perform a comparative evaluation of different dataset balancing techniques. They claim that imbalance alone is not the only reason for poor classifier results, but that it is also related to noisy/overlapping data. According to their experiments, over-sampling methods perform generally better and *SMOTE* + Tomek/ENN provide good results for datasets with few positive examples.

Given the nature of our approach to detecting actionable alerts, it seems plausible that the amount of false positives will be non negligible, and thus there is a high risk that there will be a significant performance impact. Since, the number of examples of the positive class is also smaller than the negative one, there seems to be a necessity to apply a combination of balancing and cleaning to the dataset.

The *imblearn* library ([32]) contains different techniques to clean the dataset by performing under-sampling:

- **Tomek's Links** between two samples is defined as: $d(x, y) < d(x, z)\, and\, d(x, y) < d(y, z)$ for all other samples z. According to the strategy one or both samples forming a Tomek link are removed.
- **Clean dataset by using nearest neighbours**
  - *Edited Nearest Neighbours*: applies a nearest-neighbors algorithm and removes samples which do not agree enough with their neighborhood.
  - *Repeated Edited Nearest Neighbours*: same as before but the algorithm is repeated multiple times.
  - *AllKNN*: same as before but with each run the number of nearest neighbors is increased.
  - *Condensed Nearest Neighbors*: uses a 1 nearest neighbor rule to iteratively decide if a sample should be removed (adding a sample at a time to the minority set).
  - *One Sided Selection*: combines Tomek's Links with Condensed Nearest Neighbors
  - *Neighbourhood Cleaning Rule* will focus on cleaning the data than condensing them.

To see the effect of these methods, we run them on a test dataset. We avoid the two most invasive alerts that together count for more than half of the total number of alerts (around 190k alerts remain).

The following table counts the number of samples cleaned by each method (default parameters):

	Removed samples
Tomek's Links	1
Edited Nearest Neighbours	34
Repeated Edited Nearest Neighbours	38
AllKNN	37
Condensed Nearest Neighbors	106949
One Sided Selection	204
Neighbourhood Cleaning Rule	95

From seven tested methods only one removes a significant amount of samples (56% less samples respectively). From further inspection it turned out that the method removed almost all samples of the majority class, which is not something that we want. The other methods have barely an effect on the dataset. That could mean that the dataset does not contain a lot of noise.

TO DO: maybe insert before and after graph for data cleaning/balancing? Not enough removed samples!

5 Ranking the output of Static Analysis

We analyze the performance of four different approaches to ranking SA alerts:

- 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 rank alert types based on that information.
- Detecting bug-prone methods and prioritizing alerts to those parts of code.

5.1 Feedback Rank with Z-Score

5.1.1 Feedback Rank

Feedback Rank [15] is a simple technique that ranks alerts on the probability of them being actionable. It takes as input three location features for predicting if an alert is actionable: package, file, and function where the alert being analyzed is situated.

As explained on the literature review (section 3.1.1), Feedback Rank is based on the assumption that true and false positives are clustered by code locality. The project space is divided into two major regions, one that contains mostly true positive, and one that contains mostly false positives (*TPRegion, TFRegion*). Each package, file or function is considered to belong to one of these two regions.

A Bayesian Network (BN) is used to calculate the probabilities of an alert or cluster of alerts belonging to a certain region. The network consist thus of one node for each of the artifacts (package, file, function), which in turn connect to a node representing alerts that belong to that specific location combination (see fig. 25). The initial configuration of the network is learned from historical data (actionable alerts calculated as explained on section 4.3). The probabilities of each artifact belonging to one of these two regions, *TPRegion* or *TFRegion* are adjusted when training the network with the extracted data.

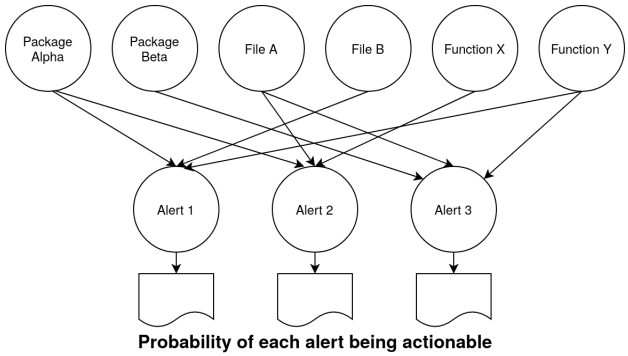


Figure 25: Example of a simple Bayes Network for predicting 3 alerts

According to the Kremenek et al. [15], 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. In this implementation, a static version is used. That means that the network is trained once and remains unchanged when predicting the rest of the alerts.

To construct the BN, we used the Pomegranate library [33], trained with the extracted alert data from the version history.

5.1.2 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 [14], 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 (see section 3.1.1 for more details).

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 indispensable.

5.2 Detecting Actionable Alerts via Machine Learning

Different techniques focused on automatically classifying alerts in true/false positives or actionable/unactionable alerts, by constructing classifiers based on code or change metrics [9, 17].

Instead of classifying alerts as true or false positives, we focus on Actionable Alerts which are 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). Actionable alert is a less restrictive definition and makes it easier to collect data. Classifying alerts as true or false would require an oracle or a large and representative dataset generated manually. Moreover, from a developer's perspective, AAs can be more useful. A true positive alert may not be necessarily important to developers and thus be equally useless as a false one (e.g, low severity, no impact on the user side).

We followed the example of Heckman and Williams [17] to conduct our research since it contains an agglomeration of alert characteristics (AC) collected from other papers.

The workflow, as explained in section 4.3, 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.

We use the scikit-learn library [34] to perform ML experiments in our evaluation.

5.2.1 Alert Characteristics

The collected features can be classified in four main categories: alert information, source code metrics, churn metrics and version history information. The complete list of the collected features can be seen on table 3.

Table 3: List of features used to predict actionable alerts

Feature	Description
category	Alert category
type	Alert type
function	Name of function where alert is located
class	Name of class where alert is located
file	Name of file where alert is located
package	Name of folder where alert is located
openRevision	Revision number when alert appeared (default base revision)
alertLifetime	current revision number - open revision number for alert
functionSize	Number of statements in function where alert is located
fileSize	Number of statements in file where alert is located
nrFunctions	Number of functions in file
nrClasses	Number of classes in file
nrParameters	Number of parameters in function where alert is located
functionComplexity	Cyclomatic completicy of function where alert is located
addedFile	number of lines added to file in this revision
deletedFile	number of lines deleted from file in this revision
modifiedFile	addedFile + deletedFile
growthFile	addedFile - deletedFile
addedTotal	number of lines added in this revision
deletedTotal	number of lines deleted in this revision
modifiedTotal	addedTotal + deletedTotal
growthTotal	addedTotal - deletedTotal
firstChangeFile	Revision number of first file change (default base revision)

lastChangeFile	Revision number of last file change (default base revision)
firstChangePackage	Revision number of first change in package (default base revision)
lastChangePackage	Revision number of last change in package (default base revision)
fileAge	lastChangeFile - firstChangeFile
fileStaleness	current revision - last change revision for file
packageStaleness	current revision - last change revision for package
lastKnownDev	Last developer to change file
mostPresentDev	Developer that made most changes to file

5.3 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 extra valuable those alerts that potentially signaled future bugs. The concept of bug related lines (BRL) is used by Kim and Ernst [1], and by Liang et al. [10].

BRLs are calculated as follows:

- We start at a base version of the source code 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 26 shows a summarization of the general process we follow to calculate BRLs.

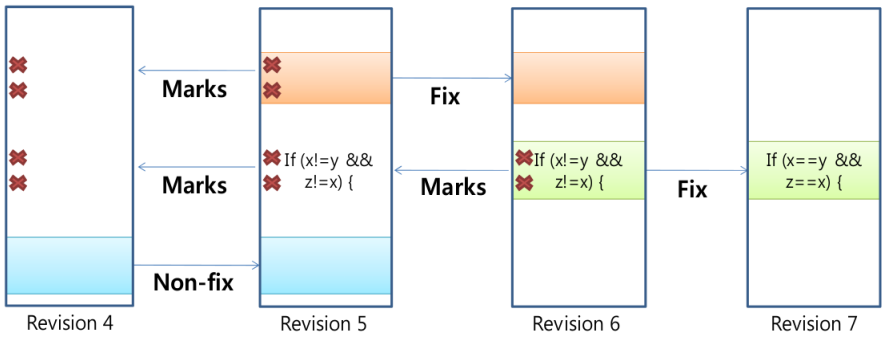


Figure 26: Calculating BRLs ([1])

Since the number 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 BRLs belong to and also consider alerts inside those methods as valuable. That allows us to extend the dataset, but also potentially weakens the data by introducing more noise.

Following the approach used by Kim and Ernst [1], we can calculate weights for warning types, and use the weights to rank alerts based on how good is that particular alert type in predicting bugs. The algorithm used by the authors is rather simple and calculates weights based on two input parameters, α and β. The former used to increase the weights of alert types belonging to the collected *BRL/BRM* set, while the later used for the actionable alert set (section 5.4).

5.4 Method Bug prediction

Following the example of Boogerd and Moonen [16], which aims to prioritize alerts based on the execution likelihood of the code pointed by the alert, we present a similar approach. 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. On the other hand, finer granularity than method level would be too hard to predict.

We follow the example of Giger et al. [23] 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. The complete list of the collected features can be seen on table 4.

Algorithm 1: Alert type priotitization algorithm

Data: Collected bug-related alerts and actionable alerts
Result: Weighted alert types
$\alpha, \beta = x, y$;
$w_t = 0 \; for \; t \; in \; alertTypes$;
for *alert in collectedAlerts* **do**
  $w_t = typeOf(alert)$;
  if *alert pointed to a BRL or BRM* **then**
    $w_t = w_t + \alpha$
  end
  else if *alert is actionable* **then**
    $w_t = w_t + \beta$
  end
end
$w_t = \frac{w_t}{|alerts \, of \, type \, t|}$

Table 4: List of features used to predict buggy methods

Metric Name	Description
methodHistories	Number of times a method was changed
authors	Number of distinct authors that changed a method
stmtAdded	Sum of all source code statements added to a method
maxStmtAdded	Maximum number of source code statements added to a method body for all method histories
avgStmtAdded	Average number of source code statements added to a method body per method history
stmtDeleted	Sum of all source code statements deleted from a method body over all method histories
maxStmtDeleted	Maximum number of source code statements deleted from a method body for all method histories
avgStmtDeleted	Average number of source code statements deleted from a method body per method history
churn	Sum of stmtAdded - stmtDeleted over all method histories
maxChurn	Maximum churn for all method histories
avgChurn	Average churn per method history
decl	Number of method declaration changes over all method histories
cond	Number of condition expression changes in a method body over all revisions
elseAdded	Number of added else-parts in a method body over all revisions
elseDeleted	Number of deleted else-parts from a method body over all revisions
cyclomaticComplexity	Current cyclomatic complexity of method
nestingDepth	Current nesting depth of method
totalStatements	Current number of statements in method
nrPaths	Current number of paths in method
nrDeclarations	Current number of declarations in method

6 Evaluating the approaches

6.1 Evaluation methods

To evaluate and compare the approaches different techniques are used:

- **K-Fold Cross Validation**: where the data is randomly divided into folds and one of those is used for testing while the rest for training. It is often used in papers containing ML approaches to ranking alerts, though it has its drawbacks. As shown in [2] and as can be seen on fig. 27, this approach uses dependent variables (most of the extracted features), that may not be available at prediction time in a real world scenario. That can lead to unreliable and excessively optimistic results. Cross Validation was only used for selecting the right classifier for this task, and not for evaluating the ranking approaches.
- **Release/Revision based testing**: avoids the drawback of the previous method, by using a *"horizontal"* train/test strategy. We fix a certain point in time (revision or release) which defines what the train set (before that point) and test set (after that point) will be. This trains the algorithms with more realistic data, but it also has its drawbacks. In a scenario where the dataset is imbalanced, a *"horizontal"* 80/20% split may leave us with very little data. Also, unlike in cross validation, we cannot train and test different models.

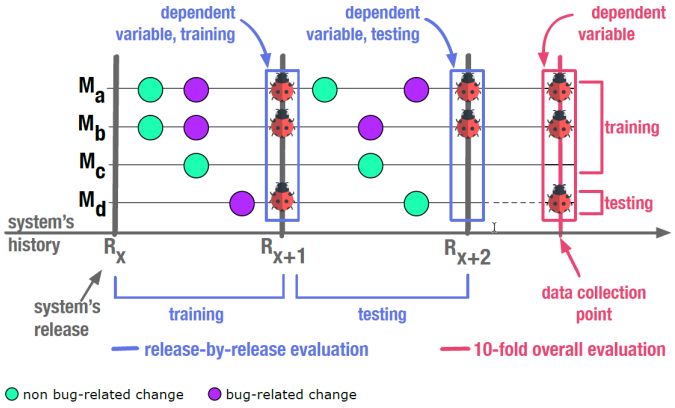


Figure 27: Release-based vs K-Fold ([2])

- **Average FP to TP [15]**: Let N be the total number of actionable alerts in a set of reports, and FP_j is the number of false positives to inspect for finding the *j*th actionable alert, starting from the last actionable alert inspected.

$$AVG_{FP-TP}(R) = \frac{\sum_{i=1}^{N} FP_j}{N}$$

- **S(R) metric [14]**: Let *N* be the total number of alerts and *act* the number of actionable ones. Let *R(i)* denote the cumulative number of actionable alerts found by a ranking scheme *R* on the *i*th inspection.

$$S(R) = \sum_{i=1}^{N} [min(i, act) - R(i)]$$

- **Comparison against random rankings**: where we compare if the ranking produced by a model is better than a random order of alerts. We calculate that by using the two previously defined metrics, Average FP to TP and S(R).
  Also we compare the number of actionable alerts found in the first *X* places in a ranked list, compared to a random list.
- **Cumulative ranking graph**: formed by plotting the number of actionable alerts found within the first N inspections.

6.2 Evaluation Metrics

Along with the classic evaluation metrics like *accuracy*, *precision* and *recall*, other metrics are used that are more appropriate for imbalanced data [35, 36].

$$Sensitivity = \frac{TP}{TP + FN}$$
$$Specificity = \frac{TN}{FP + TN}$$
$$G - Mean = \sqrt{Sensitivity * Specificity}$$

Sensitivity, or True Positive Rate is the percentage of positive examples which are correctly classified.
Specificity, or True Negative Rate, is the percentage of negative examples which are correctly classified.
G-Mean is the geometric mean of the sensitivity and specificity.

$$AUC = \frac{Sensitivity + Specificity}{2} \qquad (*approximation\ using\ trapezoid\ rule)$$

The *AUC* of a binary classifier is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance.

$$IBA_{\alpha} = (1 + \alpha * (Sensitivity - Specificity)) * Sensitivity * Specificity$$

The *Index of Balanced Accuracy* used for evaluating learning processes in two-class imbalanced domains. The method combines an unbiased index of its overall accuracy and a measure about how dominant is the class with the highest individual accuracy rate.

6.3 Results of the approaches

Experiments were run by training methods on the first 80% items of the dataset and testing on the remaining 20%.

For the actionable alert (section 5.2) and method bug prediction (section 5.4) approaches, different classifiers were considered. Three classifiers were considered, based on the nature of the data (categorical variables, ordinal, discrete values) and the state of the art in the field: *Random Forest*, *AdaBoost*, and *LightGBM*.

To decide between the three, a 10-fold validation was performed and timed in a subset of the data. *LightGBM* was finally chosen because of the good performance and speed (*AdaBoost* does not support multi-threading in the *scikit* library). The results can be seen on table 5.

Table 5: Classifier performance on a 10-fold validation on actionable alerts

	Precision	Recall	Accuracy	Time
Random Forest (30 threads)	89	97	90	14.8s
LightGBM (30 threads)	98	93	94	9.1s
AdaBoost	96	94	94	89s

In order to evaluate the ranking, two metrics are used to compare the ranked alerts with a random order (*S(R)* and *Average FP-to-TP*). The experiments are run multiple times (by shuffling the random order and calculating the metrics again). Also the number of valuable alerts in the top *X* places of the improved ranking is compared against random ranking.

6.3.1 Actionable Alerts

Alerts are ranked based on the probabiliy score produced by the classifier for the prediction target (actionable or not).

As can be seen on table 6, the choice of the balancing technique or cleaning method has little impact on the metrics (they are quite good for each case). The best performing one is *ADASYN* (fig. 28.)

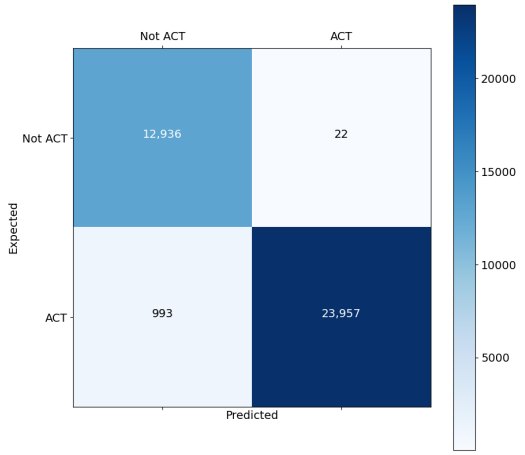
The order produced by the *LightGBM* classifier always outperforms the random order and it also contains around 50% more actionable alerts in the top 100 or 1000 places (table 7).

Furthermore, as can be seen on the cumulative graph on fig. 29, the order of the ranked alerts is close to perfect.

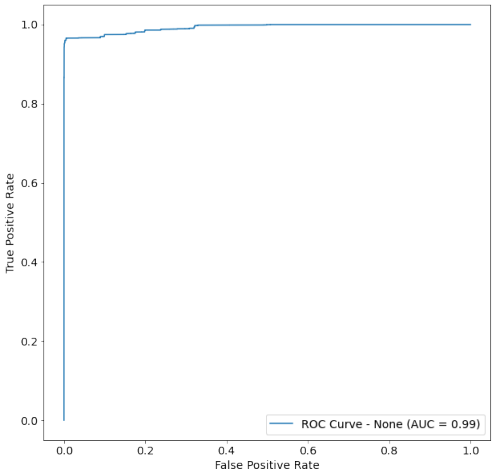
Table 6: Results of different cleaning and balancing methods for the Actionable Alerts approach

	Precision	**Recall**	**Specificity**	**F1-Score**	**Geometric**	**IBA**
Random Over Sampler	94%	93%	95%	93%	94%	88%
Random Under Sampler	95%	94%	96%	95%	95%	90%
SMOTE	94%	94%	96%	94%	95%	89%
ADASYN	**97%**	**97%**	**98%**	**97%**	**98%**	**95%**
SMOTEENN	94%	94%	96%	94%	95%	90%
SMOTETomek	95%	94%	96%	94%	95%	90%
EditedNearestNeighbours	95%	94%	96%	94%	95%	90%
RepeatedEditedNearestNeighbours	95%	95%	96%	95%	96%	91%
AllKNN	95%	95%	96%	95%	96%	91%
OneSidedSelection	95%	94%	96%	94%	95%	90%
NeighbourhoodCleaningRule	95%	95%	96%	95%	95%	90%

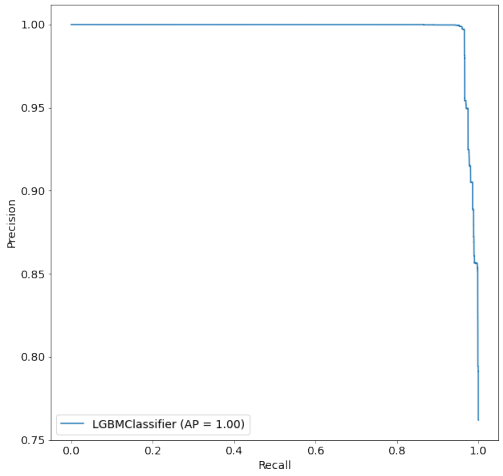
Calculated using the balanced classification report of imblearn [32]



(a) Confusion matrix



(b) ROC Curve



(c) Precision/Recall curve

Figure 28: Actionable alerts with ADASYN

Table 7: Comparison against random ranking for the Actionable Alerts approach

	S(R) metric	Average FP to TP	Number Actionable
Better than random Top 100 alerts	1000 out of 1000 runs	1000 out of 1000	52% more than random ranking
Better than random Top 1000 alerts	1000 out of 1000 runs	1000 out of 1000	51% more than random ranking
Better than random All alerts	10 out of 10 runs	10 out of 10	

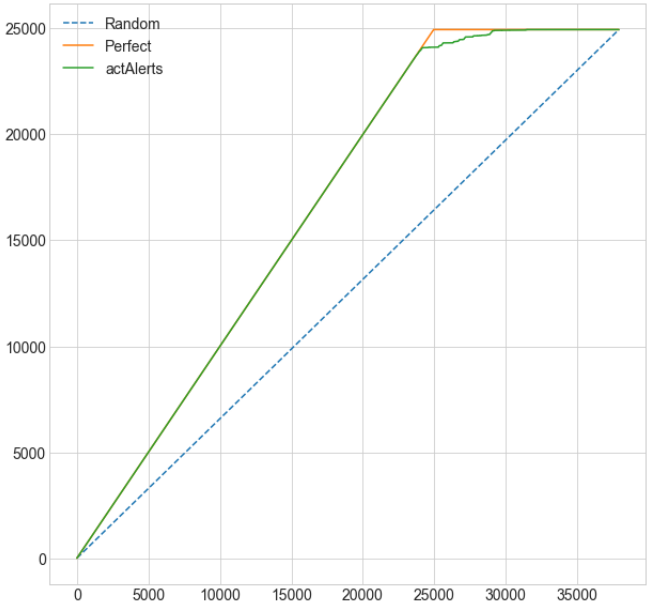


Figure 29: Cumulative ranking graph for Actionable Alerts

6.4 Method Bug Prediction

The overall results for predicting if a method will be buggy in the future, are far from optimal. Predicting bugs on a finer granularity is a hard task, but for the purpose of our use, prioritizing alerts on methods with a higher probability of being buggy, the actual achieved performance can still be regarded as a good starting point. As can be seen on table 8, the choice of the balancing technique or cleaning method has a clear impact on some the metrics. The best performing one is *One Sided Selection* (fig. 30).

Even though the main purpose of this method is not to detect actionable alerts in general, but to give extra attention to alerts belonging to buggy methods, we conduct an experiment to see if we can rank actionable alerts by using the probability of a method being buggy.

Having calculated the set of buggy methods, we can also explore if there is a correlation between the amount of alerts of a particular category/type and the method being buggy. As can be noticed on fig. 31, the amount of correlation is limited.

After calculating the probability of a method being buggy on the test set, we merge that set with the one containing actionable alerts (thus only those alerts that are located on the methods in the test set). Alerts are ranked based on the probability score produced by the classifier for the prediction target (buggy method or not).

Alerts ranked by bug-prone method probability outperform the random order in the first 100 and 1000 alerts, but not on the whole dataset (table 9).

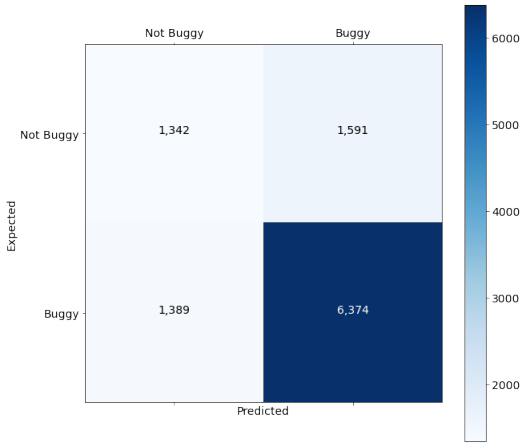
On the cumulative graphs we can see that until the first 2000 alerts, the improved ranking significantly outperforms the random one (fig. 32).

From the results we can conclude that ranking alerts based on the method being bug-prone provides an improvement against the random order.

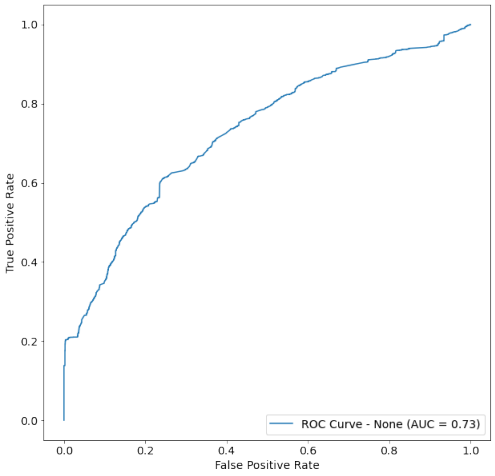
Table 8: Results of different cleaning and balancing methods for the Method Bug Prediction approach

	Precision	**Recall**	**Specificity**	**F1-Score**	**Geometric**	**IBA**
Random Over Sampler	64%	71%	34%	65%	33%	11%
Random Under Sampler	67%	72%	36%	66%	37%	14%
SMOTE	66%	72%	35%	66%	35%	13%
ADASYN	66%	72%	34%	65%	32%	11%
SMOTEENN	64%	71%	33%	64%	30%	10%
SMOTETomek	66%	72%	36%	66%	37%	14%
EditedNearestNeighbours	68%	70%	45%	68%	51%	26%
RepeatedEditedNearestNeighbours	68%	71%	45%	69%	51%	27%
AllKNN	68%	71%	46%	69%	52%	27%
OneSidedSelection	**71%**	**72%**	**54%**	**72%**	**60%**	**37%**
NeighbourhoodCleaningRule	69%	72%	47%	70%	53%	28%

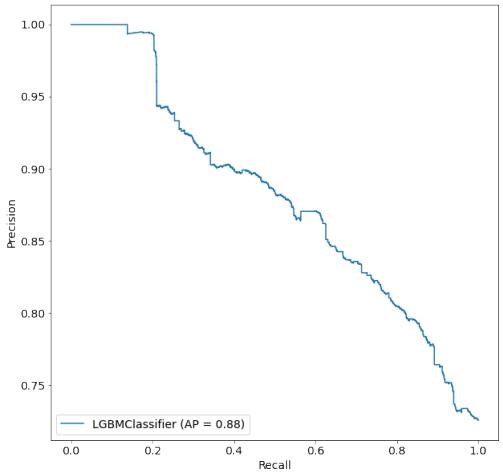
Calculated using the balanced classification report of imblearn [32]



(a) Confusion matrix

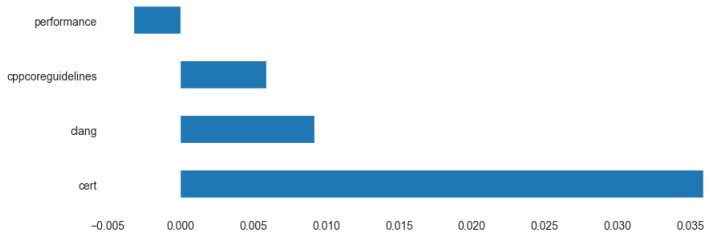


(b) ROC Curve

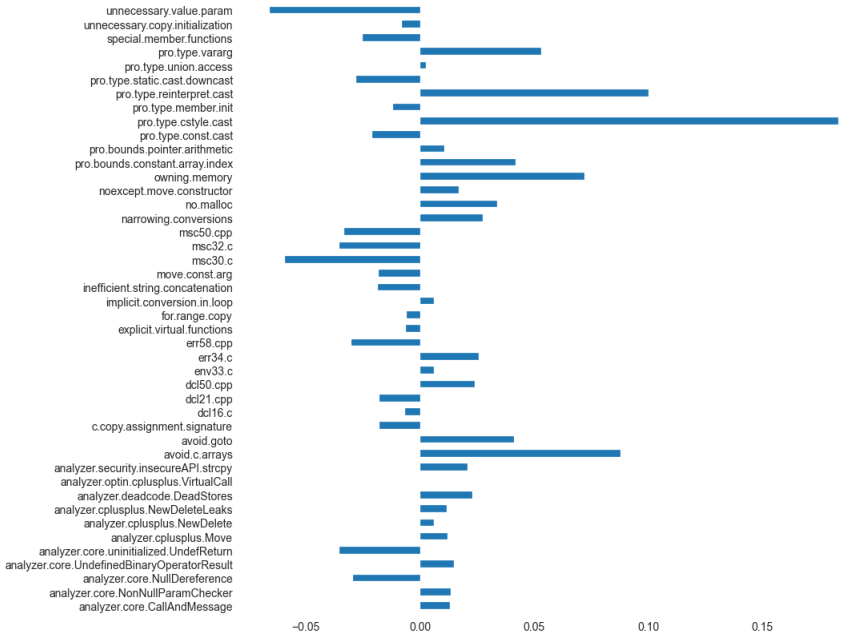


(c) Precision/Recall curve

Figure 30: Method Bug Prediction with One Sided Selection



(a) Category correlations

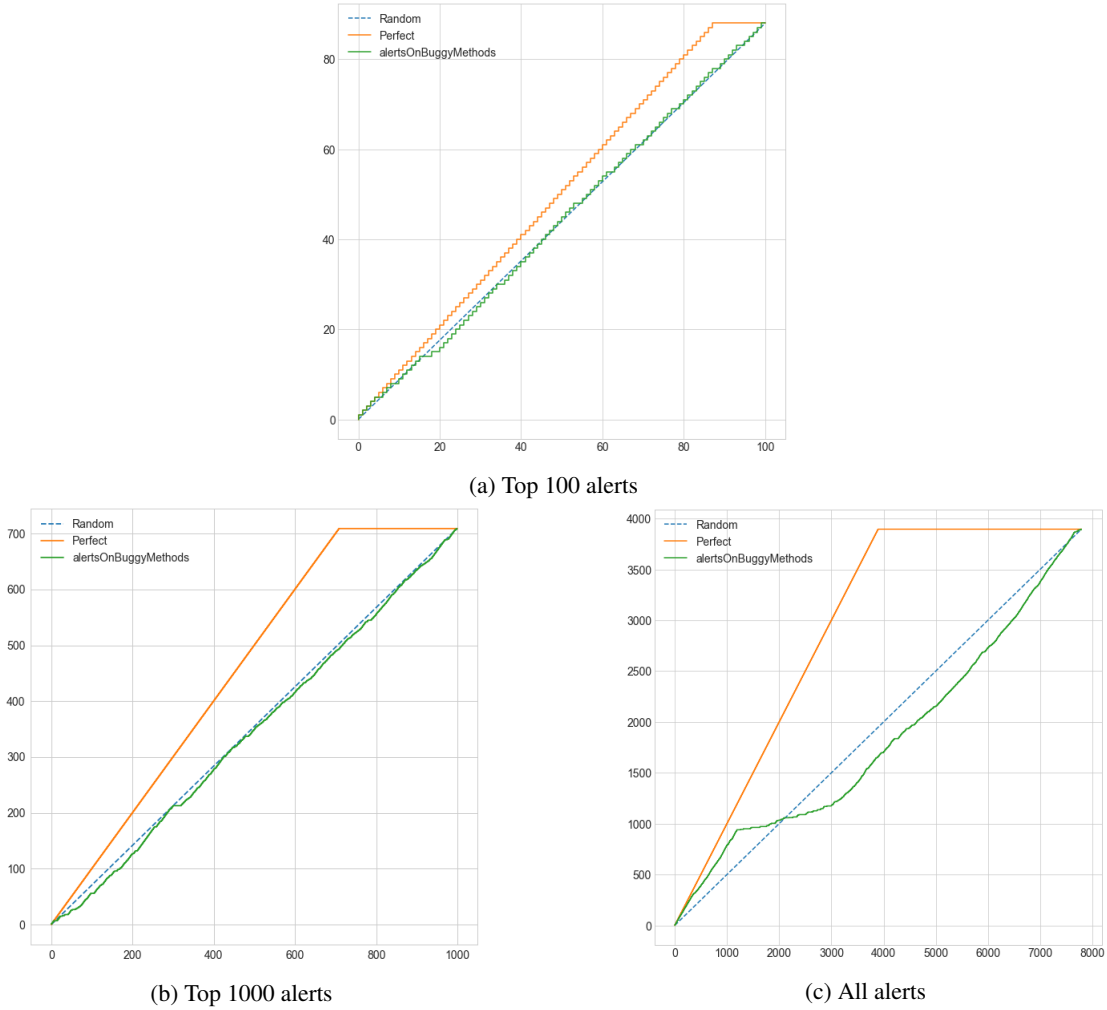


(b) Type correlations

Figure 31: Correlation of buggy methods with closed alerts

Table 9: Comparison against random ranking for the Method Bug Prediction approach

	S(R) metric	**Average FP to TP**	**Number Actionable**
Better than random Top 100 alerts	1000 out of 1000 runs	1000 out of 1000	75% more than random ranking
Better than random Top 1000 alerts	1000 out of 1000 runs	1000 out of 1000	57% more than random ranking
Better than random All alerts	0 out of 10 runs	9 out of 10	



(a) Top 100 alerts

(b) Top 1000 alerts

(c) All alerts

Figure 32: Cumulative ranking graph for Method Bug Prediction

6.5 Feedback Rank

(*) These experiments were run on a small subset of the dataset. The Bayes Network was trained on 10000 samples and tested on 1000. The reason being the abnormally high runtime (one hour to predict 1000 warnings). That could be explained by the size of the network, given that our project has hundreds of packages and thousands of files and methods.

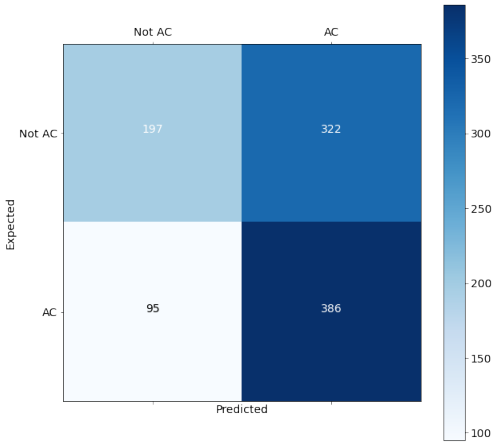
Different balancing algorithms were applied to the training set and *Bayes Network* was used as a classifier (table 10). Random over sampling seems to perform the best, but the overall results are rather poor (fig. 33).

Alerts were ranked according to the probability produced by the Bayes Network. Based on the top 100 results, Feedback Rank performs even worse than random ranking, while doing better on the whole 1000 alert set (table 11, fig. 34).

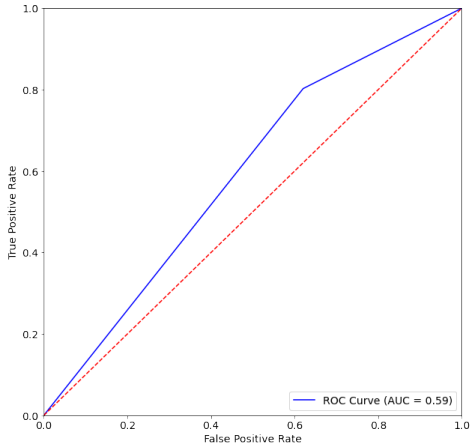
Table 10: Results of different balancing methods for the Feedback Rank approach

	Precision	**Recall**	**Specificity**	**F1-Score**	**Geometric**	**IBA**
Random Under Sampler	57%	55%	57%	52%	51%	26%
Random Over Sampler	**61%**	**58%**	**60%**	**56%**	**55%**	**30%**
None	59%	56%	58%	53%	51%	26%

Calculated using the balanced classification report of imblearn [32]



(a) Confusion matrix of test set for unbalanced training set

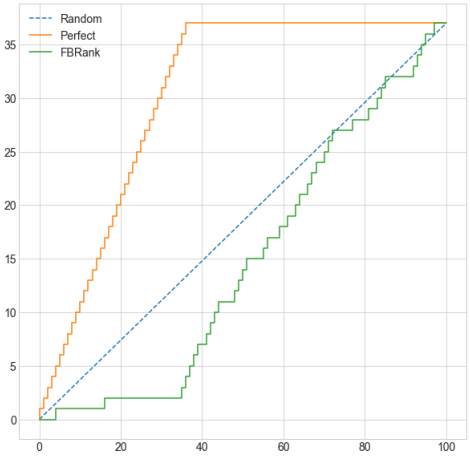


(b) ROC Curve

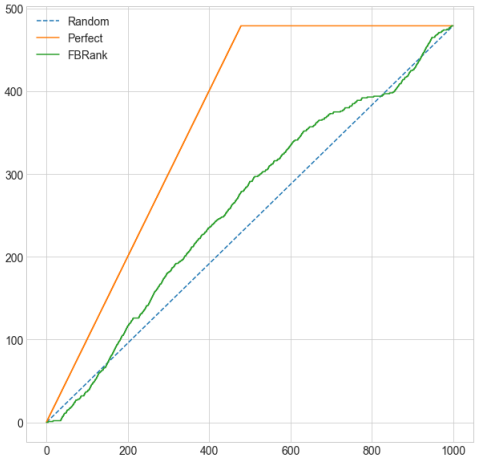
Figure 33: Feedback Rank with Random Over Sampling

Table 11: Comparison against random ranking for the Feedback Rank approach

	S(R) metric	**Average FP to TP**	**Number Actionable**
Better than random Top 100 alerts	0 out of 1000 runs	85 out of 1000	13% less than random ranking
Better than random Top 1000 alerts	1000 out of 1000 runs	869 out of 1000	



(a) Top 100 alerts



(b) Top 1000 alerts

Figure 34: Cumulative ranking graph for Feedback Rank

6.6 Bug-Related Lines

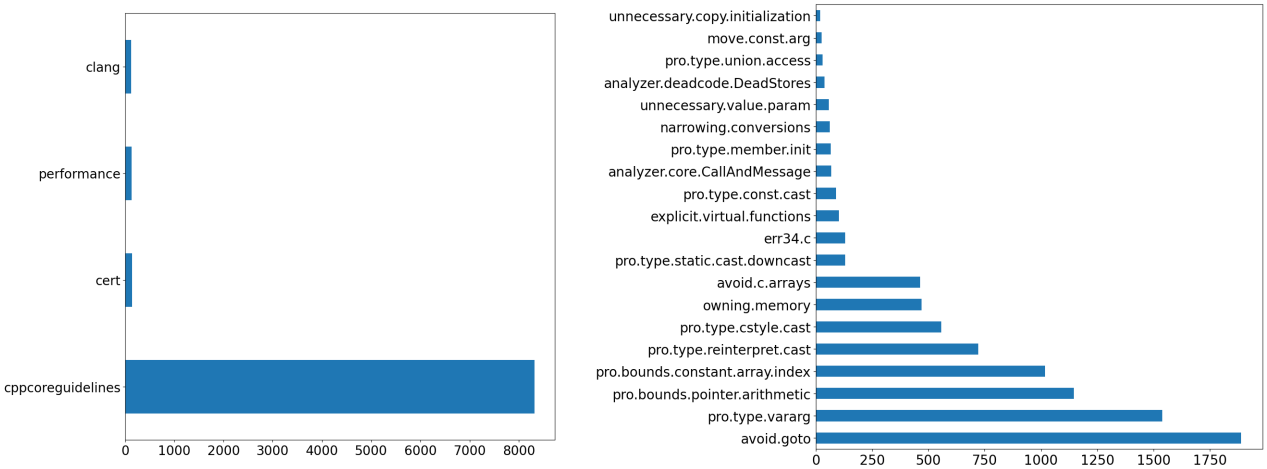
By analyzing the bug-fix revision we are able to collect around 8700 bug-related alerts (alerts pointing to a bug-related line or method). The type and category distribution of these alerts can be seen on fig. 35

Since this method only calculates weights for alert types, it does not predict if a particular alert is actionable or not. As a consequence, only the ranking is evaluated and not the other metrics.

In contrast to previous methods, the train and test set was split equally (gave better results).

The algorithm ranks alerts types based on two input weight parameters (α, β). Based on experimental evaluation the following respective values were chosen: $\alpha = 0.7, \beta = 0.3$. The particular value of these variables did not make much difference on the experiment results, rather than if of them is bigger than the other.

As can be seen from table 12 and fig. 36 the algorithms performs well both on the top 100 and 1000 alerts.



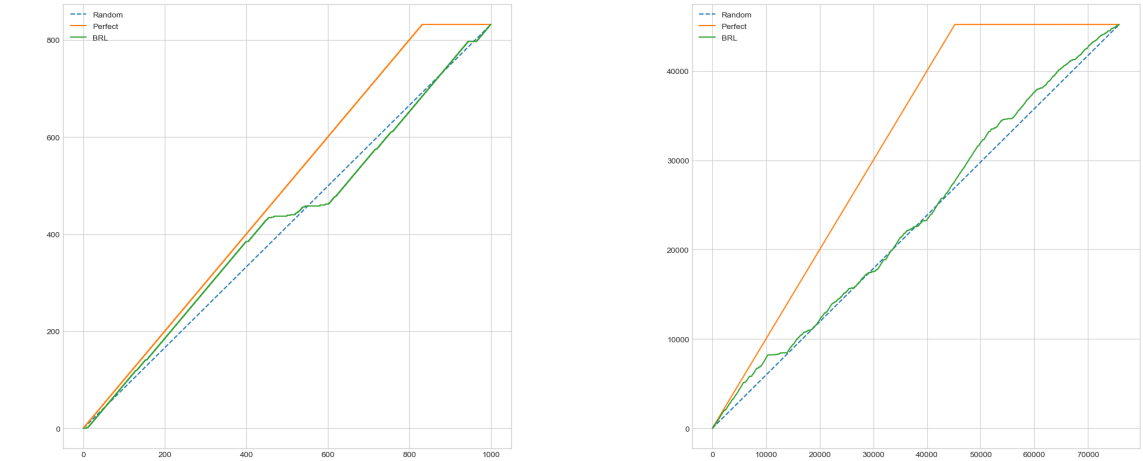
(a) Bug-related alert category distribution

(b) Most important bug-related alert types

Figure 35: Bug-related alert distribution

Table 12: Comparison against random ranking for the Bug-Related lines approach

	S(R) metric	**Average FP to TP**	**Number Actionable**
Better than random Top 100 alerts	1000 out of 1000 runs	1000 out of 1000	50% more than random
Better than random Top 1000 alerts	1000 out of 1000 runs	1000 out of 1000	39% more than random
Better than random All alerts	10 out of 10 runs	10 out of 10	



(a) Top 1000 alerts

(b) All alerts

Figure 36: Cumulative ranking graph for Bug-Related Lines

6.7 Combined approach

One of the goals of this thesis was to explore the possibility of combing different methods that cover each others weaknesses in order to achieve a better ranking. The four evaluated approaches can be split in two categories: those that try to detect actionable alerts and rank them higher (section 5.1, section 5.2) and those that try to promote alerts that can potentially discover/prevent bugs (section 5.3, section 5.4).

By analyzing the performance of the Actionable Alerts method (section 5.2), we see that there is not much room for improvement, since it is able to achieve great results for all four alert categories (see table 13).

Table 13: Classification metrics of Actionable Alerts method for individual categories

	Precision	**Recall**	**Specificity**	**F1-Score**	**Geometric**	**IBA**
cppcoreguidelines	97%	97%	98%	97%	98%	95%
performance	98%	98%	99%	98%	99%	97%
clang	98%	98%	98%	98%	98%	96%
cert	99%	99%	98%	99%	99%	97%

On the other side, if we look at the number of collected bug-related alerts (those that are located on bug-related methods on the first analyzed revision), we notice that it not even 5% of the total amount of alerts. Also almost all of them disappear with time, and can thus be considered actionable. There is in this case an big overlap in the training set of both the Actionable Alerts and Bug-Related Lines method, and since the first performs quite well, a combination is redundant.

Also there is no clear correlation between a method being buggy and the actionable alerts that it contained (fig. 31). That is something that can be explained by bugs belonging to two categories: generic bug patterns (that can be discovered by static analysis) and domain specific bugs (difficult to detect by static analysis), as described by Habib et al. [28].

Nevertheless, acting on SA alerts will improve code quality which may lead to a decreased manifestation of future bugs in the methods that are considered as bug-prone.

Given the aforementioned considerations, a logical combination is a weighted Actionable Alerts method. The probability of an alert being actionable can be weighted by the probability of the method where it is located being buggy. That can also be augmented with relative weights for the alert types of bug-related methods (calculated by diving the number of alerts for each type by the total number of bug-related alerts).

The three inputs used in the combined approach are:

- Probability of an alert being actionable.
- Probability of the method, where the alert is located, being buggy.
- Weights of alert types calculated as: $\frac{\#alerts\,of\,type_X}{\#\,of\,bugRelated\,alerts}$ for each alert type.

The problem of finding an optimal approach to such a combination can be defined as a *multi-objective optimization problem*. We have the following ensemble ranking algorithm for alerts:

$$Alert_{score} = w_1 * Actionable + w_2 * Buggy + w_3 * AlertType$$

We want to find the optimal weights such as: we have the highest possible amount of actionable alerts at the top *X* places, that also point to bug-prone methods.

By merging the different dataset, we can construct an ensemble test set for evaluating our approach. On this set, we have around 370 actionable alerts that also point to buggy methods. Since our classifiers are not perfect (especially the bug prediction one), we cannot expect to have all of them on the top of the ranked list.

We define the *objectives* of the problem as:

- Optimize the number of actionable alerts in the first 200 places.
- Optimize the number of alerts that point to bug-prone methods in the first 200 places.

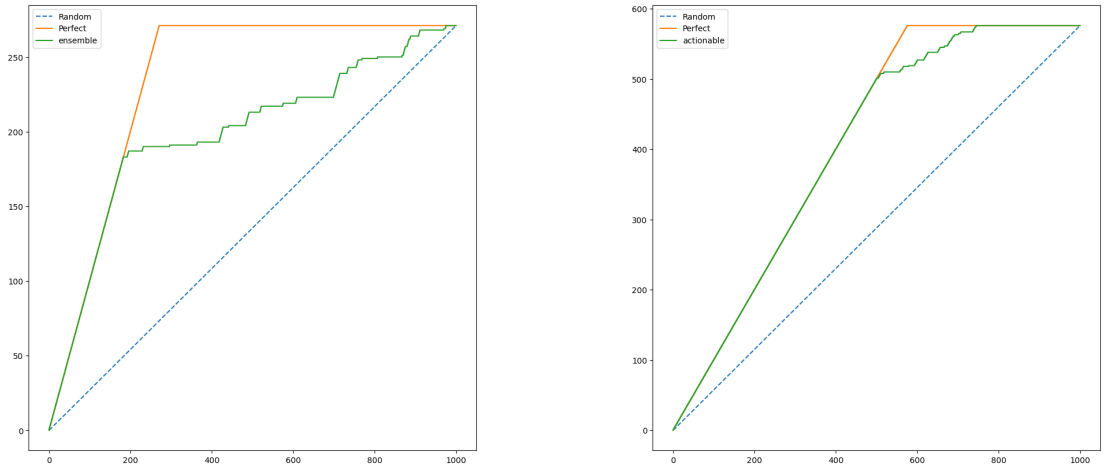
We can also define *constraints*, such as: there are at least 180 actionable alerts in the top 200 places.

To solve the problem we use the python *Platypus* library [37].

As we can see from the results in table 14 and fig. 37, we have an improvement in the ensemble method, regarding the number of bug-prone alerts in the top 200 and 1000 places. That does come with a cost, as beyond those first 200 places, the performance starts to heavily degrade (fig. 37). That should not be a big problem, since in practice it is rare to inspect that many alerts in one time. Also, with some fine tuning of the optimization problem, better results can be achieved.

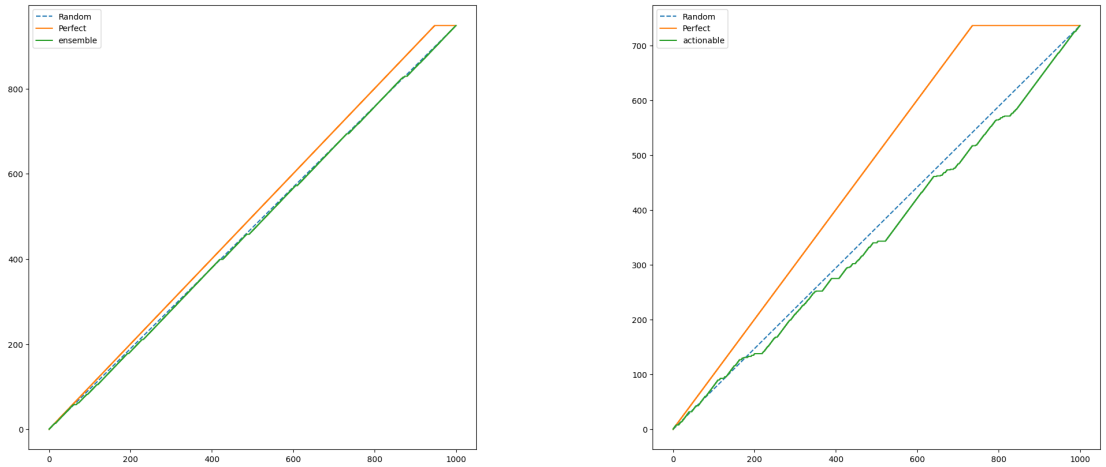
Table 14: Actionable and bug-prone alerts in ensemble ranking

	Actionable Ensemble	**Actionable Normal**	**Bug-prone Ensemble**	**Bug-prone Normal**
Top 200 alerts	185	200	187	137
Top 1000 alerts	265	1000	948	736



(a) Ensemble method (b) Actionable Alerts method

Figure 37: Cumulative ranking graph for first 1000 actionable alerts



(a) Ensemble method (b) Actionable Alerts method

Figure 38: Cumulative ranking graph for first 1000 bug-prone alerts

7 Threats to Validity

7.1 Construct validity

As already mentioned in section 4.4, the automatic nature of data collection poses a threat to validity. The collected dataset may contain an inflated amount of positive examples (either actionable alerts, bug related lines, or buggy methods). That is something that we cannot fully mitigate, but still, try to limit the risks by applying data cleaning approaches where applicable.

7.2 Conclusion validity

By using a mix of different metrics, that relate to classification and ranking, we can confidently make claims on the effect of ranking methods. First, we use metrics suitable for unbalanced data, so that the results of the majority class (unactionable alerts) does not overwhelm the results and mask classification problems for the minority class (actionable alerts). Second, we compare the ranked output of our methods with a random ranking to see if there is an improvement. Third, we train the algorithms on a balanced dataset, as to avoid potential bias.

7.3 Internal validity

By following the suggestion of Pascarella et al. [2], we use a horizontal train/test strategy, meaning that the train and test sets are extracted by sequentially dividing the sorted dataset (in order of alert appearance in time) in two parts. That way we avoid using dependent variables that are not available at prediction time on a real world scenario.

Regarding the Feedback Rank approach (section 5.1), the terrible time performance of the implemented version did not allow us to train the algorithm will full data. Its experimental results can only be seen as a general indication, but do not fully evaluate the capabilities of that method.

7.4 External validity

The scope of this thesis, conducted as an internship in a company, is to research the effectiveness of alert ranking techniques in a particular industrial environment. We do not test the generalizability of our results outside the context on which the thesis was conducted, and as a consequence, we make no claims on the behavior or performance in an external context.

8 Conclusions

8.1 Conclusions

8.1.1 RQ#1: Can we apply SA ranking techniques in an industrial environment with limited amount of data?

Although a dataset of around 190k collected total alerts (of which around 80k are actionable), seems largely adequate at first sight, it should be noted that more than 93% of them belong to one of the four alert categories, *cppcoreguidelines*, for the total number of alerts, as well as the actionable ones (fig. 20). Those are mostly stylistic checks, which are abundant in nature and easy to disappear with code changes. The remaining three categories contain a total of around 12k alerts, of which 3.5k are closed.

Nevertheless, the scores for each individual category are also good (table 13), so the dominating one does not inflate the results for the rest.

Given the overall results of the best performing method, Actionable Alerts (section 5.2), we can say that applying SA ranking techniques is possible in this industrial context, even considering the fact that SA tools are not used anymore.

8.1.2 RQ#2: Can we combine SA ranking techniques to achieve better results?

The performance of the Actionable Alerts method (section 6.3.1) is difficult to improve. What we can improve by combining multiple methods, is not only ranking alerts that are actionable, but also by directing the attention to those that point to bug-prone methods (can have a higher utility by preventing bugs). In that aspect, we do achieve better results in the top 200 places in the ranking (section 6.7).

8.1.3 RQ#3: Do pre-processing techniques provide a significant performance benefit?

By inspecting the metrics achieved by different balancing/cleaning techniques (table 6 and table 8) we can see that they do make a difference. Although, in the Actionable Alerts method, the improvement is small, in the Method Bug Prediction approach, the improvement is more noticeable.

8.2 Summary of Contributions

This thesis provides the following contributions:

- The evaluation of SA ranking techniques on an industrial codebase.

  Henrique: I think you are missing a little more detail to make this into a proper contribution.

  Kleidi: For example? I have results for each tool.

- Comparison of pre-processing methods to deal with imbalanced and noisy data.
- We conducted a comparative study of different approaches on a common codebase.

8.3 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 descriptions to have a clearer connection between a bug and an alert.

Given also the limited amount of (potentially) noisy data, the cleaning aspect and be explored even further (fine tuning techniques). The impact of pre-processing should also be considered, for example how to best deal with high-cardinality data (which is the case many features in out dataset).

Furthermore, there is a lot of room for fine tuning the optimization problem in the ensemble method. Its definition, the score function, the optimization objectives, can be analyzed even further. Also, the performance of Method Bug Prediction, can be a bottleneck for this approach, so other methods which may give better results can be tried ([24, 25]).

Last, a way to continuously improve the ranking algorithms needs to be put in place. If newly inspected alerts are tracked consistently (TP or FP), a more qualitative dataset can be collected and performance will also improve (a rather naive implementation is to give each warnings an identifier and include them in the commit messages if they were useful).

References

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

[2] 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.

[3] Paul Clarke, Rory V. O'Connor, and Brian Leavy. A complexity theory viewpoint on the software development process and situational context. In *Proceedings of the International Conference on Software and Systems Process*, ICSSP '16, page 86–90, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 1st edition edition, march 2020.

[5] Maurice Dawson, Darrell Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3:49–53, 01 2010.

[6] 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.

[7] Infer static analyzer.

[8] Sarah Heckman and Laurie Williams. A comparative evaluation of static analysis actionable alert identification techniques. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, PROMISE '13, New York, NY, USA, 2013. Association for Computing Machinery.

[9] 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.

[10] 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.

[11] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 672–681. IEEE Press, 2013.

[12] S. Allier, N. Anquetil, A. Hora, and S. Ducasse. A framework to compare alert ranking algorithms. In *2012 19th Working Conference on Reverse Engineering*, pages 277–285, 2012.

[13] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490, 2011.

[14] 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.

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

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

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

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

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

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

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

[26] 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.

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

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

[29] Clang AST.

[30] Clang Tidy.

[31] Gustavo Batista, Ronaldo Prati, and Maria-Carolina Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explorations*, 6:20–29, 06 2004.

[32] 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.

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

[34] 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.

[35] C. Ferri, J. Hernández-Orallo, and R. Modroiu. An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27 – 38, 2009.

[36] Vicente García, R. Mollineda, and José Sánchez. Index of balanced accuracy: A performance measure for skewed class distributions. volume 5524, pages 441–448, 06 2009.

[37] Platypus - Multiobjective Optimization in Python.