# RANKING THE OUTPUT OF STATIC ANALYSIS

**Kleidi Ismailaj**

*kleidi.ismailaj@student.uantwerpen.be*

August 13, 2020

## Contents

## Abstract

Henrique: It is very rare to have a reference in an Abstract. We should be careful and verify if it is really necessary.

Kleidi: It is a statement made by most of the papers regarding the subject. I could remove the citations.

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

Henrique: Is the SA used for maintenance, development, or both? If it is maintenance, we should shift the initial paragraph wih a little motivation on maintenance. If it for both, we should focus on development but also mention maintenance activities.

Kleidi: I know that OMP wants to use it in a CI environment, so in development

Henrique: I would argue that CI is both development and maintenance.

Software development and maintenance is a complex activity [4]. 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 [5].

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 [3]. 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 [1] or Facebook with its Infer static analyzer [2].

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 [1].

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 [6, 7]. 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 [8, 9]. 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 [10]. 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

Henrique: To me, a problem statement section should appear as earlier as possible in the thesis. It can even be a subsection of the introduction.

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

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

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

The research questions can be formulated as follows:

- R0: Can we apply SA ranking techniques in an industrial environment with limited amount of data (abandoned because of high false positives)?
- R1: Can we combine SA ranking techniques to achieve better results?
- R2: 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 [6, 11]. Also, they mainly focus on open source Java systems with an adequate amount of data. Kim et al. [12] 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

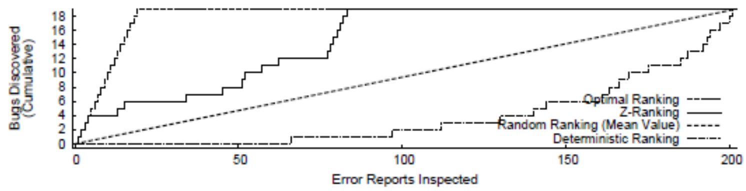
Kremenek and Engler [13] introduce *Z-ranking*, a statistical model to rank the error reports of SA tools. They make a distinction between successful and failed checks (those that satisfy a checked property and those that violate it). The underlying observation is that the most reliable error reports 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.

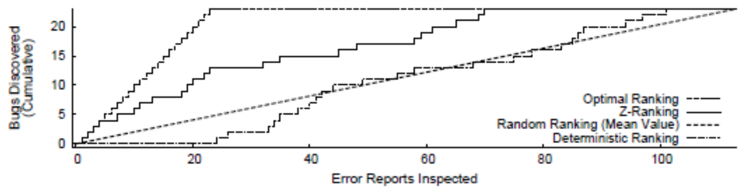
Given a grouping operator *G* that groups successful and failed checks together, we calculate the proportion of failed checks $G.\rho = \frac{G.successful}{G.failed}$. Populations are ranked both by the $\rho$ value and by the degree of confidence in its estimation. By treating these checks inside the 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)

Kremenek et al. [14] 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.

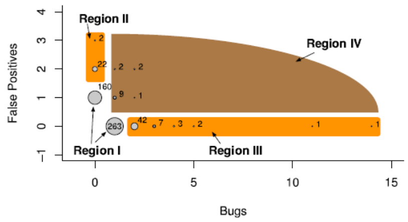
An intuitive explanation for the reason that reports cluster is given for both real and false positives. Regarding true positives, when developers do not know a rule, they will repeatedly violate it, so errors of the same type will correlate together. As for the false positives, there are three main 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.

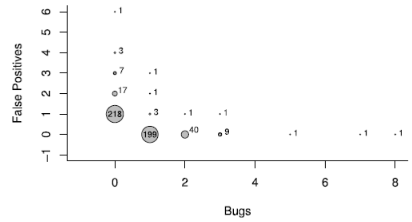
To apply the algorithm a model is needed that produces the correlations among the reports. The reports are divided into two major regions, one that contains mostly true positive ($g_B$), and one that contains mostly false positives ($g_{FP}$) (see fig. 2b). A *Bayesian Network* is used to calculate the 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 fig. 2c, 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.

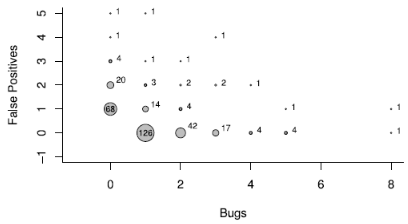
According to their tests, *Feedback-Rank* performed 2-8 times better than randomized ranking.



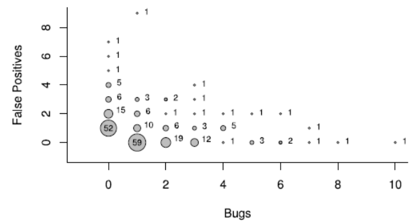

(a) Linux - Grouped by Function



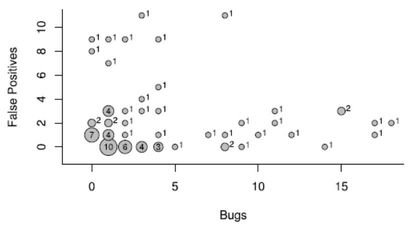

(a) System X - Grouped by Function



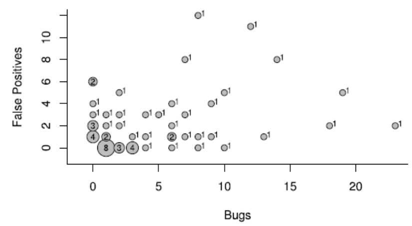

(b) Linux - Grouped by File




(b) System X - Grouped by File

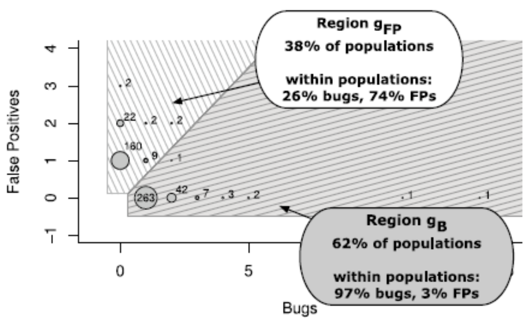



(c) Linux - Grouped by Leaf Directory

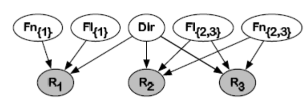



(c) System X - Grouped by Leaf Directory

(a) "L" shape of clustering




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




(c) Sample Bayesian network with 3 reports

Boogerd and Moonen [15] 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.

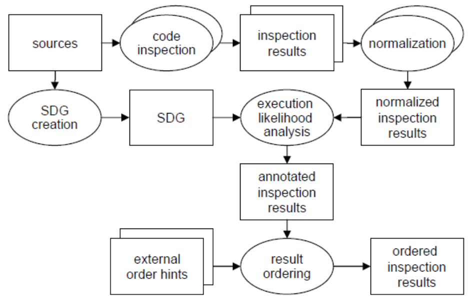



Figure 3: Workflow for the ELAN tool

Kim and Ernst [8] 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

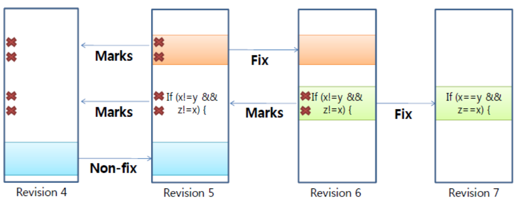
  if $i$ is removed in a non-fix change

    then $w_c = w_c + \beta$

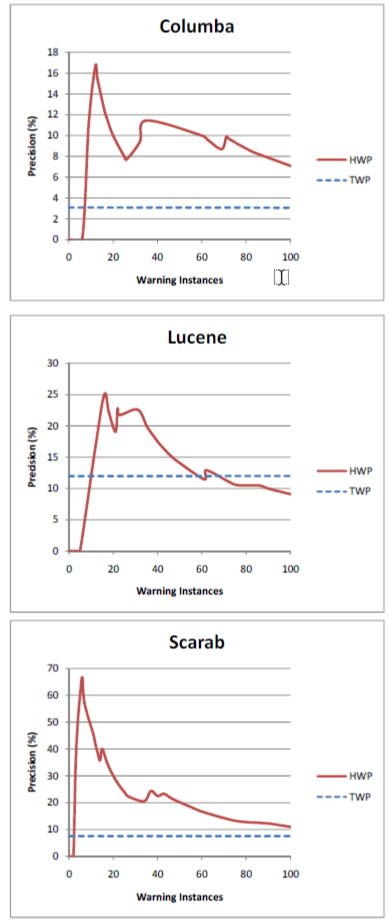
// weight normalization step

$w_c = w_c / |C|$ 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

Ruthruff et al. [7] 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 ($6 \times 3$ 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% |

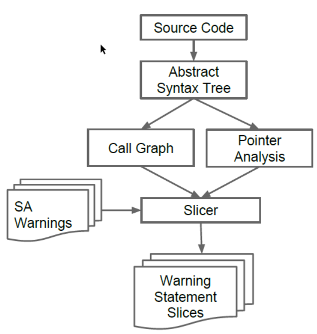
**Table 8: Predicting actionable warnings.**

(b) Results for predicting true positives and actionable warnings

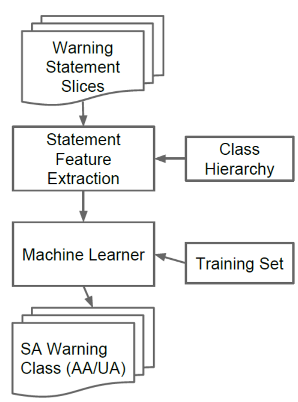
Hanam et al. [16] present a method for differentiating actionable and unactionable alerts by finding alerts with similar code patterns (alerts with similar patterns are probably of the same type). They use a feature vector based on code characteristics at the site of each SA alert along with a classifier to build a model for predicting AA. They introduce the notion of *alert patterns*, source code patterns employed by developers that are unactionable but are repeatedly flagged by SA tools (or similarly always actionable).

To extract features from the site of the warning (and near it), lightweight program slicing is used. Backwards slicing is used to detect which statements could have affected the outcome of the seed statement (place of the alert). To speed up the slicing process, all external classes are excluded from the analysis and the depth is limited to the 5 nearest statements prior to the seed.

By using the source code history of three projects to train and test their approach, they achieve considerably better results than the default ranking of a SA tool (57 vs 19 AA in the top 20% of the alert list), and a slight improvement (6%) than the existing techniques.




(a) Method for generating slices




(b) Workflow for classifying alerts

Venkatasubramanyam and Gupta [17] propose an incremental and lightweight approach to detect coding violations by using a learning system. They track warnings through the version history to detect patterns and determine which SA rules are important (and must be enabled). Their approach focuses on differential code analysis (filter/identify violations happening only on new parts of code), and on learning from the experts.

Their methodology (fig. 7) consists of database that continuously stores information about SA rules. The initial version is build by mining (at least) the last three version of the software under analysis. To train the classifier (learning system) different features are used: patterns of code where SA violations are reported, impact of the violations on code quality, confidence level of the rules (probability that a rule gives a false positive), most commonly committed errors (reflecting the developers pattern of coding) and the most recently committed errors.

New code changes made by developers are checked against the database. By using a patterns matching algorithm that compares new code with the patterns saved in the database, possible bugs can be detected. This approach potentially permits to run the SA tools less frequently, since code violations can be suggested by comparing against past saved patterns. They also suggest using *first order logic* for capturing the context around rule violations and thus learning what factors produce a false/true positive.

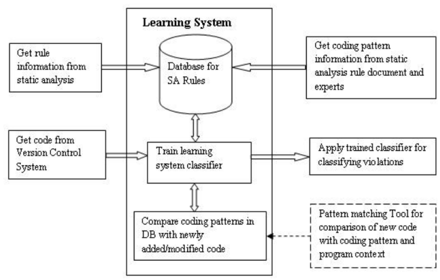



Figure 7: Workflow for incremental violation detection

Heckman and Williams [18] 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. 8) 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.

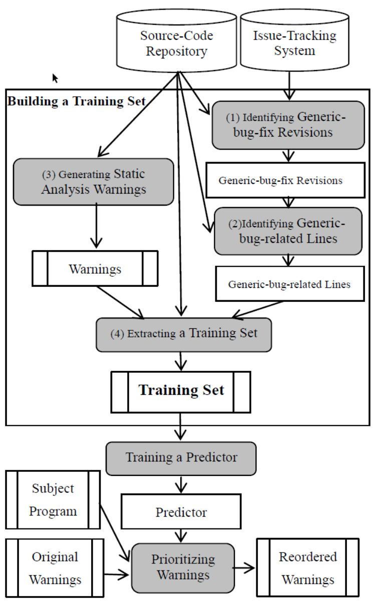
| Project | Average Precision | Average Recall | Average Accuracy |
|---|---|---|---|
| `jdom` | 89.0% | 83.0% | 87.8% |
| `runtime` | 98.0% | 99.0% | 96.8% |

Figure 8: Average metrics for all models

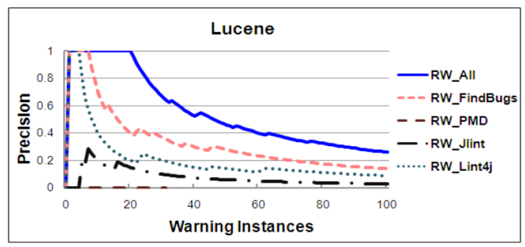
Liang et al. [9] propose an automatic approach (fig. 9a) for constructing an effective training set for warning prioritization algorithms. They introduce the notion of "generic-bug-fix revisions" vs. "project-specific-bug revisions", which differentiate bug-fixing lines depending on the sort of bug that 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.

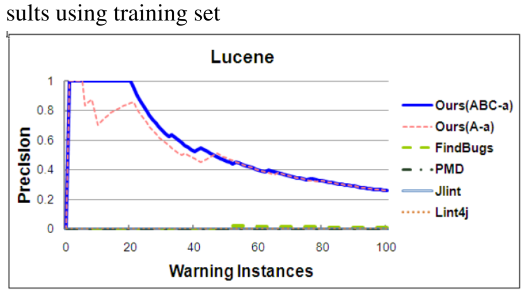
Using a *K-Nearest Neighbor* classifier (trained on the first half of the revisions) paired with a feature selection algorithm, they achieve significantly better result than the tool output, especially in the first 20 warnings range. They found that using multiple SA tools gives a better result than single tool models (fig. 9b). 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. 9c). 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

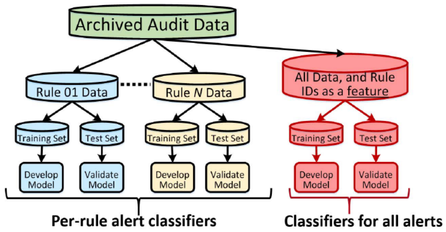



(c) Intra (A-a) vs inter (ABC-a) project training

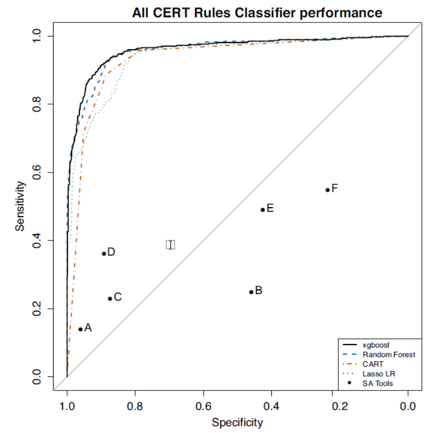
### 3.1.2 Multiple Tools

Flynn et al. [19] 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

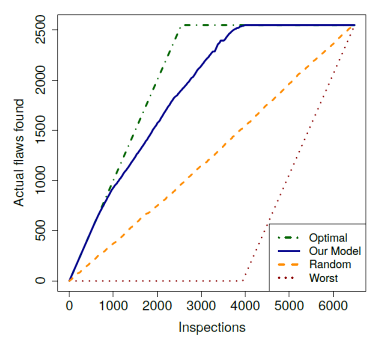
Ribeiro et al. [20] aim to reduce the false positive rate of an ensemble of static analyzers by using *Decision Trees* and *AdaBoost*. The goal is to make possible to combine the strengths of different analyzers without suffering too much from false positives. Their approach ignores source code characteristics, 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. 11a 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

### 3.1.3 Literature review papers

Heckman and Williams [21] 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.

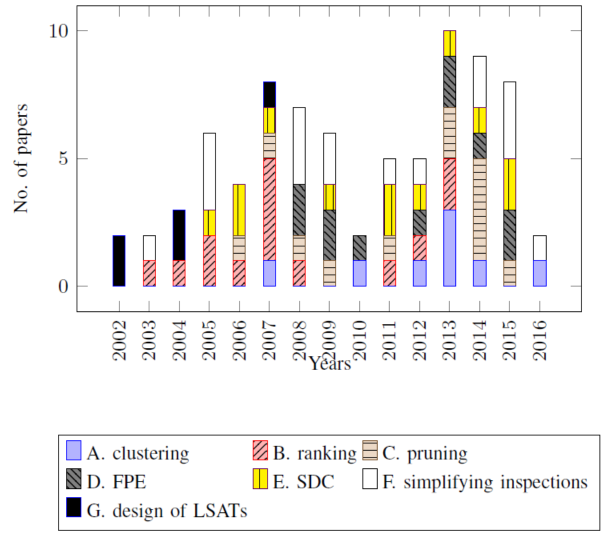
Approaches followed by AAIT's fall into seven main categories: (a) alert type selection (selecting altert types that are the most relevant for a codebase), (b) contextual information (limiting SA tools only to parts of code where that they can analyze well), (c) data fusion (combining multiple SA tools), (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..

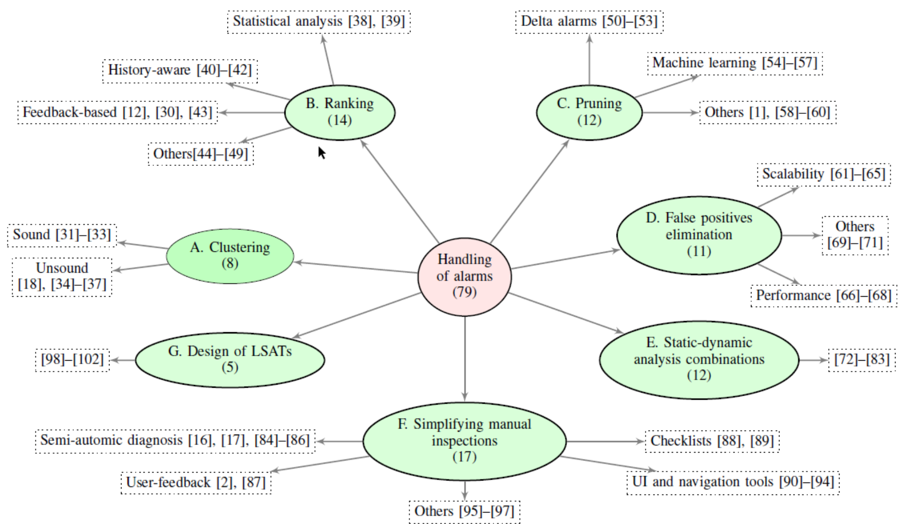
Muske and Serebrenik [22] perform a systematic review of SA alarm handling techniques (fig. 12a). They define *handling of alarms* as: (a) post-processing to reducing the manual inspection effort (using correlation, clustering, ranking...), and (b) supporting manual inspection of alarms.

Seven categories for identifying alarms are defined (fig. 12b): (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

#### 3.1.4 Comparative studies

Heckman and Williams [6] 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. [11] 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.

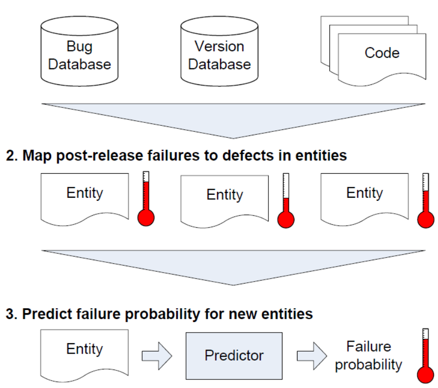
They found out that Aware and FeedbackRank perform significantly better than the other ranking approaches. In addition individual alert raking algorithms performed better than those that rank alert types. Also they did not find a clear distinction in performance between statistical and ad-hoc approaches.

### 3.2 Bug Prediction

Nagappan et al. [23] 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.




(a) Workflow for predicting defects on new entities

| Metric | Description |
|---|---|
| **Module metrics** — correlation with metric in a module *M* | |
| *Classes* | # Classes in *M* |
| *Function* | # Functions in *M* |
| *GlobalVariables* | # global variables in *M* |
| **Per-function metrics** — correlation with maximum and sum ( | |
| *Lines* | # executable lines in *f*() |
| *Parameters* | # parameters in *f*() |
| *Arcs* | # arcs in *f*()'s control flow graph |
| *Blocks* | # basic blocks in *f*()'s control flow graph |
| *ReadCoupling* | # global variables read in *f*() |
| *WriteCoupling* | # global variables written in *f*() |
| *AddrTakenCoupling* | # global variables whose address is taken in *f*() |
| *ProcCoupling* | # functions that access a global variable written in *f*() |
| *FanIn* | # functions calling *f*() |
| *FanOut* | # functions called by *f*() |
| *Complexity* | McCabe's cyclomatic complexity of *f*() |
| **Per-class metrics** — correlation with maximum and sum of m | |
| *ClassMethods* | # methods in *C* (private / public / protected) |
| *InheritanceDepth* | # of superclasses of *C* |
| *ClassCoupling* | # of classes coupled with *C* (e.g. as attribute / parameter / return types) |
| *SubClasses* | # of direct subclasses of *C* |

(b) The set of complexity metrics considered

Giger et al. [24] 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. 14a) 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. 14c).

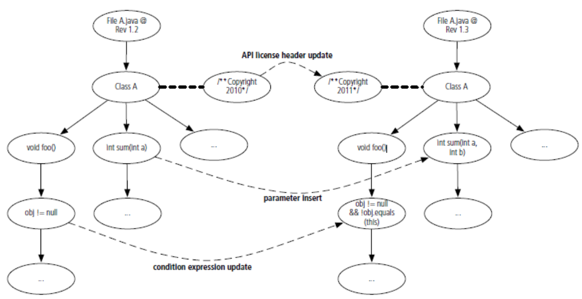
By using an extensive test set of multiple open source Java projects and by labeling each method as bug-prone or not bug-prone (using historical VCS data) they were able to measure the efficacy of different classifiers. The classifiers were trained with both source and change metrics and each separately. 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. 14b).

| 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 |

(a) 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 |

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




(c) Fine grained code changes extracted from AST comparisons

Wang et al. [25] 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. 15a).

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. 15b 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. As can be seen from fig. 15c, the DL approach outperforms the traditional methods, with an average improvement in precision and recall of 14% and 11% respectively.

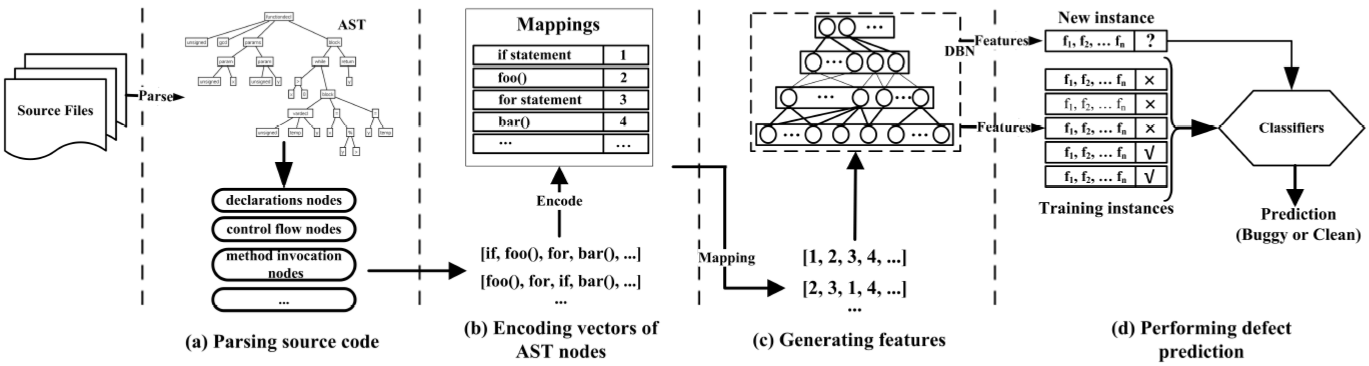
```java
int i = 9;
if (i == 9) {
  foo();
  for (i = 0; i < 10;
       i++) {
    bar();
  }
}
```

File1.java

```java
int i = 9;
foo();
for (i = 0; i < 10; i
     ++) {
  if (i == 9) {
    bar();
  }
}
```

File2.java

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




(b) Workflow for the semantic learning process

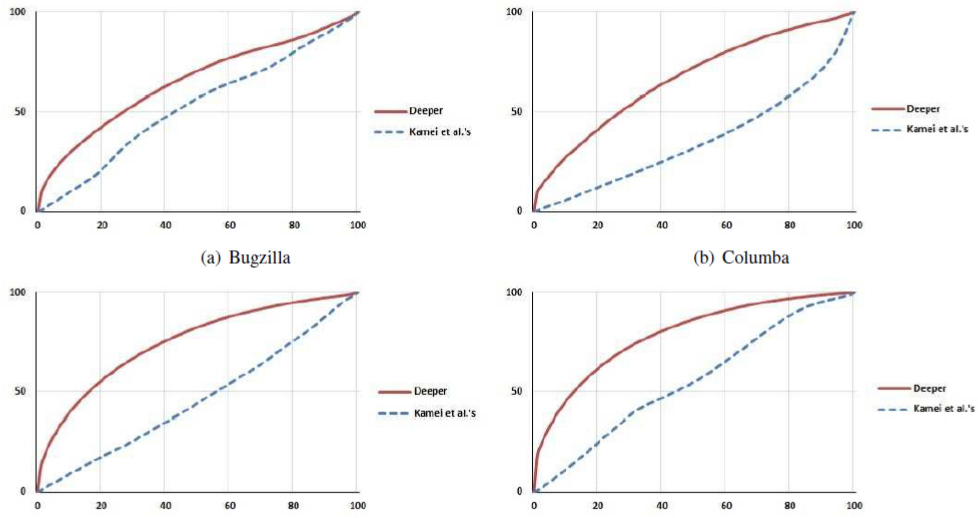
| Project | Versions (Tr->T) | Semantic P R F1 | PROMISE P R F1 | AST P R F1 |
|---|---|---|---|---|
| ant | 1.5->1.6 | 88.0 95.1 **91.4** | 44.8 51.1 47.7 | 40.5 51.4 45.3 |
| | 1.6->1.7 | 98.8 90.1 **94.2** | 41.8 77.1 54.2 | 41.2 54.7 47.0 |
| camel | 1.2->1.4 | 96.0 66.4 **78.5** | 24.8 75.2 37.3 | 32.3 55.6 40.2 |
| | 1.4->1.6 | 26.3 64.9 37.4 | 28.3 63.7 **39.1** | 29.7 51.5 38.3 |
| jEdit | 3.2->4.0 | 46.7 74.7 **57.4** | 44.7 73.3 55.6 | 45.8 47.4 46.6 |
| | 4.0->4.1 | 54.4 70.9 **61.5** | 46.1 67.1 54.6 | 50.4 40.4 44.8 |
| log4j | 1.0->1.1 | 67.5 73.0 **70.1** | 49.1 73.0 58.7 | 55.4 38.6 45.5 |
| lucene | 2.0->2.2 | 75.9 56.9 **65.1** | 73.3 38.2 50.2 | 69.5 37.4 48.4 |
| | 2.2->2.4 | 66.5 92.1 **77.3** | 70.9 52.7 60.5 | 65.9 53.1 58.8 |
| xalan | 2.4->2.5 | 65.0 54.8 **59.5** | 64.7 43.2 51.8 | 60.1 43.5 50.5 |
| xerces | 1.2->1.3 | 40.3 42.0 **41.1** | 16.0 46.4 23.8 | 25.5 22.0 23.6 |
| ivy | 1.4->2.0 | 21.7 90.0 **35.0** | 22.6 60.0 32.9 | 31.6 28.6 30.0 |
| synapse | 1.0->1.1 | 46.0 66.7 **54.4** | 45.5 50.0 47.6 | 51.5 45.7 48.4 |
| | 1.1->1.2 | 57.3 59.3 **58.3** | 51.1 55.8 53.3 | 50.7 40.5 49.0 |
| poi | 1.5->2.5 | 76.1 55.2 **64.0** | 73.7 44.8 55.8 | 70.0 31.6 43.5 |
| | 2.5->3.0 | 81.6 79.0 **80.3** | 75.0 75.8 75.4 | 72.1 46.3 55.6 |
| Average | | 63.0 70.7 **64.1** | 48.3 59.2 49.9 | 49.5 43.0 44.7 |

(c) Precision, recall and F1 score for semantic vs syntactic features

Yang et al. [26] 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 (fig. 16).




(a) Bugzilla (b) Columba

(c) JDT (d) Platform

Figure 16: Improvements over classic logistic regression approach

### 3.3 Static Analysis tools in practice

Beller et al. [27] 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. 17a). Although, most of them run the tools sporadically and without enforcing them.

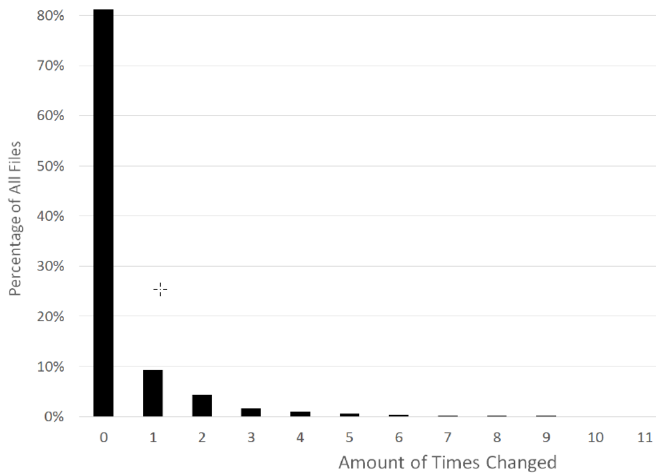
A configuration file of a SA tool, shows what rules developers deem important (enable), and what they do not deem important (disabled, perhaps because of a high false positives rate). The contents of a configuration file are hence an important indicator of how developers use SAs and how well the tool's 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. 17b).

| 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 | 50% | 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

Imtiaz et al. [28] 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. 18a). 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++

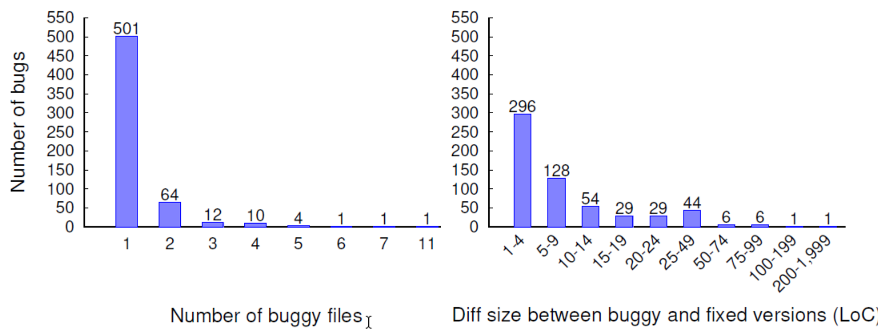
Habib and Pradel [29] study how many of all real-world bugs static bug detectors find. The results of their study show that: (a) static bug detectors find a non-negligible amount of all bugs, (b) different tools are mostly complementary to each other (see fig. 19a), and (c) current bug detectors miss 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. 19b).

| **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

Sadowski et al. [1] 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).

The different techniques that were tested are:

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

## 4.1 Collecting data

Clang AST ([30]) and Clang Tidy ([31]) was used to analyze the code. It was chosen is because it's a reliable open source tool, extensible and most importantly supported by the codebase of the company.

### 4.1.1 Clang AST

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

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

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

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

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

### 4.1.2 Clang Tidy

Clang Tidy is a modular C++ linter tool which provides an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis ([31]).

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

### 4.1.3 Workflow for collecting information

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

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

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

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

- Start by selecting a base revision in the code history.
- Fully analyze that revision of the codebase (collect all alerts output by Clang Tidy).
- Repeat for desired number of iterations (revisions):
  - Checkout next revision, detect changed parts of code, collect surrounding information (author, changed methods, etc...).
  - Run Clang Tidy only on the changed parts of code.
  - Collect any new alerts and metrics.

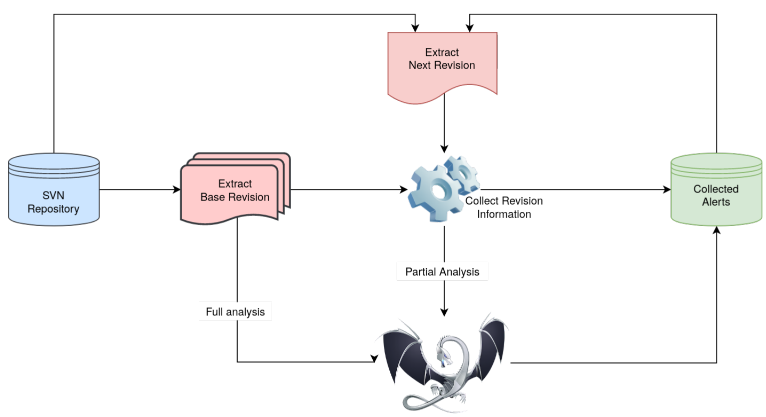



Figure 20: Collecting alerts and other information

#### 4.1.4 Assumptions made for collected data

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

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

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

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

### 4.2 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.

#### 4.2.1 Revision data

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

| | **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 |

| | **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 |

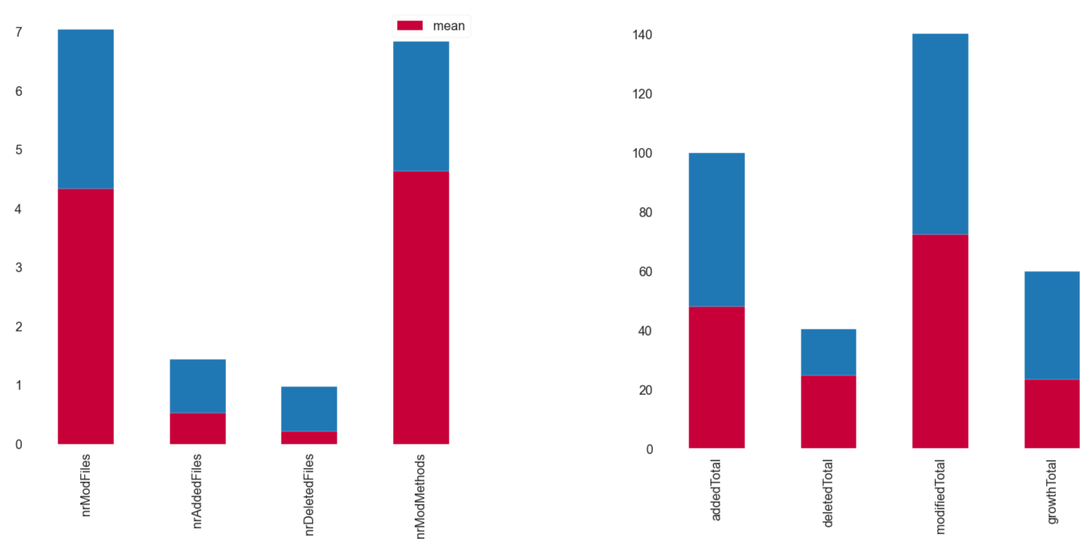
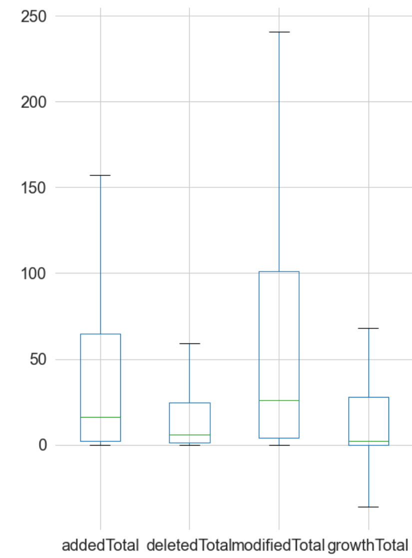
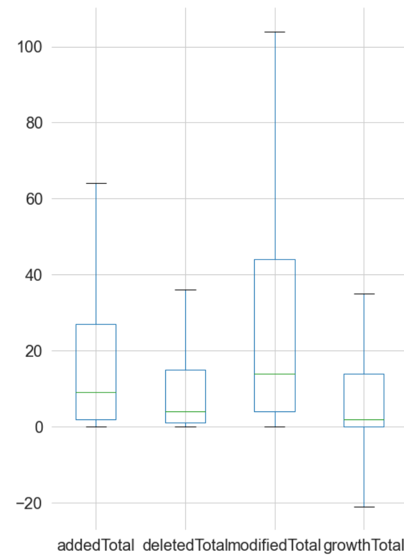



Figure 21: Average changes (*red=bug revision, blue=normal*)

As can be seen from the data, 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 fig. 22). That means that there are certain revision that are abnormally large compared to others and that negatively impact the quality of the data.

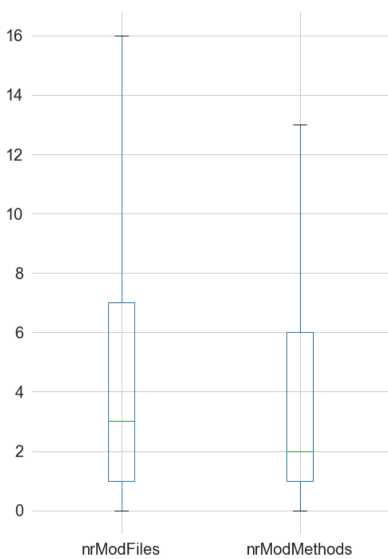



(a) Box plots for changed lines in normal revisions

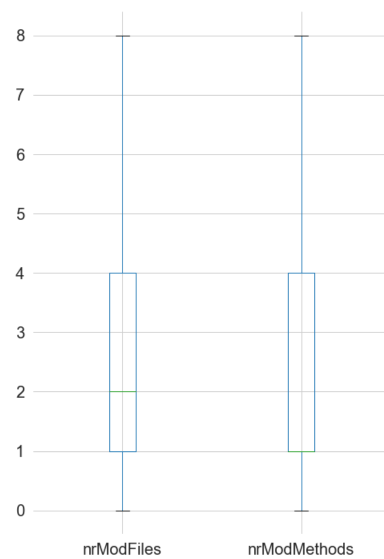



(b) Box plots for changed lines in buggy revisions

Figure 22




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




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

Figure 23

### 4.2.2 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% |

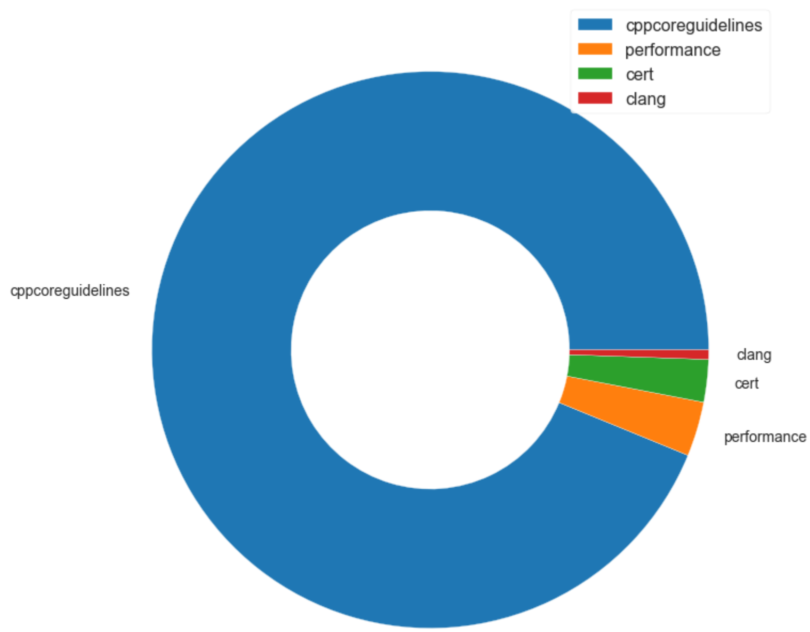



Figure 24: Distribution of collected alerts per alert category

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% |

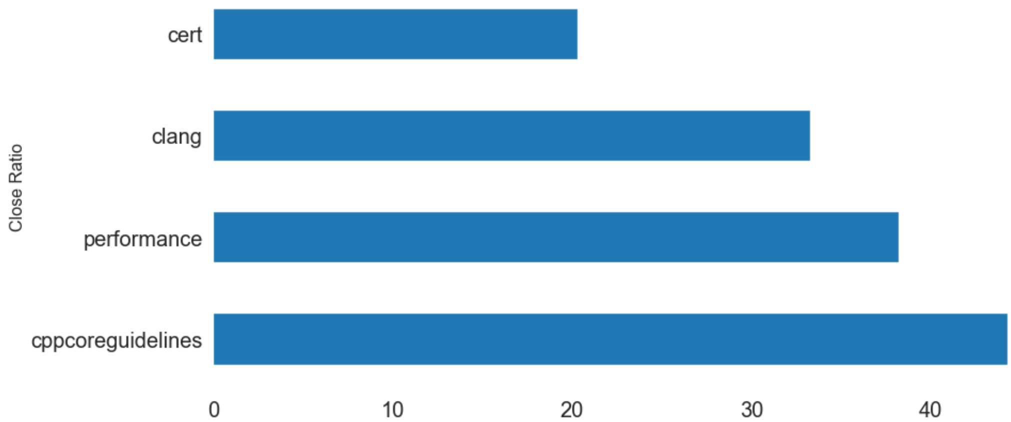
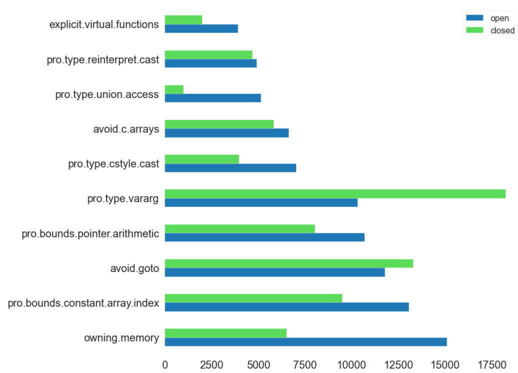


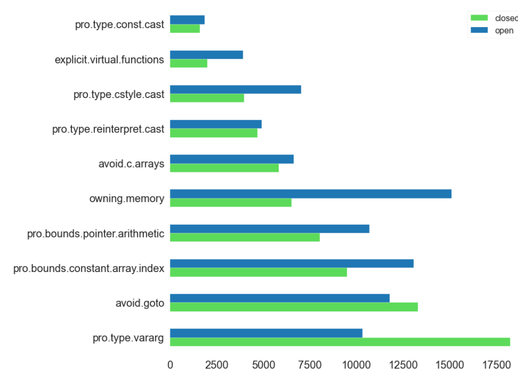

Figure 25: Ratio of closed alerts per category

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. 26). 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.

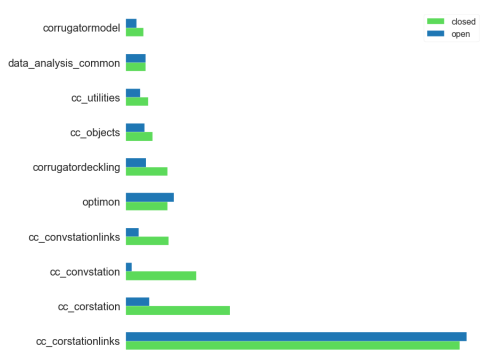



(a) Bar plot for alert types with most open alerts

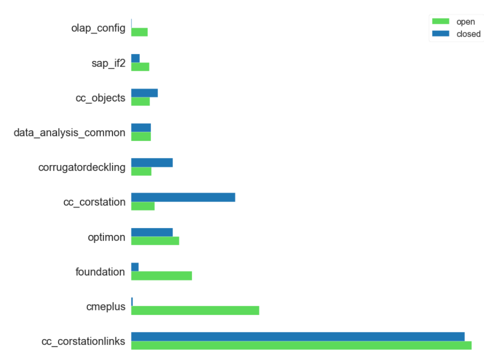



(b) Bar plot for alert types with most closed alerts

Figure 27

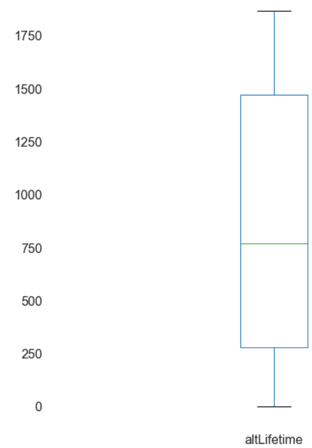



(a) Bar plot for packages with most open alerts

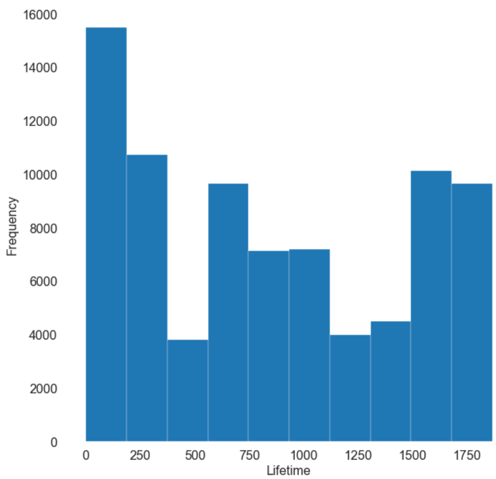



(b) Bar plot for packages with most closed alerts

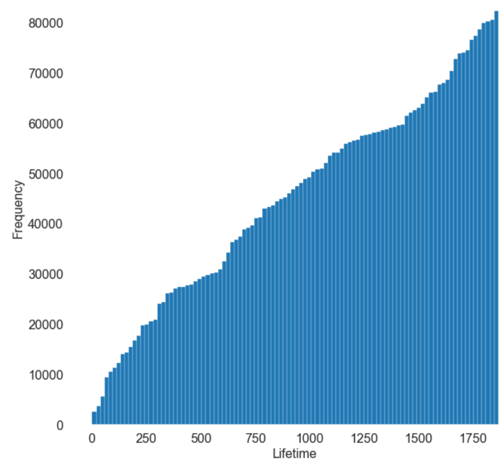
Figure 28




(a) Box plot for closed alert lifetime




(b) Histogram for closed alert lifetime




(c) Cumulative plot for closed alert lifetime

Figure 26

From the plots in fig. 27 and fig. 28, 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.

### 4.3 Dealing with unbalanced and noisy data

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

Kim et al. [12] 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 [12], 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. [32], 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 ([38]) 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

## 5.1 Feedback Rank with Z-Score

### 5.1.1 Feedback Rank

Feedback Rank [14] 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. 29). The initial configuration of the network is learned from historical data (actionable alerts calculated as explained on section 4.1.3). 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.

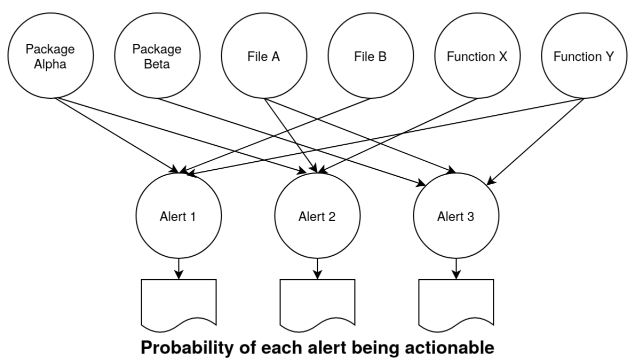



Figure 29: Example of a simple Bayes Network for predicting 3 alerts

According to the Kremenek et al. [14], 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 [13], 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 [7, 18].

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 [18] to conduct our research since it contains an agglomeration of alert characteristics (AC) collected from other papers.

The workflow, as explained in section 4.1.3, consists of iterating through the version history, collecting alerts characteristics and keeping track which alerts disappear (considered as actionable). ACs are then later used as features in ML algorithms with actionability being the target to predict.

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 1.

Table 1: 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 [8], and by Liang et al. [9].

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 30 shows a summarization of the general process we follow to calculate BRLs.

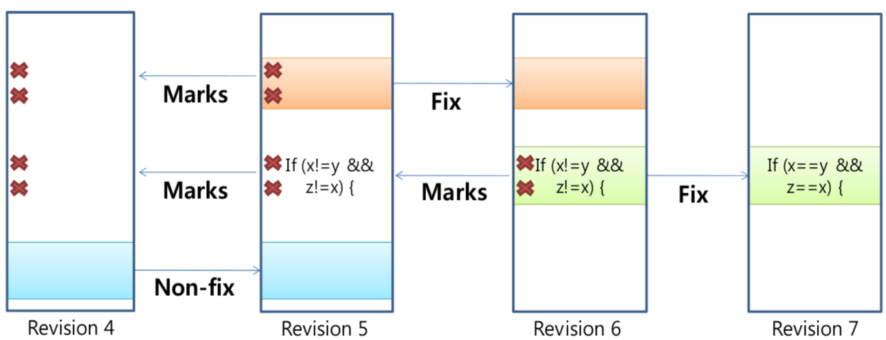



Figure 30: Calculating BRLs ([8])

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 [8], 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, $\alpha$ and $\beta$. 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 [15], 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. [24] 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 2.

**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 2: 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 [35] and as can be seen on fig. 31, 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.

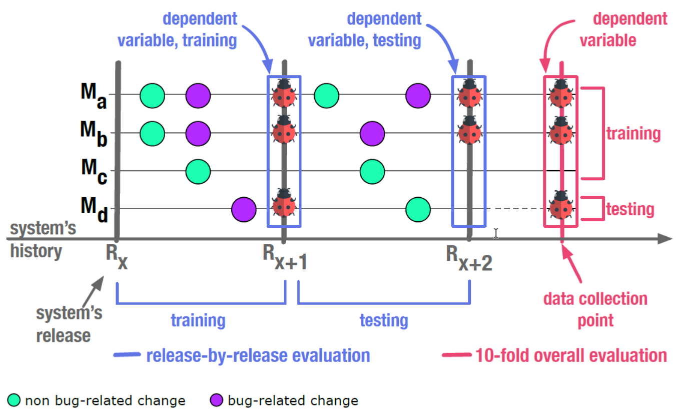



Figure 31: Release-based vs K-Fold ([35])

- **Average FP to TP ([14])**: Let N be the total number of actionable alerts in a set of reports, and $FP_j$ is the cumulative number of false positives to find the *j*th actionable alert.

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

- **S(R) metric ([13])**: 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.

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

- **Random comparison**: 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.
- **Fault detection rate curve**: the curve of a model is 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 ([36], [37]).

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

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.

**TODO: EVALUATE ENCODING METHODS**

### 6.3 Results of individual tools

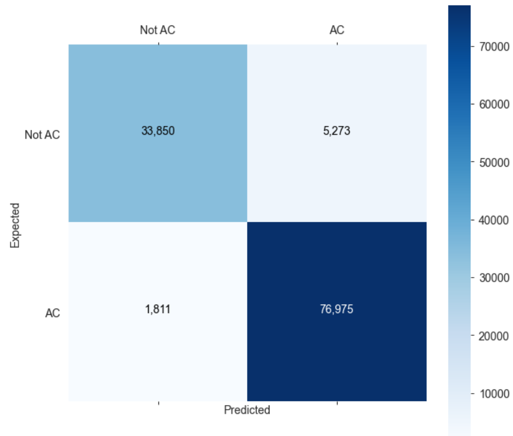
Experiments were run by training methods on the first 80% items of the dataset and testing on the remaining 20%.

#### 6.3.1 Actionable Alerts

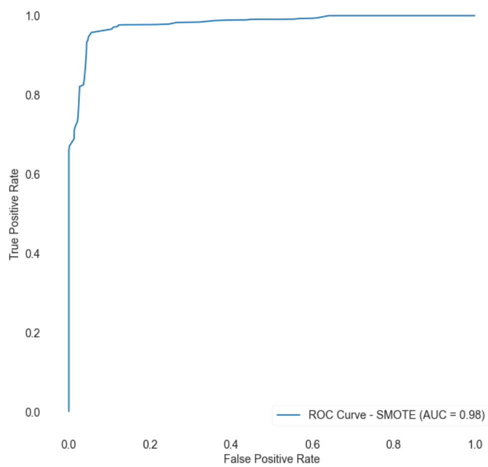
Different balancing algorithms were applied to the training set and *Random Forest* was used as a classifier, yielding the following results of the test set (*):

| | **Precision** | **Recall** | **Specificity** | **F1-Score** | **Geometric** | **IBA** |
|---|---|---|---|---|---|---|
| *Random Over Sampler* | 88% | 87% | 75% | 86% | 79% | 63% |
| *Random Under Sampler* | 91% | 90% | 82% | 90% | 85% | 73% |
| *SMOTE* | **94%** | **94%** | **90%** | **94%** | **92%** | **85%** |
| *ADASYN* | 90% | 90% | 81% | 89% | 85% | 72% |
| *SMOTEENN* | 89% | 88% | 77% | 87% | 81% | 66% |
| *SMOTETomek* | 89% | 88% | 77% | 87% | 81% | 67% |

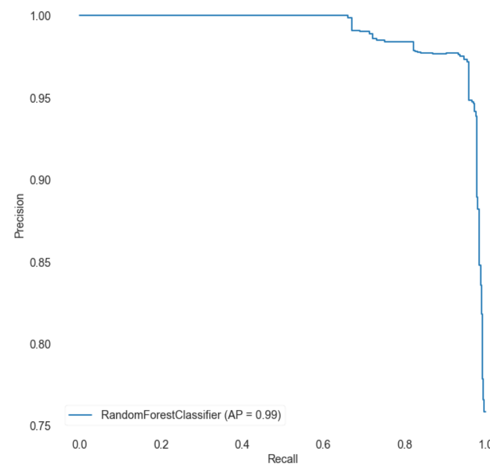
*(*) calculated using the balanced classification report of imblearn*




(a) Confusion matrix of test set for SMOTE balanced training set




(b) ROC Curve for SMOTE balanced training set




(c) Precision/Recall curve for SMOTE balanced training set

Alerts are ranked based on the probabiliy score produced by the classifier for the prediction target (actionable or not). In order to evaluate the ranking, two metrics are used to compare the ranked alerts with a random order. The experiments are run multiple times (by shuffling the random order and calculating the metrics again).

The order produced by the Random Forest classifer always outperforms the random order.

| | **S(R) metric** | **Average FP to TP** |
|---|---|---|
| *Better than random (Top 1000 alerts)* | 1000 out of 1000 | 1000 out of 1000 |
| *Better than random (All alerts)* | 10 out of 10 | 10 out of 10 |

Furthermore, the order of the ranked alerts if compared against the random and perfect order by using the cumulative graph.

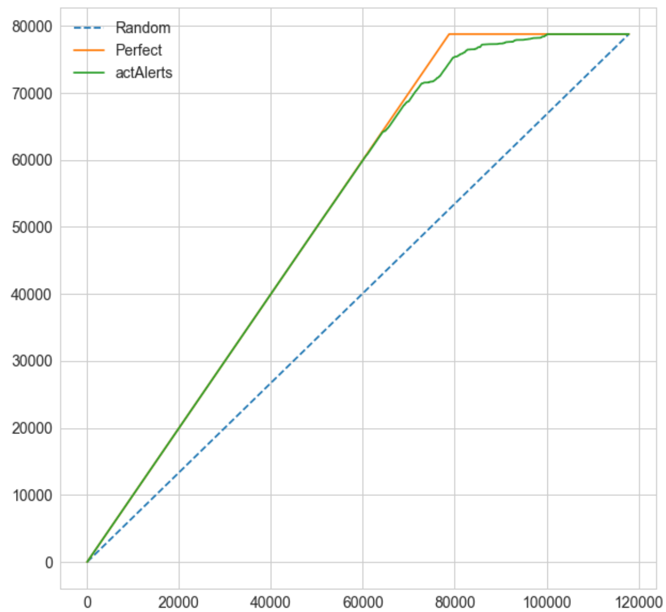
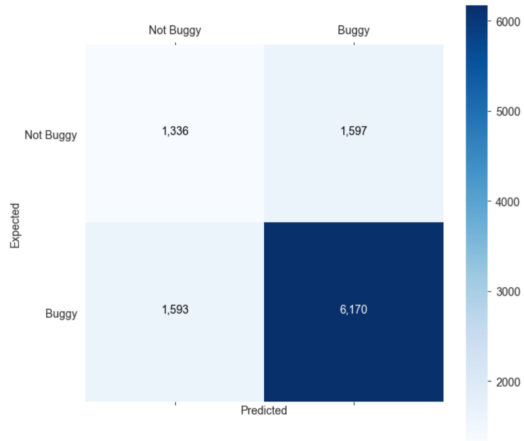


Figure 33: Cumulative graph for all sorted alerts

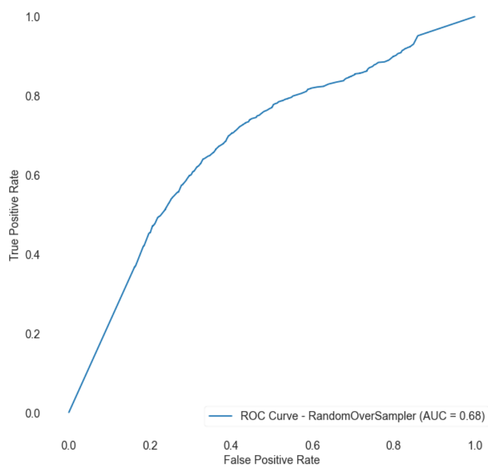
### 6.4 Method Bug Prediction

Different balancing algorithms were applied to the training set and *Random Forest* was used as a classifier. The following results show the score of the classifier in predicting if a method will be buggy in the future:

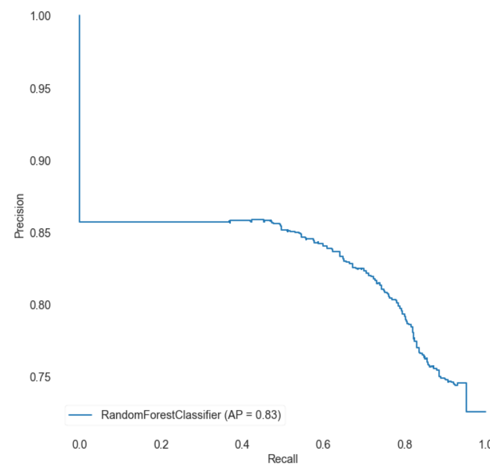
| | Precision | Recall | Specificity | F1-Score | Geometric | IBA |
|---|---|---|---|---|---|---|
| *Random Over Sampler* | **70%** | **70%** | **55%** | **70%** | **60%** | **37%** |
| *Random Under Sampler* | 65% | 70% | 39% | 66% | 42% | 18% |
| *SMOTE* | 68% | 70% | 48% | 69% | 53% | 29% |
| *ADASYN* | 67% | 70% | 45% | 68% | 51% | 26% |
| *SMOTEENN* | 66% | 70% | 41% | 67% | 45% | 21% |
| *SMOTETomek* | 68% | 70% | 47% | 69% | 52% | 28% |




(a) Confusion matrix of test set for Random Oversampling balanced training set




(b) ROC Curve for Random Oversampling balanced training set




(c) Precision/Recall curve for Random Oversampling balanced training set

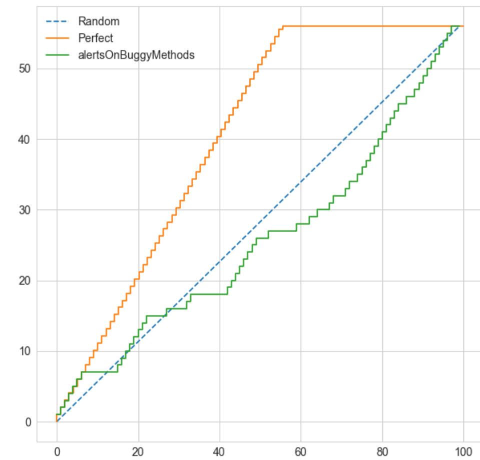
After calculating the probability of a method being buggy on the test set, we merge that set with the one containing actionable alerts. The goal is to see if ranking alerts higher when they belong to a bug prone method is useful in terms of predicting if that alert will be actionable in the future.

Alerts are ranked based on the probabiliy score produced by the classifier for the prediction target (buggy or not). In order to evaluate the ranking, two metrics are used to compare the ranked alerts with a random order. The experiments are run multiple times (by shuffling the random order and calculating the metrics again).

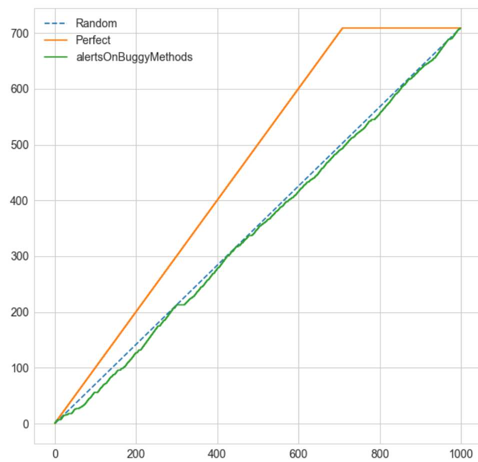
Alerts ranked by bug-prone method probability do not over-perform the random order in the first few alerts, but outperform the random order in the overall dataset.

| | **S(R) metric** | **Average FP to TP** |
|---|---|---|
| *Better than random (Top 100 alerts)* | 13 out of 1000 | 775 out of 1000 |
| *Better than random (Top 1000 alerts)* | 996 out of 1000 | 1000 out of 1000 |
| *Better than random (All alerts)* | 10 out of 10 | 9 out of 10 |

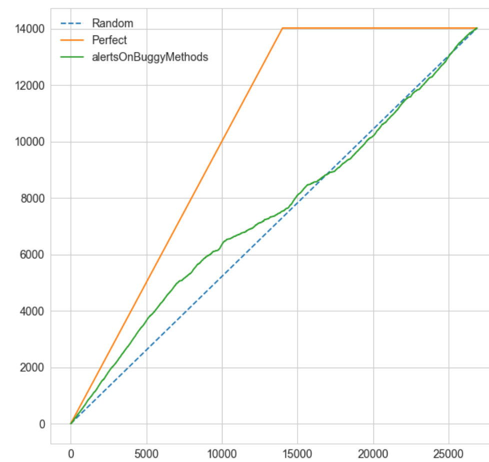
The order of the ranked alerts if compared against the random and perfect order by using the cumulative graph for different thresholds, the first 100, 1000 and all alerts.




(a) Cumulative graph for the first 100 alerts




(b) Cumulative graph for the first 1000 alerts




(c) Cumulative graph for all alerts

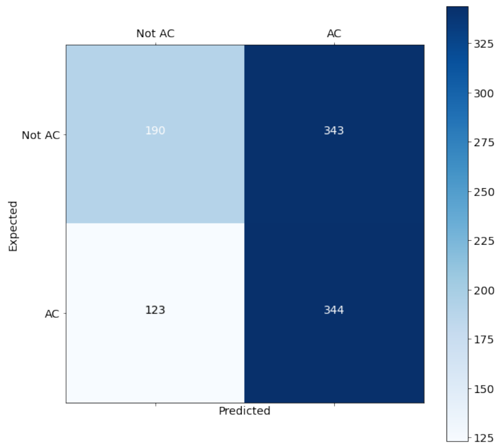
From the results we can conclude that ranking alerts based on the method being bug-prone does not provide good results. That does not mean that this method is meaningless. Its main purpose is to give extra attention to alerts belonging to buggy methods, which if acted upon may improve code quality and lower the probability of the method causing a bug in the future.

### 6.5 Feedback Rank

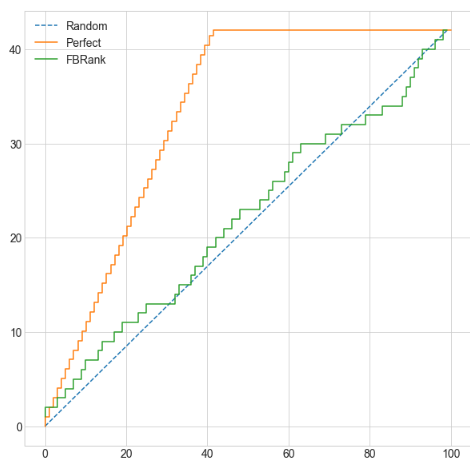
*(*) These experiments were run on a small subset of the dataset. The Bayes Network was trained on 10000 samples and tested on 1000. The reason being the abnormally high runtime (factor 100 or more than other approaches).*

Different balancing algorithms were applied to the training set and *Bayes Network* was used as a classifier. The following results show the score of the classifier:

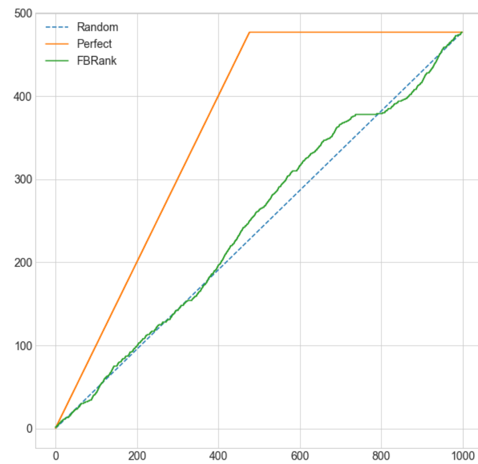
| | **Precision** | **Recall** | **Specificity** | **F1-Score** | **Geometric** | **IBA** |
|---|---|---|---|---|---|---|
| *Random Over Sampler* | 56% | 53% | 56% | 52% | 51% | 26% |
| *Random Under Sampler* | 57% | 54% | 57% | 52% | 52% | 27% |
| *None* | **58%** | **56%** | **57%** | **54%** | **53%** | **28%** |




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




(b) Cumulative graph for the first 100 alerts




(c) Cumulative graph for the first 1000 alerts

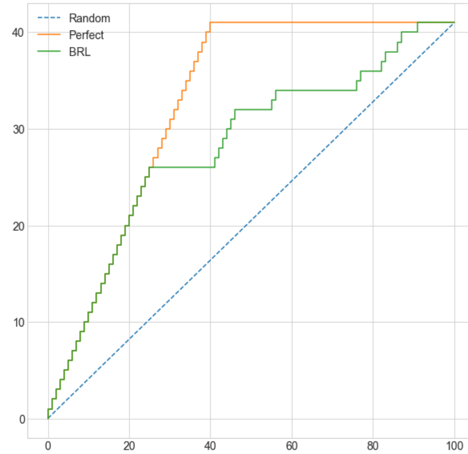
| | S(R) metric | Average FP to TP |
|---|---|---|
| *Better than random (Top 100 alerts)* | 1000 out of 1000 | 63 out of 1000 |
| *Better than random (Top 1000 alerts)* | 958 out of 1000 | 859 out of 1000 |

From the metrics as well as the cumulative graph, we can see that Feedback Rank does not perform well on the reduced dataset it was tested on.

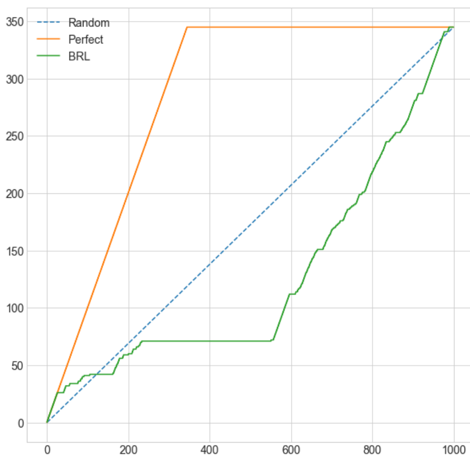
### 6.6 Bug-Related Lines

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.

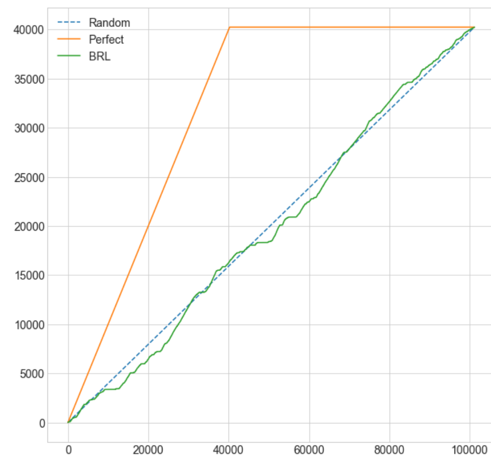
The algorithm ranks alerts types based on two input weight parameters $(\alpha, \beta)$. Based on experimental evaluation the following respective values were chosen: $\alpha = 0.7, \beta = 0.3$.




(a) Cumulative graph for the first 100 alerts




(b) Cumulative graph for the first 1000 alerts




(c) Cumulative graph for all alerts

The algorithm performs well for the first 100 alerts, but then the performance decreases.

| | **S(R) metric** | **Average FP to TP** |
|---|---|---|
| *Better than random (Top 100 alerts)* | 1000 out of 1000 | 993 out of 1000 |
| *Better than random (Top 1000 alerts)* | 0 out of 1000 | 996 out of 1000 |

### 6.7 Combined technique

**TO DOOOOOOO**

# 7 Threats to Validity

Henrique: In any empirical research, we must analysed and categorize the threats to validity of the experiments. The threats are classified into four cateegories: Construct Validity, Conclusion Validity, Internal Validity, and External Validity

# 8 Conclusions

## 8.1 Conclusions

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

## 8.2 Summary of Contributions

This thesis provides the following contributions:

- Building a workflow to extract information by making use of the Clang toolset and version history (SVN).
- Evaluating SA ranking techniques on an industrial codebase.
- Evaluating preprocessing techniques to deal with imbalanced and noisy data.
- Comparing different approaches on a common codebase.
- Exploring the utility of combining different methods.

## 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 messages/descriptions to have a clearer connection between a bug and an alert (did the alert really predict the bug?).

Given also the limited amount of data, new or improved approaches that can generalize easily are needed. In that regard, research can focus on the type of features extracted from the code or version history that are discriminative enough to make correct classification even with limited data. The impact of pre-processing can be explored even further, an example being on how to deal with high-cardinality data better, best encoding methods etc...

Furthermore, given the diverse nature of alerts current tools have, from finding bugprone construct, stylistic alerts, library-oriented alerts, to security or performance-oriented checks, different methods can be tailored that maximize performance within these subsets of alerts. For example, a simple method like Z-Ranking ([13]) can be more suited to predict stylistic alerts than others.

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

## References


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

[2] Infer static analyzer.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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

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

[27] 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.

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

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

[30] Clang AST.

[31] Clang Tidy.

[32] 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.

[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] 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.

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

[37] 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.

[38] 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.