Ranking the output of static analysis

Kleidi Ismailaj

kleidi.ismailaj@student.uantwerpen.be

July 17, 2020

Contents

Abstract **4**

1 Introduction **4**

2 Problem statement **5**

3 Background **5**

4 Literature review **5**

4.1 Dealing with False Positives 6

4.1.1 Single Tool 6

4.1.2 Multiple Tools 13

4.1.3 Literature review papers 14

4.1.4 Comparative studies 16

4.2 Bug Prediction 17

4.3 Static Analysis tools in practice 20

5 Ranking the output of Static Analysis **23**

5.1 Collecting data 23

5.1.1 Clang AST 23

5.1.2 Clang Tidy 23

5.1.3 Workflow for collecting information 24

5.1.4 Assumptions made for collected data 24

5.2 Data overview 25

5.3 Dealing with unbalanced and noisy data 28

5.4 Feedback Rank and Z-Score 29

5.4.1 Z-Score 29

5.5 Detecting Actionable Alerts via Machine Learning 29

5.5.1 Alert Characteristics 30

5.5.2 Pre-processing data 30

5.6 Bug related lines 31

5.7 Bug prediction 31

5.8 Combining the techniques 33

5.8.1 How to combine the strengths? 33

5.8.2 Better results? 33

5.9 Evaluation methods 33

5.9.1 Evaluation Metrics 33

5.9.2 Preparing the dataset 34

5.10 Results of individual tools 34

5.10.1 Random test set 34

5.10.2 Release based testing . . . 34
5.10.3 Alert ratio . . . 34
5.10.4 Average false positives to inspect . . . 34

6 Threats to Validity 34

7 Conclusions 34
7.1 Conclusions . . . 35
7.2 Summary of Contributions . . . 35
7.3 Future Research . . . 35

Abstract

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

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

1 Introduction

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

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

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

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

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

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

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

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

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

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 warnings (or the context on which the warnings appear).

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

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

The research questions can be formulated as follows:

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

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

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

3 Background

Henrique: A Background section is not an Appendix. Specially for a thesis such as this, the main concepts you want to describe are important for a reader to better understand the techniques used.

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

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

  Henrique: We usually describe Precision, Recall and other metrics used in the Evaluation as a subsection inside the evaluation.

4 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
  - Combining multiple tools
  - Looking at survey papers that describe the state of the research and state of the art techniques.
  - Comparative studies evaluating different methods.

- *Bug prediction*: Rank the alerts in problematic parts of code higher.

- Information on *real world usage of SA tools*, problems and suggested solutions.

4.1 Dealing with False Positives

4.1.1 Single Tool

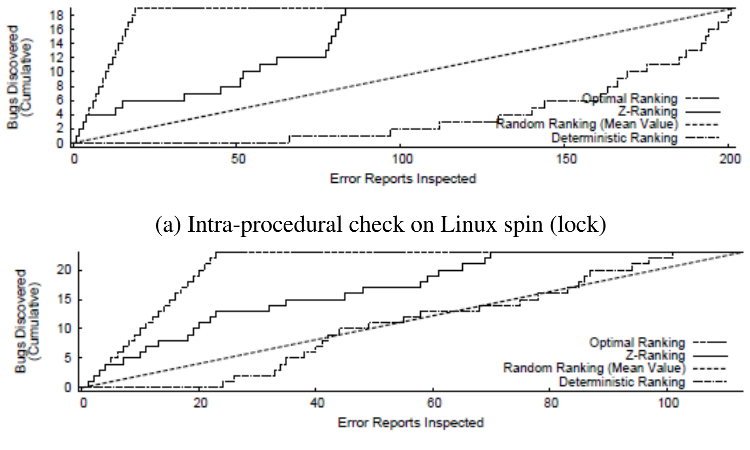
Kremenek and Engler [2] introduce *Z-ranking*, a statistical model to rank the error reports of SA tools. They make a distinction between successful and failed checks (those that satisfy a checked property and those that violate it). The underlying observation is that the most reliable error reports are those that generated few failed checks and many successful checks, since the actual amount of bugs in code is relatively small. An explosion of failed checks is a likely indicator that something is going wrong with the analysis. Reports are sorted based on the calculated *z-test* statistic (based on the relative frequency of successful and failed checks).

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

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

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

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



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

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

Kremenek et al. [3] 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.

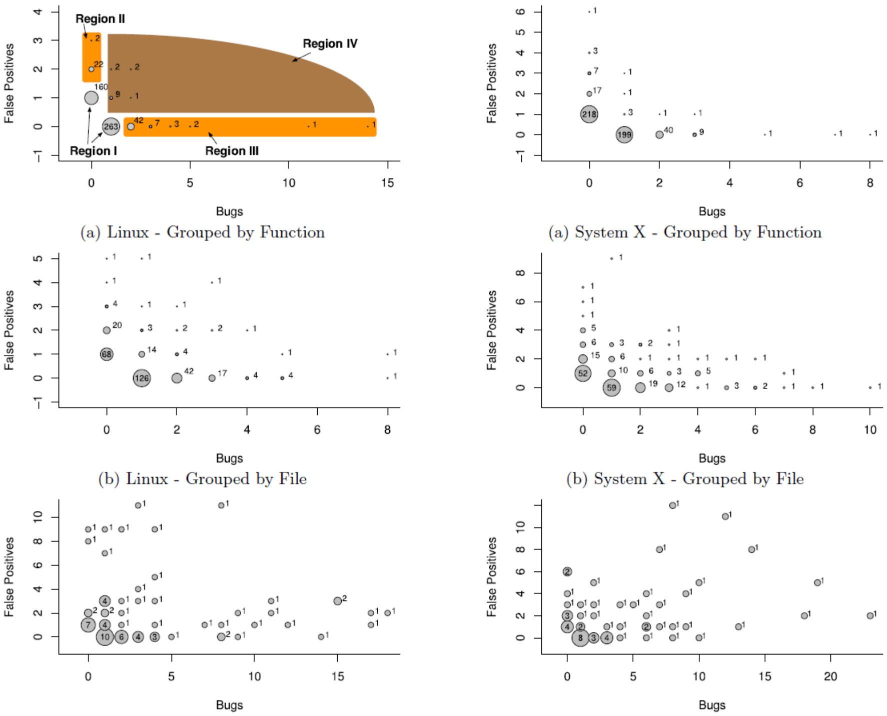
An intuitive explanation for the reason that reports cluster is given for both real and false positives. Regarding true positives, when developers do not know a rule, they will repeatedly violate it, so errors of the same type will correlate together. As for the false positives, there are three main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

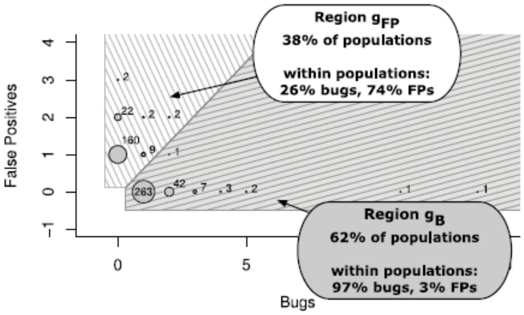


(a) Linux - Grouped by Function (a) System X - Grouped by Function

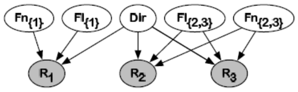
(b) Linux - Grouped by File (b) System X - Grouped by File

(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 [4] 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.

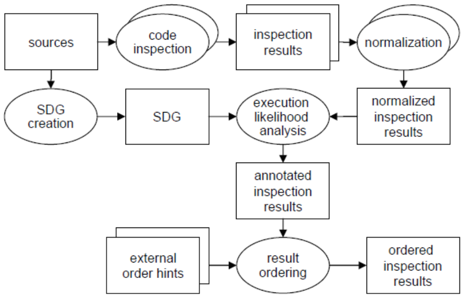
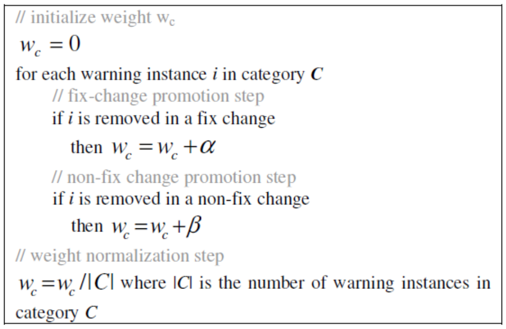

sources
code inspection
inspection results
normalization
SDG creation
SDG
execution likelihood analysis
normalized inspection results
annotated inspection results
external order hints
result ordering
ordered inspection results

Figure 3: Workflow for the ELAN tool

Kim and Ernst [5] 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%.

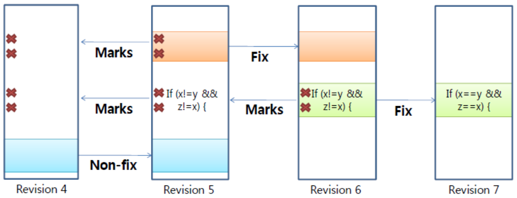
They set a weight to each warning category to represents its importance. The weight will be proportional to the number of warnings eliminated by changes (where fix-changes have the biggest weight, fig. 4a). Selecting the top weighted warnings improves precision up to 17%, 25%, and 67% respectively. Precision 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).

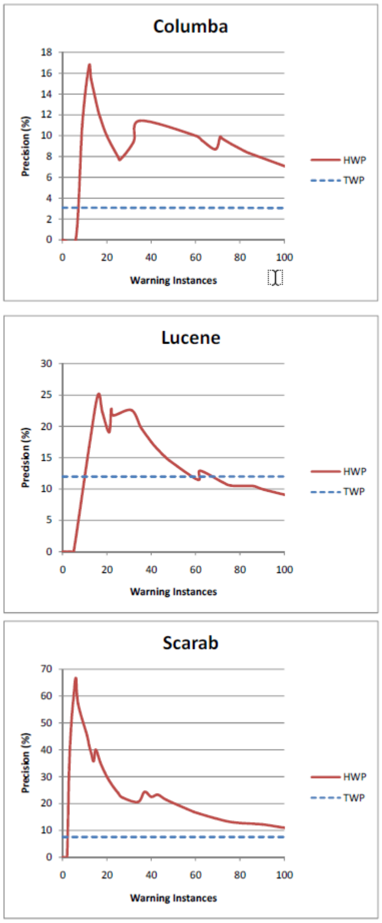


```
// initialize weight w_c
$w_c = 0$
for each warning instance $i$ in category $C$
    // fix-change promotion step
    if $i$ is removed in a fix change
        then $w_c = w_c + \alpha$
    // non-fix change promotion step
    if $i$ is removed in a non-fix change
        then $w_c = w_c + \beta$
// 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

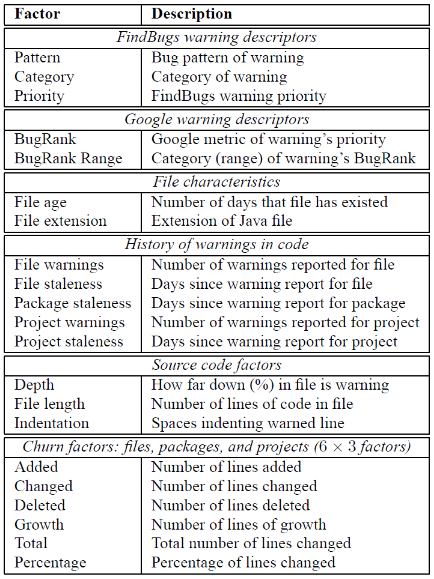


(c) Line marking approach

Ruthruff et al. [6] use *logistic regression* models to not only reduce the number of false positives in the output of SA tools, but also to predict actionable warnings. Warnings are not always acted on by developers even if they reveal true defects. The reason may be that the defects may have little impact and require significant effort for little perceived benefit. Furthermore, they introduce a statistical methodology for discarding features with low predictive power and thus avoiding the capture of expensive data. Information to build the models is mainly drawn from: (a) light-weight code complexity metrics for post-release bug prediction, (b) file (history) information to predict fault counts within individual files. The features include the history of warnings, source code characteristics, churn factors, and warnings descriptors (fig. 5a).

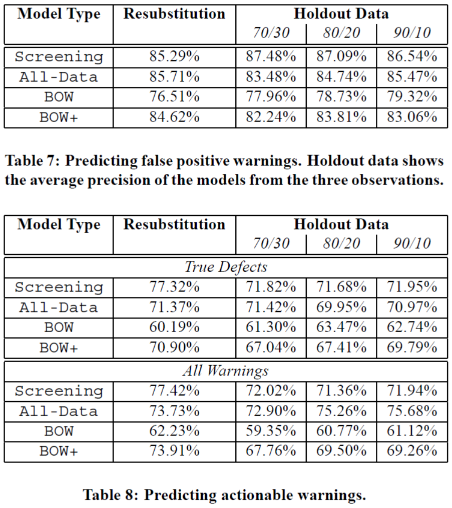
Their screening methodology, for selecting an independent subset of predictor features, consists of up to four stages, and attempts to identify at least six predictive features. The stages respectively consider 5%, 25%, 50%, 100% of the warnings, continuously removing features with low predictive power.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

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



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

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



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

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

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

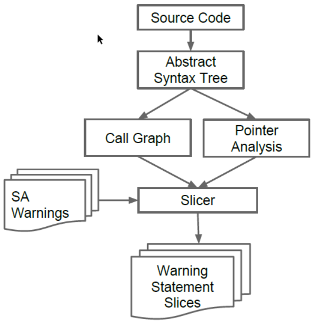
Table 8: Predicting actionable warnings.

(b) Results for predicting true positives and actionable warnings

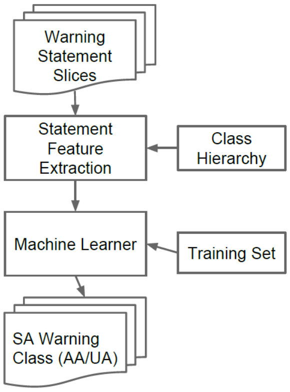
Hanam et al. [7] present a method for differentiating actionable and unactionable alerts by finding alerts with similar code patterns (alerts with similar patterns are probably of the same type). They use a feature vector based on code characteristics at the site of each SA alert along with a classifier 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 [8] 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.

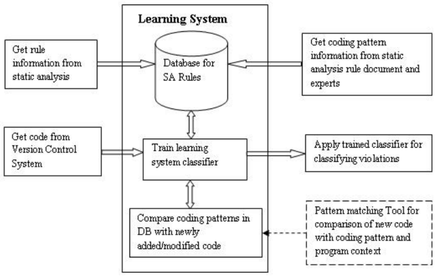
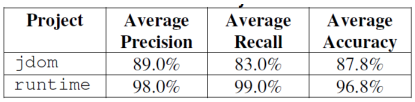

Learning System
Get rule information from static analysis
Database for SA Rules
Get coding pattern information from static analysis rule document and experts
Get code from Version Control System
Train learning system classifier
Apply trained classifier for classifying violations
Compare coding patterns in DB with newly added/modified code
Pattern matching Tool for comparison of new code with coding pattern and program context

Figure 7: Workflow for incremental violation detection

Heckman and Williams [9] 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.



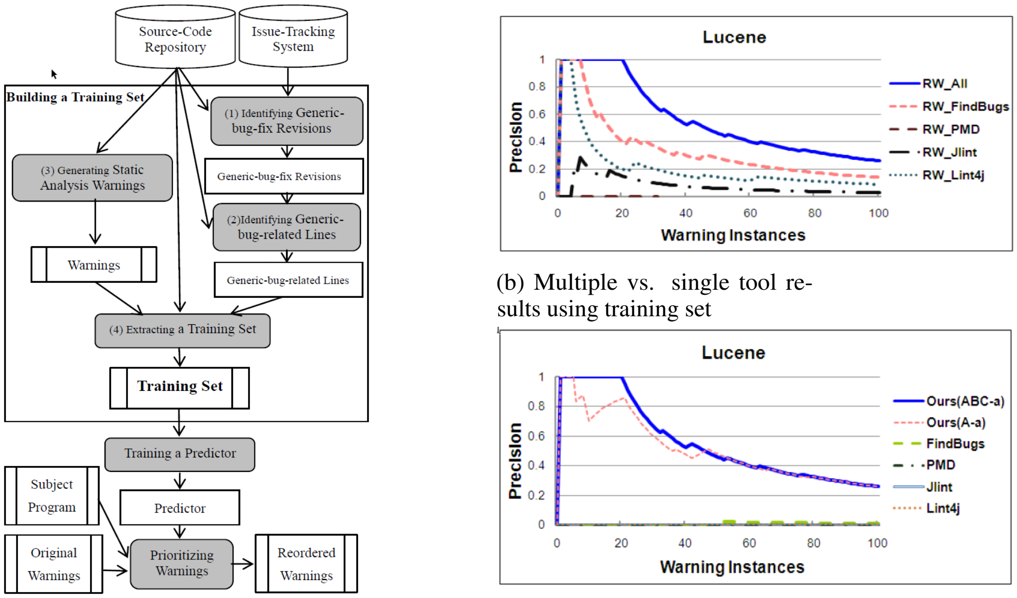
Project	Average Precision	Average Recall	Average Accuracy
jdom	89.0%	83.0%	87.8%
runtime	98.0%	99.0%	96.8%

Figure 8: Average metrics for all models

Liang et al. [10] 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

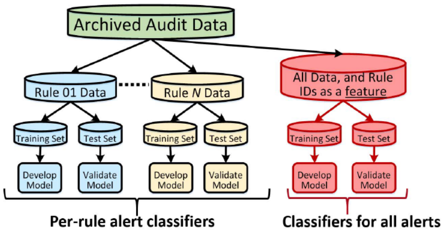
(b) Multiple vs. single tool results using training set

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

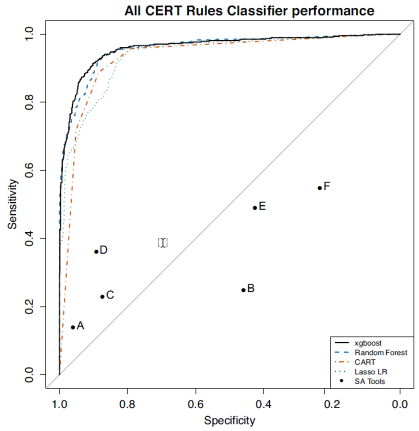
4.1.2 Multiple Tools

Flynn et al. [11] 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

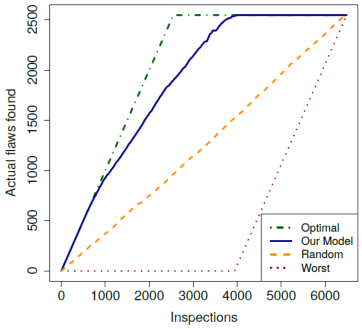
Ribeiro et al. [12] aim to reduce the false positive rate of an ensemble of static analyzers by using *Decision Trees* and *AdaBoost*. The goal is to make possible to combine the strengths of different analyzers without suffering too much from false positives. Their approach ignores source code characteristics, 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

4.1.3 Literature review papers

Heckman and Williams [13] 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.

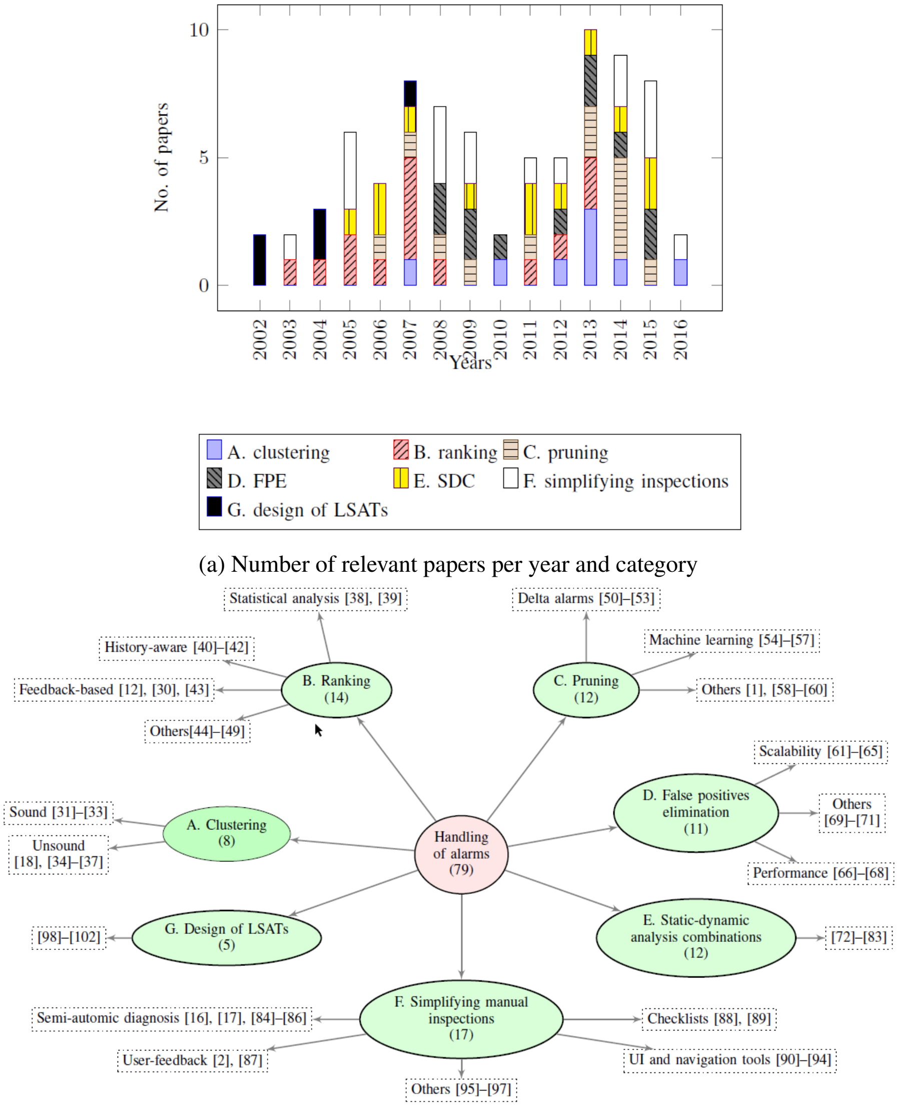
Approaches followed by AAIT's fall into seven main categories: (a) alert type selection (selecting altert types that are the most relevant for a codebase), (b) contextual information (limiting SA tools only to parts of code where that they can analyze well), (c) data fusion (combining multiple SA tools), (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 [14] 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

4.1.4 Comparative studies

Heckman and Williams [15] 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. [16] 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.

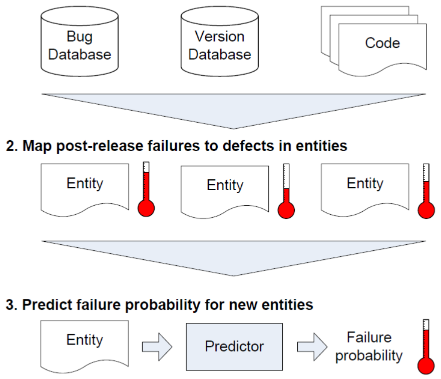
They found out that Aware and FeedbackRank perform significantly better than the other ranking approaches. In addition individual alert raking algorithms performed better than those that rank alert types. Also they did not find a clear distinction in performance between statistical and ad-hoc approaches.

4.2 Bug Prediction

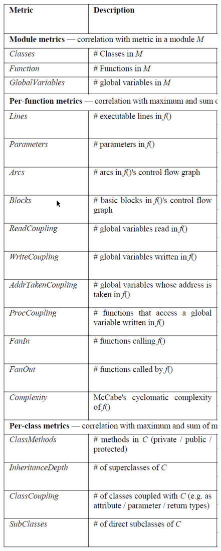
Nagappan et al. [17] use code complexity metrics to predict the likelihood of *post-release* defects for new entities (failures that occurred in the field six months after the release). Although, according to their findings, these metrics are correlated to failure prone entities, there is no universal 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.


Bug Database
Version Database
Code
2. Map post-release failures to defects in entities
Entity
Entity
Entity
3. Predict failure probability for new entities
Entity
Predictor
Failure probability

(a) Workflow for predicting defects on new entities



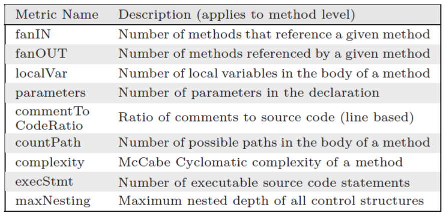
Metric	Description
Module metrics — correlation with metric in a module *M*	
Classes	# Classes in *M*
Function	# Functions in *M*
GlobalVariables	# global variables in *M*
Per-function metrics — correlation with maximum and sum	
Lines	# executable lines in *f*()
Parameters	# parameters in *f*()
Arcs	# 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	
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. [18] 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).

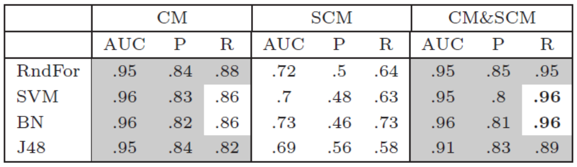
The models are based on source code metrics that are applicable on method level (fig. 14a) while change metrics are based on fine-grained operations extracted from AST comparisons (tree edit operations needed to transform one AST into the other, combined with semantic information from the source code, 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).



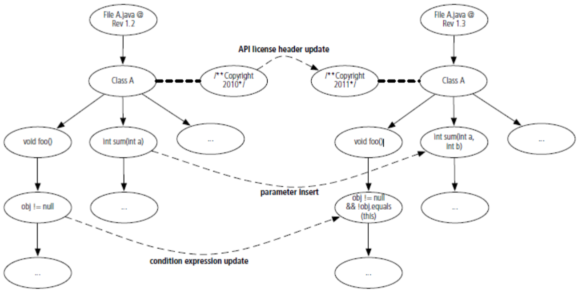
Metric Name	Description (applies to method level)
fanIN	Number of methods that reference a given method
fanOUT	Number of methods referenced by a given method
localVar	Number of local variables in the body of a method
parameters	Number of parameters in the declaration
commentTo CodeRatio	Ratio of comments 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



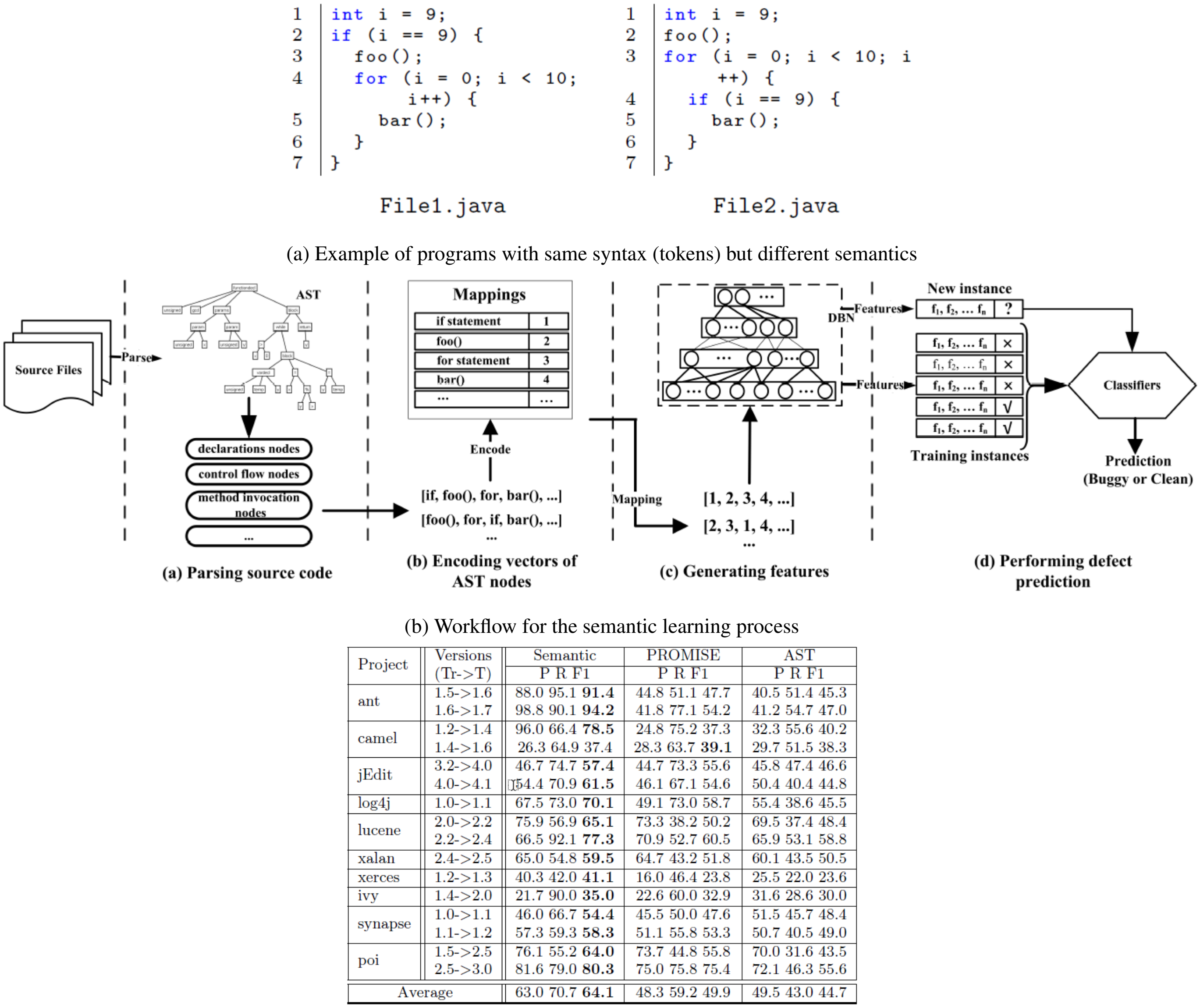
(c) Fine grained code changes extracted from AST comparisons

Wang et al. [19] 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

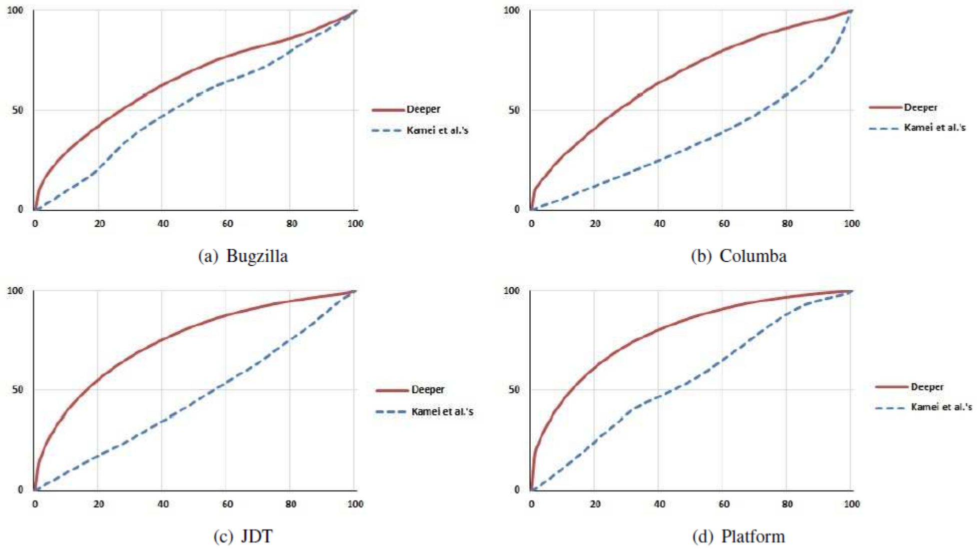
Project	Versions (Tr->T)	Semantic P R F1	PROMISE P R F1	AST P R F1
ant	1.5->1.6	88.0 95.1 **91.4**	44.8 51.1 47.7	40.5 51.4 45.3
	1.6->1.7	98.8 90.1 **94.2**	41.8 77.1 54.2	41.2 54.7 47.0
camel	1.2->1.4	96.0 66.4 **78.5**	24.8 75.2 37.3	32.3 55.6 40.2
	1.4->1.6	26.3 64.9 37.4	28.3 63.7 **39.1**	29.7 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. [20] 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

4.3 Static Analysis tools in practice

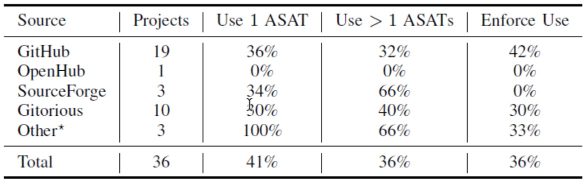
Beller et al. [21] conduct a large scale evaluation of how SA tools are used in practice in open-source systems. They study the prevalence of SA tools, their configurations and how they evolve.

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.

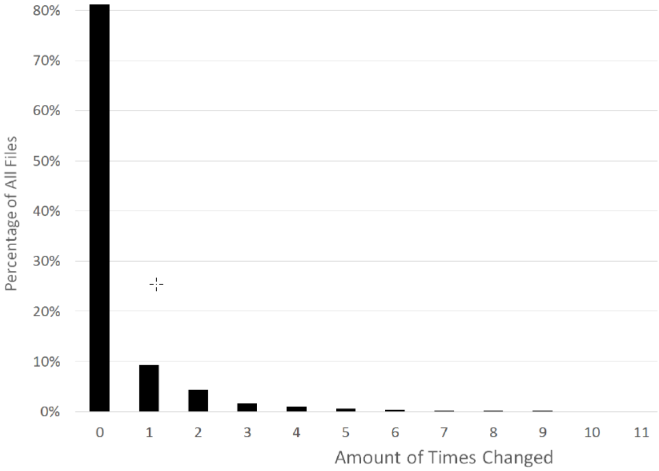
In the analyzed projects, most enabled rules belong to the maintainability category, and only 35% of the enabled rules belong to a functional category. Both the majority of actively enabled and disabled rules are maintainability-related.

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	30%	40%	30%
Other*	3	100%	66%	33%
Total	36	41%	36%	36%

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



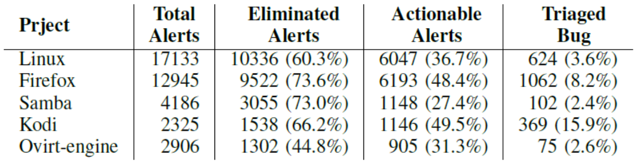
(b) Changes in configuration of SA tools

Imtiaz et al. [22] 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.

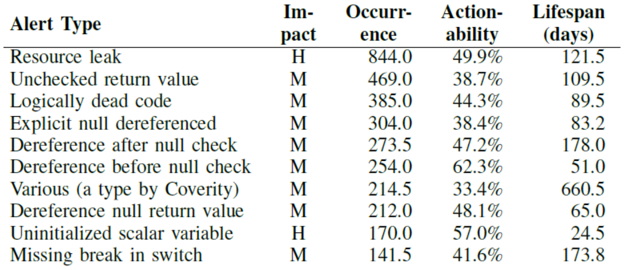
The median lifespan of actionable alerts varies between projects, ranging from 36 to 245 days, and the complexity of code changes is generally low. This means that developers generally take a long time to fix the alerts despite the fixes being low in complexity.

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

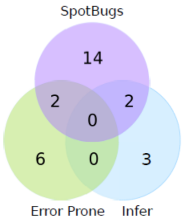
(b) Top 10 alert occurrences for C/C++

Habib and Pradel [23] 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.

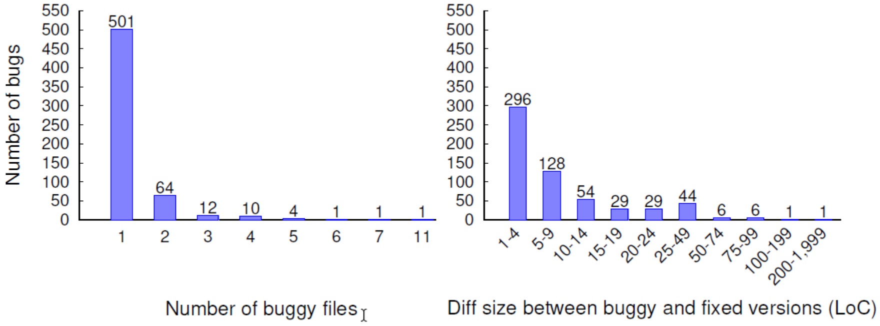
The three bug detectors together reveal 27 of the 594 studied bugs (4.5%). Some of the missed bugs could have been found by variants of the existing detectors, while most of them are domain-specific problems that do not match any existing bug pattern that the SA tool have. By manually analyzing a small 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).

5 Ranking the output of Static Analysis

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

The different techniques that were tested are:

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

5.1 Collecting data

Clang AST ([24]) and Clang Tidy ([25]) was used to analyze the code. It was chosen is because it's a reliable open source tool, extensible and most importantly supported by the codebase of the company.

5.1.1 Clang AST

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

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

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

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

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

5.1.2 Clang Tidy

Clang Tidy is a modular C++ linter tool which provides an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis ([25]).

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 and used during code analysis:

- Clang Static Analyzer checks
- Checks related to C++ Core Guidelines
- Checks that target bugprone code constructs
- Checks related to CERT Secure Coding Guidelines
- Checks related to Boost library

5.1.3 Workflow for collecting information

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

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

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

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

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

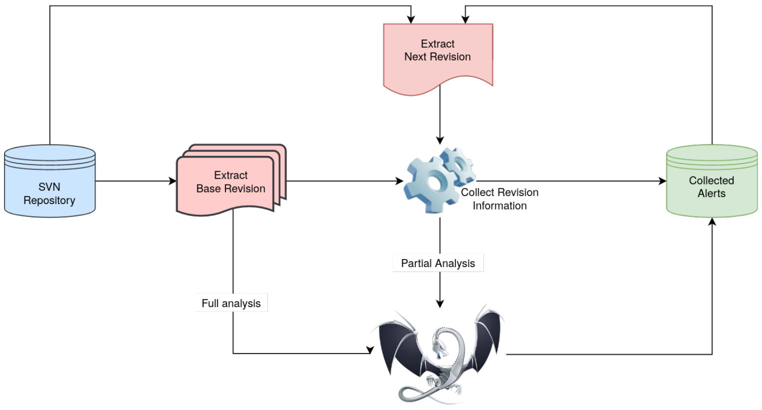


Figure 20: Collecting alerts and other information

5.1.4 Assumptions made for collected data

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

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

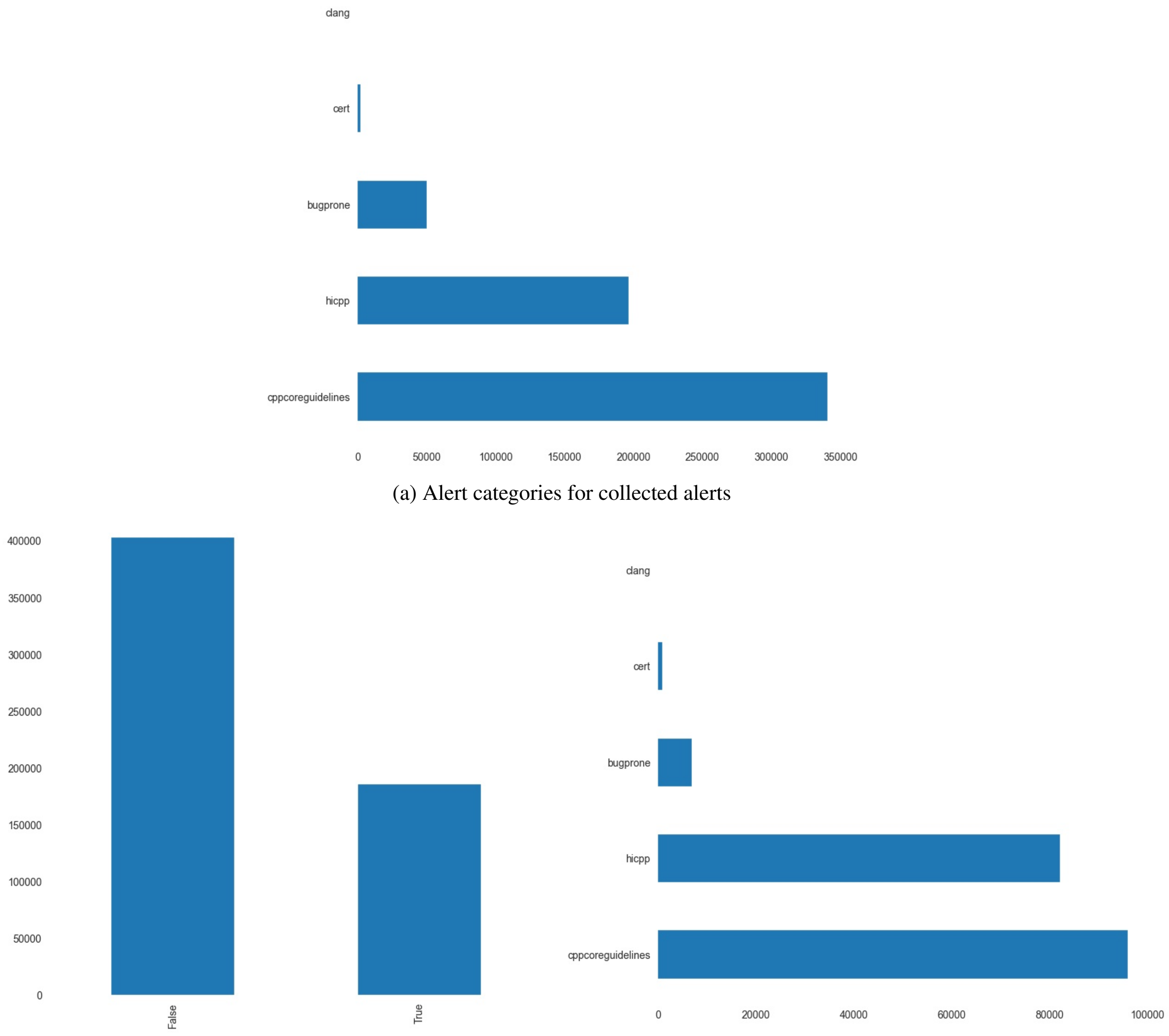
Another algorithm uses alerts pointing at past bugs as a way to prioritize future alerts. Also in this case an assumption is needed: the alerts that pointed to past bugs, were directly related to that bug. This may not always be true and can cause the aforementioned problems as a result.

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

5.2 Data overview

TODO: Analyze of large portion of revisions and produce statistics
TODO: Bug statistics for bug related lines?

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 (**TODO: be more precise: There are X packages, Y files, Z lines of code. Analyzed X revisions, from Y date to Z date.**).

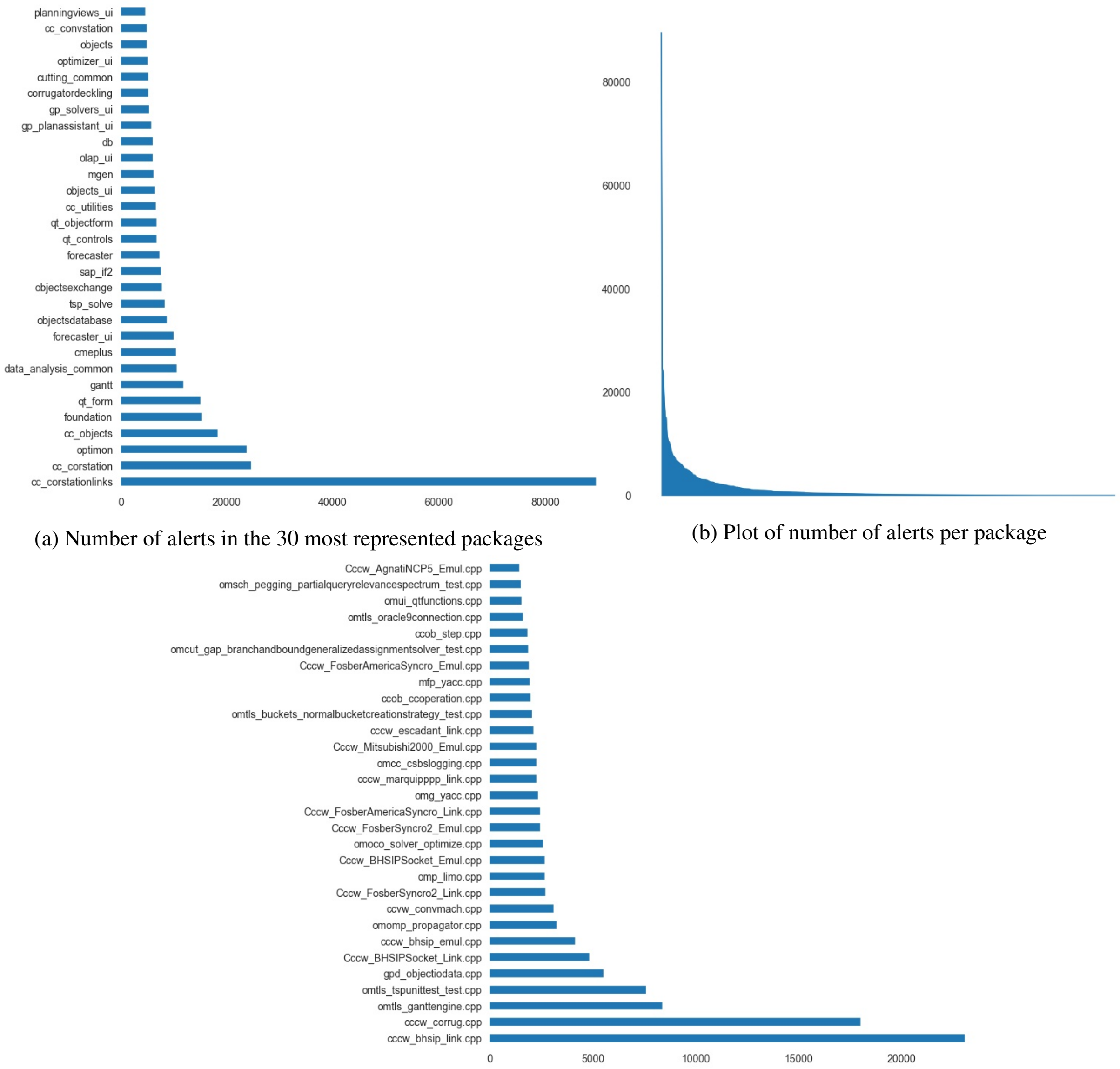


(a) Alert categories for collected alerts

(b) Number of open and closed alerts

(c) Number of closed alerts per category

As can be seen from the plots, there is a mismatch between the number of open/closed alerts, as well as alerts categories (some of them are under represented like clang/cert).

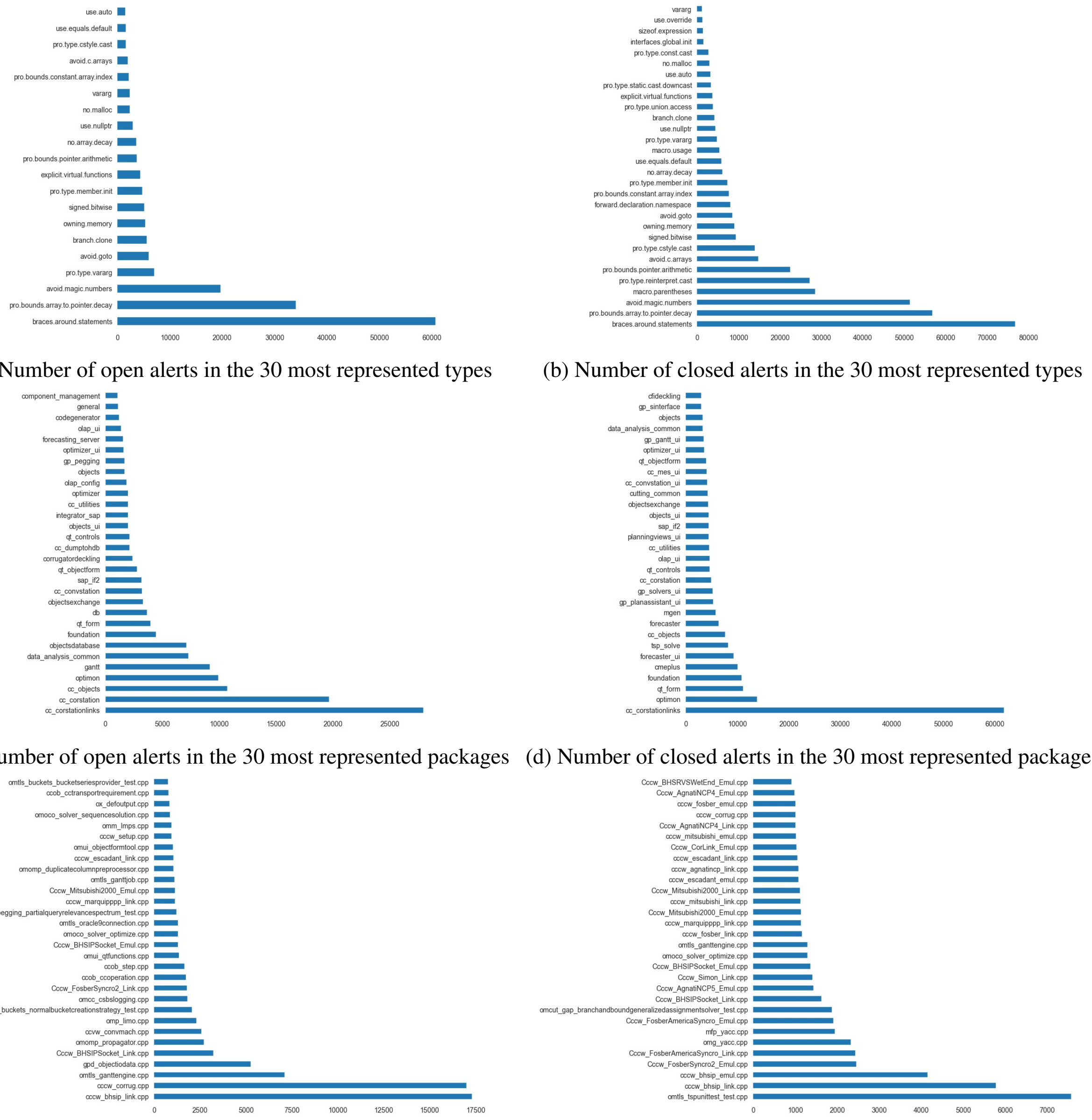


(a) Number of alerts in the 30 most represented packages

(b) Plot of number of alerts per package

(c) Number of alerts in the 30 most represented files

Regarding the distribution of the alerts in packages and files, we see similar trends: some of the packages/files contain a big chunk of the total alerts and most of them contain little to no alerts.



(a) Number of open alerts in the 30 most represented types

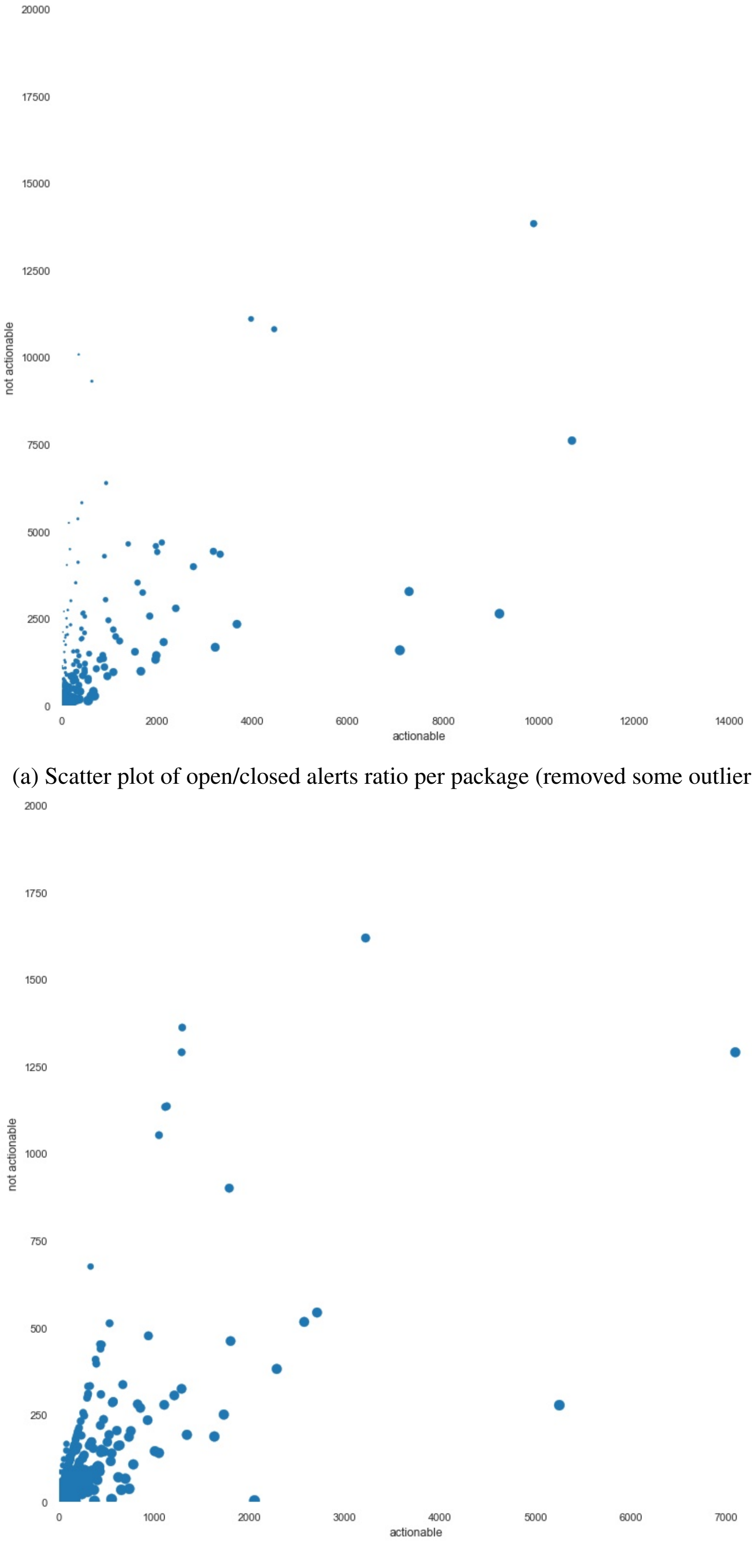
(b) Number of closed alerts in the 30 most represented types

(c) Number of open alerts in the 30 most represented packages

(d) Number of closed alerts in the 30 most represented packages

(e) Number of open alerts in the 30 most represented files

(f) Number of closed alerts in the 30 most represented files



(a) Scatter plot of open/closed alerts ratio per package (removed some outliers)

(b) Scatter plot of open/closed alerts ratio per file (removed some outliers)

5.3 Dealing with unbalanced and noisy data

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

Kim et al. [26] 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 [26], 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. [33], 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 rather limited, there seems to be a necessity to apply a combination of balancing and cleaning to the dataset.

TO DO: maybe insert before and after graph for data cleaning/balancing?

5.4 Feedback Rank and Z-Score

Feedback Rank ([3]) combined with Z-Score ([2]) is a simple technique which ranks alerts on the probability of them being true/actionable. It consists of three basic features: package, file, function of the alert being analyzed (though may be extended as needed). By constructing a Bayesian Network based on these three features and the history of the project (which alerts have been proven valuable) it produces a probability ranking about which alerts are most useful.

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

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

The initial configuration of the network can learned from historical data (extracted as explained on section 5.1.3). According to the original authors 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.

To construct the BN, the Pomegranate library was used ([27]), trained with the extracted alert data from the version history.

TO DO: used as static, explain zranking assumption, add prediction/train time

5.4.1 Z-Score

To break ties when the probabilities provided by the BN are equal between alerts, the Z-Score metric is used based on the number of alerts on the same file. Z-Score is used in Z-Ranking ([2]), which makes use of the observation that the most reliable error reports are those that generated few failed checks and many successful checks, since the actual amount of bugs in code is relatively small.

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

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

5.5 Detecting Actionable Alerts via Machine Learning

Different research papers have focused on automatically classifying alerts in true/false positives or actionable/unactionable, by constructing classifiers based on code or change metrics ([6], [9]).

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

Research is conducted by following the example of [9], since it contains an agglomeration of alert characteristics (AC) collected from other research papers.

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

The scikit-learn library was used to perform ML experiments ([28]).

5.5.1 Alert Characteristics

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

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

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

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

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

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

5.5.2 Pre-processing data

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

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

Imbalanced dataset A quick look at the number of closed or not closed alerts shows that the dataset is heavily unbalanced. In order to train the ML algorithms we need first to balance the dataset. Two main techniques were tried: random undersampling and SMOTE (by using the library from [29]).

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

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

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

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

5.6 Bug related lines

Another automatic way to determine which alerts are useful is to check if they pointed to lines that were changed during bug fixes. By doing so, we are regarding as valuable only those alerts that potentially signaled future bugs. The concept of bug related lines (BRL) is used in [5] and [10].

BRLs are calculated as follows:

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

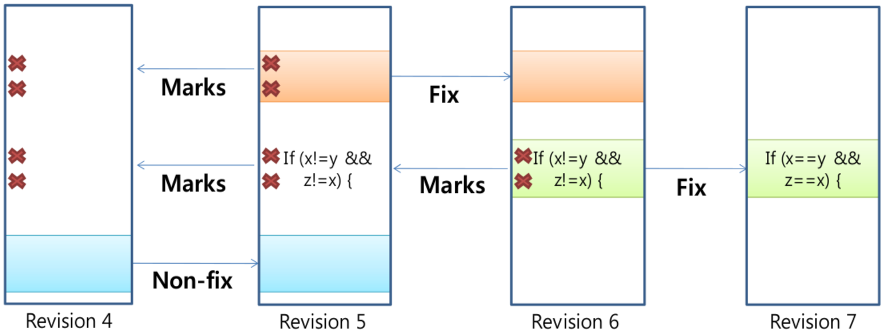


Figure 25: Calculating BRLs ([5])

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

Given that the number of alerts collected this way, is a lot less than the total number of alerts, this method is more suited for ranking alert types rather than individual alerts (as used in [5]).

5.7 Bug prediction

Following the example of [4], 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, while anything lower than method level would be too hard to predict as bug prone.

We follow the example of the [18] 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.

Algorithm 1: Alert type priotitization algorithm

```
Data: Collected bug-related 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 then
        $w_t = w_t + \alpha$
    end
    else if alert pointed to a BRM then
        $w_t = w_t + \beta$
    end
end
```

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

5.8 Combining the techniques

5.8.1 How to combine the strengths?

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

5.8.2 Better results?

Does it provide better results?

5.9 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 [30] and as can be seen on fig. 26, 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.
- **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.
- **Alert ratio**: where we compare how much better is the ranking produced by a model with a random order of alerts. We calculate that by checking how many more alerts are better placed in the ranked version than the random one.
- **Average inspected false positives**: similar to the previous method where we compare the ranking produced by a model against a random one, but by calculating the average number of times an unactionable alert has to be inspected before reaching an actionable one.
- **Fault detection rate curve**: the curve of a model is formed by the percentage of all actionable alerts found within the first *X* alerts of the alert ranking algorithm.

  TO DO: see [16] and apply only on top x%? Fault detection rate curve?

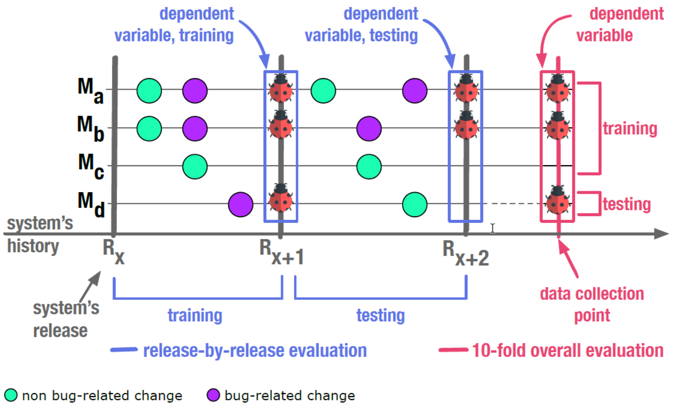


Figure 26: Release-based vs K-Fold ([30])

5.9.1 Evaluation Metrics

Along with the classic evaluation metrics like *accuracy*, *precision* and *recall*, other metrics are used that are more appropriate for imbalanced data ([31], [32]).

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

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.

5.9.2 Preparing the dataset

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

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

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

TODO: EVALUATE OVERSAMPLING METHODS

TODO: EVALUATE ENCODING METHODS

5.10 Results of individual tools

5.10.1 Random test set

5.10.2 Release based testing

5.10.3 Alert ratio

5.10.4 Average false positives to inspect

6 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

7 Conclusions

TO DO: FEASIBILITY STUDY BECAUSE OF LIMITED AMOUNT OF DATA

7.1 Conclusions

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

7.2 Summary of Contributions

This thesis provides the following contributions:

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

7.3 Future Research

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

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

Furthermore, given the diverse nature of alerts current tools have, from finding bugprone construct, stylistic alerts, library-oriented alerts, to security or performace-oriented checks, different methods can be tailored that maximise performance within these subsets of alerts. For example, a simple method like Z-Ranking ([2]) can be more suited to predict stylistic alerts than others.

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

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] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

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

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

[24] Clang AST.

[25] Clang Tidy.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.