A Model Building Process for Identifying Actionable Static Analysis Alerts

Sarah Heckman and Laurie Williams
North Carolina State University
sarah_heckman@ncsu.edu and *williams@csc.ncsu.edu*

Abstract

Automated static analysis can identify potential source code anomalies early in the software process that could lead to field failures. However, only a small portion of static analysis alerts may be important to the developer (actionable). The remainder are false positives (unactionable). We propose a process for building false positive mitigation models to classify static analysis alerts as actionable or unactionable using machine learning techniques. For two open source projects, we identify sets of alert characteristics predictive of actionable and unactionable alerts out of 51 candidate characteristics. From these selected characteristics, we evaluate 15 machine learning algorithms, which build models to classify alerts. We were able to obtain 88-97% average accuracy for both projects in classifying alerts using three to 14 alert characteristics. Additionally, the set of selected alert characteristics and best models differed between the two projects, suggesting that false positive mitigation models should be project-specific.

1. Introduction

Automated static analysis tools can be used to identify potential source code anomalies, which we call *alerts*, early in the software process that could lead to field failures [9]. Ayewah et al. [2] state, "In many cases, the person who writes the code is responsible for reviewing the [alert], deciding whether it's relevant, and resolving the issue." Inspection of each alert by a developer is required to determine if the alert is an indication of an important anomaly that the developer wants to fix, which we call a *true positive* (TP) or an *actionable alert* [3, 17]. When an alert is not an indication of an anomaly or is deemed unimportant to the developer (e.g. the alert indicates a programmer mistake inconsequential to program functionality), we call the alert a *false positive* (FP) or an *unactionable alert* [3, 17]. Static analysis tools may generate an overwhelming number of alerts [11], the majority of which are likely to be unactionable [9]. FP mitigation techniques utilize information about the alerts, called *alert characteristics* (AC), to prioritize alerts by the likelihood of being actionable or to classify alerts into actionable and unactionable groups [7, 10-12, 17, 19].

Current models [7, 8, 10-12, 17, 19] do not systematically consider potential ACs or models, and could be improved by an investigation into the best predictors and models. Additionally, models that work for one program may be ineffective on other programs. For any FP mitigation model, we want to extract two types of useful information from static analysis alert data: 1) sets of ACs that are predictive of actionable alerts; and 2) which models (using the predictive ACs) are best at classifying. Machine learning can be used to identify predictive ACs and machine learners (specific machine learning algorithms) can generate models for classifying alerts.

The goal of our research is *to reduce the number of unactionable static analysis alerts requiring inspection by a developer.* We hypothesize that the important ACs and machine learners will vary by project. Therefore, we propose a four-stage FP mitigation model building process via machine learning for building project-specific models: 1) gather the ACs about alerts generated from static analysis; 2) use attribute subset evaluation algorithms to select important sets of ACs; 3) use the selected ACs and machine learners to build predictive models; 4) select the best model by comparing their predictive power.

To demonstrate our four-stage FP mitigation model building process, we performed a case study on two open source applications, `jdom` and `org.eclipse .core.runtime` (abbreviated as `runtime`), that are part of the FAULTBENCH [7] benchmark for evaluation of FP mitigation models. For each project, we gathered the alerts; size and complexity metrics; source code history; and code churn. We considered rule-based, decision tree, linear function, k-nearest neighbor, and Bayesian network machine learners for building FP mitigation models. We compare the two subject programs to test our hypothesis that important ACs and machine learners will vary by project.

The rest of the paper is structured as follows: Section 2 presents related work, Section 3 describes the candidate ACs, Section 4 describes the model building process, Section 5 presents the research methodology, Section 6 presents the results, and Section 7 concludes and presents future work.

2. Related work

This section describes the related work in the areas of FP mitigation techniques and machine learning applied to software engineering problems.

2.1. False positive mitigation techniques

Our prior research [7, 8] has proposed a project-specific, in-process, FP mitigation prioritization technique that utilizes the alert's type and location at the source folder, class, and method levels. The model, AWARE-APM [7, 8], uses developer feedback in the form of *alert suppression* and *alert closures*. Suppressing an alert is an explicit developer action to indicate the alert is unactionable. Closure is determined by comparing subsequent static analysis runs. If the alert is not in the later run, the alert is closed. After a developer inspects the alert and takes an action on that alert, the prioritization of the remaining alerts is adjusted from the feedback. We evaluated three versions of AWARE-APM model on FAULTBENCH subject programs and found an average accuracy of 67-76% [7]. The precision and recall were in the 16-19% and 25-42% range, respectfully, for the benchmark programs. The low accuracy suggests that while the models may work well for some programs, the models do not work well for others. Additionally, the alert type and alert location together and in isolation may not be the best predictors of actionable alerts.

Ruthruff et al. [17] screened 33 ACs from 1,652 alerts sampled from Google's code base (spanning multiple projects) to develop logistic regression models for predicting actionable and unactionable alerts. Ruthruff et al. describe a screening process whereby ACs were selected for the model. The generated models contained 9-15 ACs and had an accuracy ranging from 71-87%. Ruthruff et al. [17] compared their generated models to a linear regression model containing all ACs and models developed by Bell et al. [4, 15] for predicting the number of faults. Overall, the models generated by Ruthruff et al. generally had a higher accuracy than the other models. Additionally, the time to gather the data to build the generated model was substantially shorter than the time to build the model with all ACs. Many of the ACs suggested by Ruthruff et al. are used in our research in addition to other project specific metrics. We also consider additional machine learners and built models for individual applications.

Kim and Ernst [10, 11] describe two static analysis alert prioritization techniques that utilize data mined from source code repositories. The first technique uses the average lifetime of alerts sharing the same type to prioritize the alert types [10]. The lifetime of an alert is the time (in days) between alert creation and alert closure. Kim and Ernst assumed that alert types with shorter lifetimes have a higher ranking (e.g. alerts fixed quickly are likely important).

The second technique is a history-based alert prioritization that weights alert types by the number of alerts closed by fault- and non-fault-fixes. A fault-fix is a source code change where the developer fixes a fault or problem and a non-fault-fix is a change where a fault is not fixed, like a feature addition [11]. Alerts may be closed during any code modification, and are therefore considered actionable, but Kim and Ernst expect that those alerts closed during fault-fixes are more important when predicting actionable alerts.

The history-based alert prioritization presented by Kim and Ernst [11] improves the alert precision by over 100% when compared to the precision when alerts are prioritized by tool severity. However, the precision ranged from 17-67%, possibly due to alert closures lacking a causal relationship with the root cause of an anomaly-fix. We include the alert lifetime, measured in revisions instead of days, as a candidate AC. We also utilize source code repository mining for other ACs. Unlike Kim and Ernst, we are interested in classifying individual alerts rather than the alert type.

Williams and Hollingsworth [19] created a static analysis tool which evaluates how often the return values of method calls are checked in source code. A method is flagged with an alert when the return value for the method is inconsistently checked in calling methods. Williams and Hollingsworth use the HISTORYAWARE prioritization technique to prioritize methods by the percentage of time the return value for the methods are checked in the software repository and the current version of the code. The results show a FP rate of 70% and 76% when using the HISTORYAWARE prioritization technique on two case studies involving httpd[1] and Wine[2] applications, respectively. The HISTORYAWARE technique mines data from the source code repository, which we also do, but for different

[1] http://httpd.apache.org/
[2] http://www.winehq.org/

ACs. Instead of using alert type specific information to identify actionable alerts, we use ACs that can prioritize or classify many alert types.

Kremenek et al. [12] show that static analysis alerts in similar locations tend to be homogeneous. On average, 88% of methods, 52% of files, and 13% of directories with two or more alerts contained homogeneous alerts. Kremenek et al. created a FEEDBACK-RANK algorithm whereby the developer's feedback is used to prioritize the remaining alerts. The static analysis tools used by Kremenek et al. take advantage of understanding where the tool checked for an alert, but did not find a potential anomaly [13]. Kremenek et al. [12] prioritize the alerts via a Bayesian Network [20].

2.2. Machine learning

Machine learning "is the extraction of implicit, previously unknown, and potentially useful information about data" [20]. As discussed above, Kremenek et al. [12] used a Bayesian network and Ruthruff et al. [17] used logistic regression, both machine learners, to predict alerts. However, these are not the only machine learning techniques that could be applicable to FP mitigation. The following research discusses machine learning in the context of similar software engineering problems: identifying or predicting latent faults.

Brun and Ernst [5] present a technique that builds support vector machines and decision tree machine learners to classify and prioritize dynamic program properties by the likelihood the property is fault-revealing. Overall, Brun and Ernst found that the support vector machine learner increased the relevance of important program properties by 50 times for C programs and 4.8 times for Java programs. Brun and Ernst [5] suggest their program analysis technique of training and applying machine learners to identify program faults is applicable to static analyses. We extend their technique into a process for applying machine learning to identifying both important ACs and actionable alerts, which also includes AC selection.

Song et al. [18] use the association rule mining machine learner to find attribute patterns that are predictive of latent defects in the source code similar or related to previously-found defects. Song et al. also use association rule mining to predict the effort to find and fix the defect. Defects were predicted with an accuracy of 96.6%. The defect isolation and fix efforts were compared with other machine learning techniques: PART, C4.5, and Naïve Bayes. The association rule mining technique for defection isolation effort prediction had an average accuracy of 93.9% and was about 25% higher than the other machine learners. The association rule mining technique for defect fix effort prediction had an average accuracy of 94.7% and was around 23% higher than the other machine learners. Unlike Song et al. we are only using machine learning to identify actionable static analysis alerts, and not the effort to fix them since the data is currently unavailable. While we are interested in potentially identifying rules, other machine learning techniques may be more applicable to identifying actionable alerts.

Arisholm et al. [1] compared models built by various machine learners to predict locations in industrial Java software that are likely to contain faults. Arisholm et al. found that there were not many significant differences between the precision, recall, and the area under receiver operation characteristic (ROC) curves for the generated models. However, they did find that the cost associated with the generated models were significantly different. We are paralleling their research by evaluating machine learners for predicting actionable static analysis alerts.

3. Alert characteristics

False positive mitigation techniques have used several ACs, like alert type [10-12, 17], priority [11, 17], and the history of code changes [17, 19] to predict or prioritize actionable alerts. We have defined 51 candidate ACs that apply to each alert. These ACs come from three sources: a static analysis tool, a metrics tool, and the source code repository. The ACs are presented in five categories in Sections 3.1 through 3.5. References are provided where ACs have been used in other FP mitigation models, and explanations about AC generation are presented where the name may be unclear. For a more detailed explanation, see [6].

3.1. Alert identifier and alert history

A static analysis tool generates *alert identifiers* (the first eight characteristics below) at alert creation, and the *alert history* (the last characteristic below) is generated via a program that compares the alerts between software revisions. The alert identifier and alert history characteristics are below:

- **Project name**.
- **Package name:** package name could be generalized to the folder containing a source file [7, 8, 12].
- **File name** [7, 8, 12].
- **Method signature:** name and parameter types of the method or function containing the alert [7, 8, 12]. An alert may not have an enclosing method.
- **Alert type:** the type of potential anomaly (e.g. null pointer, etc.) [7, 8, 10, 11, 17, 19].

- **Alert category:** a high level categorization of alert types (e.g. security, correctness) [17].
- **Priority:** the priority of the alert defined by a static analysis tool [10, 17].
- **File extension** [17].
- **Number of alert modifications:** the number of times an alert's line number or priority has been changed over the alert's lifetime.

3.2. Software metrics

Nagappan et al. [14] show that code complexity metrics correlate with failure-prone modules. Additionally, Bell et al. [4, 15] have utilized code size metrics to predict fault counts. Actionable alerts could be considered faults; therefore, software metrics could be predictive of actionable alerts. For characteristics containing several granularities (e.g. method, file, etc.) the metric is collected for each level. The software metric characteristics used for this study are below:

- **Size:** the number of non-comment source statements (NCSS) within the method, file [17], or package declaration containing the alert.
- **Number of methods:** collected at the class and package levels.
- **Number of classes:** collected at the file (e.g. inner classes) and package levels.
- **Cyclomatic complexity:** measures the number of paths through a method [16] containing an alert. Ruthruff et al. [17] use indentation as a measure of complexity.

3.3. Source code history

The models by Williams and Hollingsworth [19], Kim and Ernst [10, 11], and Ruthruff et al. [17] use ACs obtained from a project's source code repository to predict actionable alerts. Instead of recordings the dates of a change, we use revisions. A *revision* is a set of changes committed to the source code repository together. For all of the characteristics listed below, except developers, we record the revision number. Below are the source code history characteristics:

- **Alert open revision** [10].
- **Developers:** set of developers who made changes to the file containing an alert between the alert's open revision and the prior revision analyzed [10].
- **File creation revision** [17].
- **File deletion revision**. Alerts closed due to a file deletion are not considered actionable [10, 11, 17]. These alerts are removed if the file deletion revision is less than or equal to the closure revision.
- **Latest modification revision**: last modification to a file, package, or project on or before the last revision.

3.4. Source code churn

Source code churn measures the amount of change made to a file, package, or project over time [17]. Each of the general code churn metrics are measured between the prior analyzed revision and the open revision for the alert. The churn metrics are measured for the file, package, and project that contain the alert. The source code churn characteristics are below:

- **Added lines** [17].
- **Deleted lines** [17].
- **Growth:** the difference between added and deleted lines [17].
- **Total modified lines:** the sum of added and deleted lines [17].
- **Percent modified lines:** percent of total modified lines out of all modified lines for the project [17].

3.5. Aggregate characteristics

Aggregate candidate ACs come from the above ACs and provide a deeper understanding about an alert. Prior models measure age in days [10, 17]. Instead, we measure age as the number of revisions between two revisions. Using revisions is still a measure of time, but also provides a measure of work. The aggregate characteristics are below:

- **Total alerts for revision:** number of alerts identified on or before an alert's open revision. If an alert is opened and 10 other alerts already exist, the number of alerts for the revision is 11.
- **Total open alerts for revision:** number of open alerts identified on or before an alert's open revision. Continuing with the above example, suppose that three of the 10 existing alerts are closed. Therefore, the number of open alerts for the revision is eight.
- **Alert lifetime:** the age of the alert [10]. For a closed alert, the alert lifetime is the difference between the close and open revisions. Otherwise, the lifetime is the difference between the last revision in the study and the open revision.
- **File age:** the age of the file [17]. For a deleted file, the file age is the difference between the deletion and creation revision. Otherwise, the file age is the difference between the last revision in the study and the file creation revision.

- **Alerts for an artifact:** the number of alerts in the method [7, 8], file [7, 8, 17], package [7, 8], or project [17] containing an alert across all revisions.
- **Staleness:** amount of time between last revision and the last change of the file, package, or project [17].

4. Model building process

Witten and Frank [20] outline a strategy for using machine learning to find patterns in data. For the rest of this section, we describe the process we used, based on Witten and Frank's strategy, to gather data, select ACs, create, and evaluate machine learning models. Teams can use the proposed process to build FP mitigation models using their development history. Periodically, the team can use the process to rebuild models using their most recent development activity.

4.1. Data Collection

There are four steps of data collection required to gather all of the defined ACs: 1) generate the subject revision history; 2) the subject build process; 3) alert classification; and 4) AC generation.

4.1.1. Generate subject revision history. The history for the subject programs comes from source code repository, like CVS[3] or SVN[4]. If the subject program does not have a source code repository, the release history can be used instead. For projects with a large revision history, using a subset of revision can reduce the analysis time.

4.1.2. Subject build process. For each revision in the full or subset history, we check out and build the project and any associated projects required for a complete build. If the project does not build, we move on to the next full or subset revision. Projects that do not build provide inconsistent static analysis data. After building the project(s), we gather size and complexity metrics and static analysis alerts using tools appropriate to the programming language and environment.

4.1.3. Alert classification. The alert classification as actionable or unactionable is our dependent variable in machine learning. An analysis program classifies the automatically findable actionable alerts. Starting with the earliest revision, the sets of alerts between two revisions are compared. An alert is identified by the project name, package name, file name, method signature, alert type, and one of either the FindBugs [9] identifier or line number. Alerts within the same revision that share the same identifying details are duplicate alerts and are considered as the same alert within the revision. Comparisons of the alerts between revisions, using the identifying details, classify the actionable alerts for a project.

When iterating through the revisions, an alert is opened if the alert is not in any of the prior revisions [7, 8]. An alert closure occurs when the alert was in a prior revision, but is not reported in a later revision [7, 8]. An alert is reopened if the alert was closed in a prior revision and reported in a later revision. We only consider the last alert closure, if there is one, for identifying actionable alerts.

The classifications of alerts that remain open at the last revision of the source code are unknown. There are two possibilities of classification for these alerts. The first is to have a developer inspect some or all of the open alerts and determine if the alert is actionable or unactionable. By inspecting all of the alerts there is a full oracle. The other option is to classify all of the open alerts as unactionable. The reasoning is that if developers have not fixed the anomaly associated with the alert during the history of the project, the alert may not be important.

Alert classification follows the steps below:

1. If the alert was closed after going through all of the revision history for the project, the alert is actionable.
2. If the alert was closed due to a file deletion, the alert is neither actionable nor unactionable, and is removed from the alert set used in model building.
3. The remaining alerts can be classified via inspection or all can be marked as unactionable.

4.1.4. AC generation. The analysis program continues by generating the AC values for each alert. Because we consider each distinct alert individually, the ACs are specific to that alert though the AC may have the same value as similar alerts (e.g. alerts opened during the same revision will have the same *alerts for revision* value).

4.2. AC selection

AC selection is important in machine learning because redundant and irrelevant characteristics reduce classifier performance [20]. Additionally, there could be diminishing returns whereby an AC contributes so insignificantly to a model that the time for collection of an AC outweighs the small increase in predictive power. We want to choose the best subset of candidate ACs to use when classifying alerts as actionable or not. Attribute selection algorithms identify the ACs that are

[3] http://ximbiot.com/cvs/wiki/
[4] http://subversion.tigris.org/

associated with the alert's classification, and any algorithms appropriate to the data under analysis can be applied.

4.3. Machine learning model creation

After the ACs are generated for each alert and the alert oracles are supplied, machine learning is applied to the alert set to generate models for predicting actionable and unactionable alerts. When building models we do ten, ten-fold cross validations [20]. In cross validation, the set of alerts are randomly separated into ten approximately equal sets, and nine of the sets train the model that is tested by the last set. Each of the ten sets is a test set, and the process is repeated ten times.

4.4. Model evaluation

We are interested in three metrics to evaluate how well each machine learner performs: precision, recall, and accuracy [7]. *Precision* is the percentage of alerts classified as actionable that were actionable. *Recall* is percentage of alerts classified as actionable out of all actual actionable alerts. *Accuracy* is the percentage of correct actionable and unactionable classifications.

5. Research methodology

The goal of our research is to reduce the number of unactionable static analysis alerts requiring inspection by a developer. We hypothesize that the important ACs and machine learning algorithms will vary by project. Therefore, we want to use the proposed model building process to identify and compare the best ACs and models for two subject programs. The remainder of this section outlines the research methodology for using the model building process on our subject programs.

The subjects programs, `jdom` and `runtime`, are part of the FAULTBENCH v0.1 [7] benchmark. FAULTBENCH contains a suite of subject open-source programs written in Java, static analysis alert oracles for the last revision of the program, and repeatable procedures for evaluating FP mitigation techniques. The alert oracles are stored in spreadsheets that identify the alerts generated by FindBugs [9] and are classified via inspection of the alert by Heckman [7]. Demographics about the subjects used in this study are in Table 1. We use the FAULTBENCH alert oracle and evaluation metrics to compare different machine learners for FP mitigation over the history of the subject programs.

Table 1: Subject programs

	jdom	**runtime**
Domain	data format	software dev.
Size (LOC)	9035-13146	2066-15516
Time Frame (mm/dd/yy)	05/27/00 – 11/22/07	05/02/01 – 08/07/08
# of Revisions	1165	1324
# Sampled Revisions	48	54
# Built Revisions	29	41
Total Alerts	420	853
Actionable Alerts	163	756
Unactionable Alerts	215	36
Deleted Alerts	42	65

The history of each subject program was obtained by mining the source code repository. Evaluation of each revision provides the most accurate alert history; however, there were over 1000 revisions for both projects. We only evaluated every 25th revision starting with the first revision. Additionally, the last revision was included.

Our subject programs are written in Java, and data collection consisted of batch scripts that would check out each revision from CVS, build the revision, and run the JavaNCSS[5] metrics tool and the FindBugs [9] static analysis program. The `jdom` project contained build scripts, while `runtime` was built using a headless Eclipse[6] build process. The metrics and alerts were recorded in .xml files.

Weka [20] is used for AC selection and model building. Weka [20] is a free machine learning tool developed by Witten and Frank at the University of Waikato in New Zealand. Weka contains standard machine learning algorithms for attribute selection and model building using cross-fold validation. We are interested in identifying one or more ACs that are correlated with an alert's classification. However, AC sets with more than 15 ACs were not considered. Too many ACs may reduce the predictive power of a model and the data collection and model building times may increase with additional ACs [20].

We considered three search strategies for selecting ACs [20]: *BestFirst, GreedyStepwise,* and *RankSearch.* All three algorithms finish quickly. BestFirst and GreedyStepwise add ACs as they increase the predictive power of the set. RankSearch evaluates each AC individually and returns the best ACs. For RankSearch, we evaluated the ACs by the information

[5] http://www.kclee.de/clemens/java/javancss/

[6] Eclipse is an integrated development environment with incremental build. A headless version of Eclipse runs without the user interface.

gained from that AC and the ratio of the information gain to the number and size of possible AC values.

The attribute subsets were evaluated using three algorithms [20]: *CfsSubsetEval, WrapperSubsetEval,* and *ConsistencySubsetEval*. *CfsSubsetEval* [20], identifies subsets of ACs that are highly predictive but unrelated to or independent of the other ACs in the set. The second attribute selection algorithm, *WrapperSubsetEval* [20], uses a machine learner and cross-validation to choose the best attribute set. We used the J4.8 decision tree machine learner, which is based on the C4.5 algorithm, because the learner can mimic the decisions a developer may make when inspecting an alert. The third attribute selection algorithm, *ConsistencySubsetEval* [20], finds attribute sets that have consistent class values (actionable or unactionable) within the full set of alerts. Using homogeneous ACs should help classification.

The following machine learning algorithms could classify alerts in ways understandable to developers: classification rules, decision trees, linear models, k-nearest neighbor, and Bayesian networks. Additionally, each machine learning algorithm builds classification models with nominal independent variables. Classification rules provide a set of conditions that, if met, provide the classification for the alert [20]. Decision trees involve tests at each node that lead down different paths of a tree [20]. Linear models work best on numeric data and provide a mathematical equation of the predicted ranking or classification of an alert [17, 20]. Nearest neighbor algorithms investigate the *k* nearest neighbors and weigh the contribution of each neighbor by a distance measure to classify alerts [20]. Bayesian networks are a probabilistic model of the selected attributes [12, 20]. Each machine learner was run with default options in Weka [20] unless otherwise stated.

6. Research results

We hypothesize that the important ACs and machine learners will vary by project. The selected ACs and the best models for the two subject programs are compared to evaluate the hypothesis.

6.1. Selected ACs

The number of ACs selected ranged from four to 13 for `jdom` and from three to 14 for `runtime`. For all attribute subset evaluators, BestFirst and GreedyStepwise identified the same ACs. Overall, both projects had five distinct sets of 15 or less ACs, which demonstrates that there are alert characteristics for both projects that are predictive of actionable or unactionable alerts. Table 2 lists the number of times that an AC is contained in one of the five distinct AC sets for each project.

Table 2: Selected ACs

Alert Characteristics	`jdom`	`runtime`
package name	0	1
file name	1	2
method name	1	2
bug type	0	2
alert category	1	0
file size	1	0
package size	2	0
number of functions in file	1	3
number of functions in package	1	2
open revision	1	3
developer	0	1
file creation revision	3	2
latest file modification	1	3
latest package modification	1	0
package growth lines	1	0
total alerts for revision	2	3
total open alerts for revision	3	3
alert lifetime	5	5
file age	2	0
alerts in file	3	0
alerts in package	2	0
file staleness	1	2

The *alert lifetime* was in every AC subset for both `jdom` and `runtime`, which implies that the length of time the alert is in the code is predictive of the actionablility of the alert. Kim and Ernst [10] hypothesize that alerts with short lifetimes are more likely to be actionable alerts; however, in our data there is not a clear binary split between the lifetime of actionable and unactionable alerts. Some of the alerts with the shortest lifetime were unactionable.

The *file name* and *method name* ACs, as alert identifiers, were selected for both `jdom` and `runtime`. The `runtime` project also contains the *package name* and *bug type* alert identifiers. These ACs were selected either by the ConsistencySubsetEval or the CfsSubsetEval information gain RankSearch, which implies that these ACs tend to be homogeneous within a specific value and have many possible values. While these ACs are project specific and easy to obtain, they may not be the most predictive, especially if the AC value uniquely identifies an alert.

Both projects had AC subsets that contained the *number of functions in file* and *package*. Additionally, the alert's *open revision* and the counts of alerts at each open revision (e.g. *total alerts for revision* and *total open alerts for revision*) were important for both projects. Finally, both `jdom` and `runtime` share several file characteristics like *file creation revision, latest file modification*, and *file staleness*, which most likely follows from the conjecture that there must be a change in a file for either alert creation or closure.

What may be more interesting is what ACs were not included in any AC subset. All but one (e.g. *package growth lines*) of the churn metrics were not included, which is similar to Ruthruff et al.'s [17] findings. Additionally, the *method size*, number of *alerts in method*, and *cyclomatic complexity* were also unimportant, potentially because the method granularity is too low level for an accurate prediction. The *number of classes in file* and *package* level were unimportant while the number of functions was important.

While half of the selected ACs were common between `jdom` and `runtime`, the other half of ACs were different and suggest that there is not a generic set of ACs for all FP mitigation models. Therefore, supporting our hypothesis, AC selection should be project-specific.

6.2. Machine learners analysis

The average precision, recall, and accuracy of the subject programs are presented in Table 3. All of the machine learners and AC subsets have greater than 65% accuracy for `jdom` with an average accuracy of 87.8%. Additionally, the precision and recall for `jdom` were 89% and 83%, respectively. The `runtime` project had a much higher precision, recall, and accuracy at 98%, 99%, and 96.9%, respectively.

Table 3: Average precision, recall, and accuracy

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

These values surpass the precision reported by Kim and Ernst [11]. Many of the individual machine learner, AC subset pairs performed even better than the models presented by Ruthruff et al. [17] suggesting that fewer characteristics are required to obtain good accuracy, and alert classification by project may be more accurate.

For the `jdom` project, 56.9% of alerts were unactionable, while 4.5% of alerts were unactionable for `runtime`. Unlike `jdom`, most of the alerts for `runtime` (95%) were closed by the last revision. We can look at the average confusion matrix [1, 20] for `jdom` and `runtime` in Table 4. The average confusion matrix contains the average values of the classifications for each cross-validation. There were 37-38 alerts tested for each validation run for `jdom` and 79-80 alerts tested for `runtime`. For both `jdom` and `runtime`, there are less than four incorrectly classified instances, which show the models minimize the number of unactionable alerts a developer may inspect while maximizing the number of alerts provided for inspection.

Table 4: Confusion matrix

		Actual classification			
		Actionable		Unactionable	
		`jdom`	`runtime`	`jdom`	`runtime`
Predicted classification	Positive	13.6	74.5	2.0	1.4
	Negative	2.7	1.1	19.5	2.2

The accuracy of the individual machine learners and each AC subset for `jdom` for the selected ACs are presented in Table 5. Each column represents the set of ACs given to the machine learner (by row) [20]. The machine learners are divided by type: the first five are rule based learners; the next four are decision tree learners; simple logistic regression is a linear learner; the following three are k-nearest neighbor learners; and the final two are Bayesian learners. The best model and AC subset pair for `jdom` was KStar and ConsistencySubsetEval with BestFirst search. KStar is a nearest neighbor search meaning that for `jdom`, similar alerts were predictive of new alerts.

The best model and AC subset pair for `runtime` with 98.8% accuracy was LMT and ACs selected with CfsSubsetEval and BestFirst search. LMT is a decision tree with logistic regression equations at the leaves [20]. The only model with an average accuracy over 98% across all alert characteristic sets, was JRip, a rules based learner. The poor performers for `runtime` were Conjunctive Rules at 95.4% accuracy, Bayes Net at 95.8%, and Naïve Bayes at 90.84% accuracy. The best models for the two subject programs differed which supports our hypothesis that FP mitigation models should be project specific.

6.3. Time analysis

An additional consideration in FP mitigation is the time to obtain AC data and train the model. Table 6 presents the time for data collection and model building for the subject programs to the nearest minute. The model building time is the average time for each of the

Table 5: Accuracy results of machine learners on `jdom`

Classifier	Cfs BestFirst	Cfs RankSearch GainRatio	Consistency BestFirst	Wrapper BestFirst	Wrapper RankSearch GainRatio	Average
Decision Table	92.8	91.9	90.5	90.9	91.4	91.5
Conjunctive Rule	79.8	79.7	71.0	66.8	80.3	75.5
PART	91.3	92.8	78.0	91.7	93.3	89.4
Ridor	89.9	90.7	89.5	89.3	90.2	90.0
JRip	91.3	92.8	89.2	88.6	92.2	90.8
ADTree	89.5	91.1	84.5	88.5	90.9	88.9
J48	90.9	92.2	86.2	93.0	92.5	91.0
REPTree	89.5	90.6	81.3	88.4	89.3	87.8
LMT	90.1	92.4	88.9	92.0	92.9	91.2
Simple Logistic Reg.	89.8	90.4	85.8	73.5	93.0	86.5
KStar	92.5	92.5	98.7	92.4	91.3	93.5
LWL	82.8	88.2	72.5	70.4	87.9	80.3
IbK	94.1	93.0	88.1	90.0	93.3	91.7
Bayes Net	89.8	88.0	91.3	88.7	82.7	88.1
NaïveBayes	84.6	83.8	83.9	67.2	80.8	80.1
Average	89.2	90.0	85.3	84.8	89.5	87.8

ten, ten-fold cross validations for each of the selected AC sets across all of the machine learners. The most costly individual model to build for `jdom` and `runtime` was LMT at an average of 25-42 seconds. LMT is a tree with logistic regression functions at the leaves.

Table 6: Time for data collection and model training

Subject	Data Collection (hh:mm:ss)	Model Building (hh:mm:ss)
`jdom`	01:23:23	< 00:00:01
`runtime`	02:16:33	00:00:04

Our data collection times take into account gathering all of the AC data. When considering smaller sets of ACs, the time for data collection will decrease. Larger revision windows would also decrease the time to gather AC data. Additionally, recent alert data could be considered (e.g. history from the past three months) because more recent alert data could predict better than the full project history.

6.4. Threats to validity

There are three main threats to validity for this work: construct validity, internal validity, and external validity. The threat to construct validity is in the measurement and calculations of the ACs. The measurement were based on related work, and modifications occurred when the conditions of the experiment required variation of the measurements. Each calculation is explained briefly in Section 3 and in more detail in [6].

For this research, internal validity concerns how data were gathered and aggregated. A script gathered data for each revision, and a Java program generated each of the ACs. Errors within the script or program could invalidate some of the results. The script was manually tested, while the program has a suite of automated unit test cases to verify the data are read correctly and new data are generated properly.

While the goal of FAULTBENCH is to minimize external validity (generalizability of results) by providing a breath of sample programs, we only evaluated two of the six benchmark programs because they were the largest projects with the longest and most stable revision history. Therefore, our results may not generalize, but additional running of the process on other subjects will minimize this threat to validity.

7. Conclusions and future work

We present a process for using machine learning to create FP mitigation models that consists of 1) AC data collection; 2) AC subset selection; and 3) model building via machine learners; and 4) model selection. We hypothesize that the important ACs and machine learners will vary by project. We found the common ACs for `jdom` and `runtime` are: file name, method name, number of functions in a file and package, alert creation revision, file creation revision, latest file modification, total alerts and total open alerts for a revision, and the alert lifetime. Eleven additional ACs were specific to one of the two projects. The best model for `jdom` was the k-nearest neighbor model, KStar, and for `runtime` was the decision tree model, LMT. *The difference between selected ACs and the best models between projects suggests that FP mitigation models should be project-specific.*

Further work is required to evaluate the generated models to ensure they are not overfit and predictive of future alerts. If these models predict future alerts well, then the models should be evaluated against models proposed in the literature [7, 8, 10-12, 19]. Additional work in finding important ACs and models is required to provide additional evidence to the findings. Finally, the important ACs uncovered by this research could be used to generate more intuitive models from static analysis domain knowledge that may perform better than the models generated from machine learning.

8. Acknowledgements

This research is funded by an IBM PhD Fellowship awarded to the first author for the 2008-2009 academic year. The authors would like to thank the North Carolina State University RealSearch Reading Group for their feedback on this paper.

9. References

[1] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software," *18th IEEE International Symposium on Software Reliability Engineering*, Trollhattan, Sweden, November 5-9, 2007, pp. 215-224.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," in *IEEE Software*. vol. 25, no. 5, 2008, pp. 22-29.

[3] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating Static Analysis Defect Warnings On Production Software," *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, San Diego, CA, USA, June 13-14, 2007, pp. 1-8.

[4] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for Bugs in All the Right Places," *International Symposium on Software Testing and Analysis*, 2006, pp. 61-71.

[5] Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning Over Program Executions," *26th International Conference on Software Engineering*, Edinburgh, Scotland, May 26-28, 2004, pp. 480-490.

[6] S. Heckman and L. Williams, "A Measurement Framework of Alert Characteristics for False Positive Mitigation Models," North Carolina State University TR-2008-23, October 6, 2008.

[7] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," *2nd International Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, October 9-10, 2008, pp. 41-50.

[8] S. S. Heckman, "Adaptively Ranking Alerts Generated from Automated Static Analysis," in *ACM Crossroads*. vol. 14, no. 1, 2007, pp. 16-20.

[9] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, October 24-28, 2004, pp. 132-136.

[10] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," *International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, May 19-20, 2007, p. 27.

[11] S. Kim and M. D. Ernst, "Which Warnings Should I Fix First?," *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 3-7, 2007, pp. 45-54.

[12] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004, pp. 83-93.

[13] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," *10th International Static Analysis Symposium*, San Diego, California, 2003, pp. 295-315.

[14] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *28th International Conference on Software Engineering*, Shanghai, China, May 20-28, 2006, pp. 452-461.

[15] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," *International Symposium on Software Testing and Analysis*, 2004, pp. 86-96.

[16] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. Boston: McGraw Hill, 2005.

[17] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach," *30th International Conference on Software Engineering*, Leipzig, Germany, May 10-18, 2008, pp. 341-350.

[18] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Transactions on Software Engineering,* vol. 32, no. 2, pp. 69-82, February, 2006.

[19] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions on Software Engineering,* vol. 31, no. 6, pp. 466-480, 2005.

[20] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. Amsterdam: Morgan Kaufmann, 2005.