Case Studies of a Machine Learning Process for Improving the Accuracy of Static Analysis Tools

by

Peng Zhao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Static analysis tools analyze source code and report suspected problems as warnings to the user. The use of these tools is a key feature of most modern software development processes; however, the tools tend to generate large result sets that can be hard to process and prioritize in an automated way. Two particular problems are (a) a high false positive rate, where warnings are generated for code that is not problematic and (b) a high rate of non-actionable true positives, where the warnings are not acted on or do not represent significant risks to the quality of the source code as perceived by the developers. Previous work has explored the use of machine learning to build models that can predict legitimate warnings with logistic regression [38] against Google Java codebase. Heckman [19] experimented with 15 machine learning algorithms on two open source projects to classify actionable static analysis alerts.

In our work, we seek to replicate these ideas on different target systems, using different static analysis tools along with more machine learning techniques, and with an emphasis on security-related warnings. Our experiments indicate that these models can achieve high accuracy in actionable warning classification. We found that in most cases, our models outperform those of Heckman [19].

Acknowledgements

First, I would like to extend my sincere gratitude to my supervisor, Michael Godfrey, for his helpful guidance on my thesis.

I am deeply grateful for Andrew Walenstein, Andrew Malton, and Jong Park for their great help and useful suggestions to my thesis.

Special thanks go to my family and friends for their support.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large computer systems invariably have bugs. To combat this problem, many kinds of tools have been designed to analyze software systems in different ways. One class of these tools, known as *static analyzers* or static analysis tools, analyze programs by examining the source code, such as looking for poor programming style and known bad designs, extracting dependency information about program entities such as control flow graphs, and building various kinds of models of the structure of the program source code. These tools are called static because they perform their tasks without actually executing the system. *Findbugs* is an open source example of a static analyzer for Java that is in wide use.

Many companies employ static analysis tools as part of their development process; they may use them for a variety of purposes, such as performing quality assurance or as part of a code review process. In particular, systems that require unusually strong assurance of reliability and trust may use static analysis tools to look for critical bugs and detect known security vulnerabilities. For example, in our studies we have used a commercial tool which we shall refer to as CSAT (Commercial Static Analysis Tool); used by development teams that work on high-security systems, this tool pays special attention to well-known classes of security vulnerabilities, such as the Common Weakness Enumeration database (CWE). [1] Performing a myriad of checks for well-known problems such as *Buffer Overflow*, *Injection*, and *Null Pointers Dereferencing*, CSAT also allows developers to add their own checks to their CSAT installation. For example, Blackberry adds dozens of their own checkers to look out for specific security concerns in their systems.

One of the major problems of existing static analysis tools is the amount of non-automatable work that it takes to get useful results. By their nature, static analysis

[1] A non-disclosure agreement with our industrial partner prevents us from identifying CSAT by name.

tools have only an imperfect idea of the meaning of the underlying code; they tend to assume that all possible problems should be reported, resulting in a high false-positive rate. Furthermore, it is known that off-the-shelf static analysis tools often report perceived problems that developers may not consider worth acting on. For example, in Google [38], for *Findbugs*, only 55% of the warnings are addressed after loging into a bug tracking system. Thus, even if a reported problem is a true-positive, it may not be considered to be *actionable* by the development team. In BlackBerry, fewer than 10% of the warnings from CSAT were acted on by engineers.

Listing 1.1 shows an actionable warning example of demonstration from CSAT that detected in Apache HTTPD 2.2.10. The fix of the warning (bug) in Apache HTTPD 2.2.16 is shown in Listing 1.2.

```
    char kb[MAX_STRING_LEN];
    int i = 0;
    rv = apr_dbm_firstkey(htdbm->dbm, &key);
    if (rv != APR_SUCCESS) {
        fprintf(stderr, "Empty database -- %s\n", htdbm->filename);
        return APR_ENOENT;
    }
    while (key.dptr != NULL) {
        rv = apr_dbm_fetch(htdbm->dbm, key, &val);
        if (rv != APR_SUCCESS) {
            fprintf(stderr, "Failed getting data from %s\n", htdbm->filename);
            return APR_EGENERAL;
        }
        strncpy(kb, key.dptr, key.dsize);
        kb[key.dsize] = '\0';
        fprintf(stderr, "    %-32s", kb);
```

Listing 1.1: An actionable warning that CSAT detected in Apache HTTPD 2.2.10 usage of unsafe String API “strncpy”

The checker that caused the warning is looking for occurences of the C library function `strncpy` which is historically unreliable and not a best practice according to security experts. The function copies a certain number of characters from source to destination. It does not require a terminator at the end of the destination string so it could be susceptible to a variety of exploits such as buffer overflow.

In this case, `strncpy` copies `key.dptr` to `kb` with the length of `key.dsize`. Then \0-terminator is added to the end of `kb` before printing `kb` out. If `key.dsize>=` `MAX_STRING_LEN` then there is a buffer overflow in kb. avoid using kb, so there won't have buffer overflow In Listing 1.2, `printf` prints out characters from `key.dptr` directly without using `db` which avoids buffer overflow.

```
    while (key.dptr != NULL) {
        rv = apr_dbm_fetch(htdbm->dbm, key, &val);
        if (rv != APR_SUCCESS) {
            fprintf(stderr, "Failed getting data from %s\n", htdbm->filename);
            return APR_EGENERAL;
        }
        /* Note: we don't store \0-terminators on our dbm data */
        fprintf(stderr, "    %-32.*s", (int)key.dsize, key.dptr);
        cmnt = memchr(val.dptr, ':', val.dsize);
        if (cmnt)
            fprintf(stderr, " %.*s", (int)(val.dptr+val.dsize - (cmnt+1)), cmnt + 1);
        fprintf(stderr, "\n");
```

Listing 1.2: Fix of the example of the actionable warning in Apache HTTPD 2.2.16 that removed the usage of String API `strncpy`

A non-actionable warning example in Figure 1.1 refers to the return value filename from the function *ap_make_full_path*. The return value filename is not terminated properly. The comments of the function is shown in Listing 1.3 and the code of the function `ap_make_full_path` is shown in Listing 1.4.

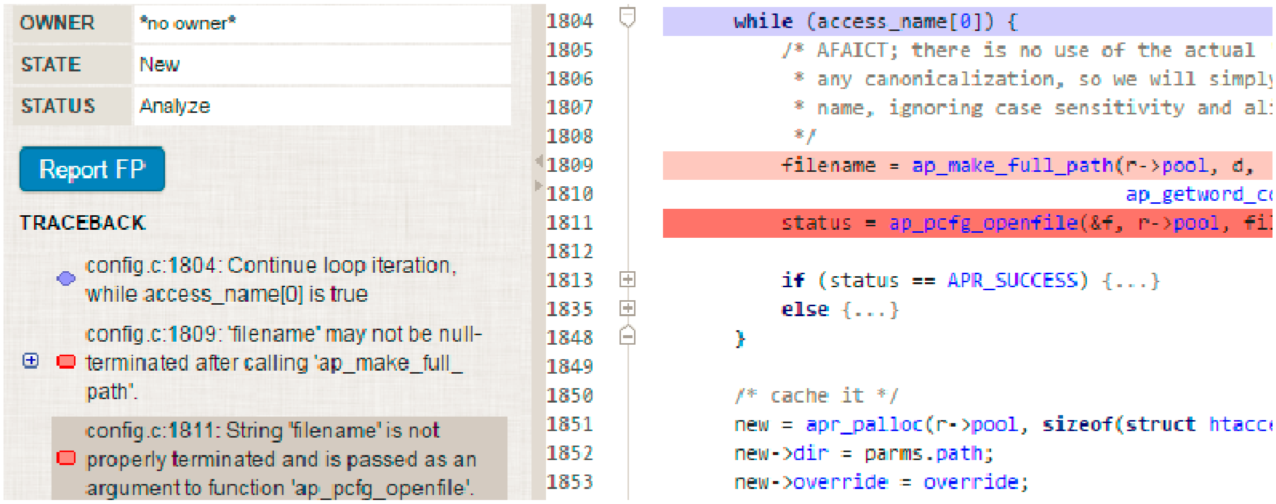

Figure 1.1: A non-actionable warning example from Apache HTTPD. In the example, the return value from function `ap_make_full_path` is not properly terminated.

```
/*
 * @return A copy of the full path, with one byte of extra
    ↪ space after the NUL
 *         to allow the caller to add a trailing '/'.
 * @note Never consider using this function if you are dealing
    ↪ with filesystem
 * names that need to remain canonical, unless you are merging
    ↪ an apr_dir_read
 * path and returned filename.  Otherwise, the result is not
    ↪ canonical.
 */
AP_DECLARE(char *) ap_make_full_path(apr_pool_t *a, const char
   ↪ *dir, const char *f)
                    AP_FN_ATTR_NONNULL_ALL;
```

Listing 1.3: Comment of the function `ap_make_full_path` that caused problems in the non-actionable warning example

```
AP_DECLARE(char *) ap_make_full_path(apr_pool_t *a, const char
   ↪ *src1,
                                      const char *src2)
```

```c
{
    apr_size_t len1, len2;
    char *path;

    len1 = strlen(src1);
    len2 = strlen(src2);
     /* allocate +3 for '/' delimiter, trailing NULL and overallocate
      * one extra byte to allow the caller to add a trailing '/'
      */
    path = (char *)apr_palloc(a, len1 + len2 + 3);
    if (len1 == 0) {
        *path = '/';
        memcpy(path + 1, src2, len2 + 1);
    }
    else {
        char *next;
        memcpy(path, src1, len1);
        next = path + len1;
        if (next[-1] != '/') {
            *next++ = '/';
        }
        memcpy(next, src2, len2 + 1);
    }
    return path;
}
```

Listing 1.4: Code in the function ap_make_full_path that generated the non-actionable warning example

The code and comment of the function ap_make_full_path verify the imperfect design of leaving one byte of extra space at the end of the return value. However, this warning has never been fixed in their later versions of the code, presumably because the developers did not consider it to be a serious risk. As long as a warning has never been acted on, we define the warning as non-actionable.

In this work, we seek to use machine learning and other automated approaches to reduce the rate of non-actionable true-positives of static analysis tools for both industrial and open source projects. We build on previous work by Ruthruff [38] and Heckman [19]. Ruthruff adopted logistic regression on highly associated features against Google's Java codebase, while Heckman enlarged the feature sets from Ruthruff and compared more popular machine learning models with selected features on open source Java projects.

Different from previous work, we explore the static analysis tool CSAT with a security focus on both open source and commercial projects. We improve the usability and value of static analysis tool by classifying actionable warnings. Ordinarily, static analysis tools neither classify nor filter their output. Our Prioritizer project serves to develop techniques to automatically classify as "actionable" and "non-actionable" warnings using machine learning models. These machine learning models are derived from training the models on labeled warnings generated by static analysis tools. One novel contribution of our work is the security focus during the analyses which causes more false positives. A second contribution of this project is the usage of machine learning techniques to handle the extremely high ratio of false positives. A third innovative aspect concerns the generation of the datasets, which consist of features and associated metrics that are extracted from the source code repository and the development context. Features and associated metrics are properties of source code repositories, source code development histories, static analysis tool settings, and static analysis tool warnings histories. Significantly, the code repository, development context, and static analysis tool warnings histories are typically not exploited by static analysis tools. However, the context information shows statistical significance in machine learning models. Some features are strongly correlated with the actionability of warnings. We expect the results of our research to increase the usability of static analysis tools for both open-source and commercial projects by using code contexts information.

Security experts at Blackberry, our industrial collaborator on this project, have selected a set of default settings for the use of CSAT; in part, this is based on CSAT's off-the-shelf default settings. Apart from this set of default settings, security experts added customized BlackBerry security checkers. The regular setting of CSAT enables its critical checkers to detect potential critical bugs in less than an hour for Apache HTTPD. Those customized BlackBerry security checkers cover a variety of known issues, such as the uses of unsafe string API, misuse of `memcpy`, and customized message handlers.

The goal of this research is to improve the accuracy of static analysis tools with a security focus related to actionability. We use supervised machine learning models and feature selection algorithms on source code and extra-code ("context") features. The security emphasis, supervised machine learning techniques for imbalanced datasets, and commercial static analysis tools distinguish our work from previous work. We propose a portable process of building machine learning models with features extracted from customized CSAT analysis and experiment with different projects in the following steps: 1) analyze different versions of source code; 2) extract features and explore strongly correlated feature sets; and 3) compare different classifiers with highly imbalanced datasets including classification, feature selection, and imbalance handling. We conduct experiments and validation on both commercial and open-source projects.

We found that thousands of warnings are generated by our use of CSAT using the target systems of Apache HTTPD, MySQL, and BlackBerry code base. Sampling hundreds of warnings from them, our models classify actionable warnings. In our experiments, models trained with the oversampling datasets correctly classify 90% of the warnings from the datasets according to their actionability. These models achieve 100% recall, which means these models classify all the actionable warnings as actionable. Our experiments improve CSAT performance within BlackBerry and also indicate a method of improving security vulnerability detection.

1.1 Problem

Static analysis tools detect potential critical bugs and generate warnings without executing the code. There are three risks in detecting actionable warnings. First, static analysis tools detect a restricted number of (actionable) warnings but involve two trade offs: false negatives versus false positives and accuracy vesus efficiency. This often results in high false positive rates. Simultaneously, the static analysis tool provides flexibility for users to adjust tradeoffs. It is historically difficult for users to successfully adjust the sensitivity without a fair amount of experimentation. Second, the tools normally output all the generated warnings without any filter or classification. With a high false positive rate, most of the warnings displayed by static analysis tools are not useful for developers. Third, the tools are typically not written to consider reporting warnings based on non-program factors, such as frequency of code update, project type, and project sensitivity, each of which has a significant value in actionable warnings prediction. However, those non-program factors are not included in the scope of most static analysis tools.

1.2 Research Questions

With the problems we came across, table 1.1 lists the research questions that are in scope for this research, and states the approach we will use.

	Research Question	Research Method
RQ1	What features are useful for predicting security vulnerability warnings?	Over-generate features that arguably are correlated to vulnerabilities and use feature selection measures and accuracy observed to rank, evaluate, and select.
RQ2	What statistical models best predict actionable warnings?	Explore supervised machine learning algorithms and analyze their execution time
RQ3	What are the differences between commercial and open-source projects?	Compare commencial and open-source projects with same techniques

Table 1.1: Three Research Questions and Methods of the thesis

1.3 Contributions

There are three main research contributions of this work:

1. A method for generating a predictive statistical model for filtering actionable static analysis tool warnings that improves upon prior work in terms of accuracy and utilizes features not explored in prior research.

2. An analysis of the utility of an extensive set of code and non-code features are predictive of actionability of static analysis tool warnings. This includes measuring and characterizing the utility of introduced features as well as previously used ones (which were never evaluated in this way).

3. A method for selecting classifiers and machine learning techniques for predicting actionable warnings from static analysis tools by combining rule-based (expert) filters with the statistical classifier.

1.4 Organization

The thesis is organized as follows:

Chapter 1 explains the problems static analysis tools are facing. Chapter 3 introduces the background of static analysis tools and security vulnerability standards. Chapter 2 discussed related work in this field. Chapter 4 describes our process and implementation. Chapter 6 and 7 concludes the thesis and assesses threats.

Chapter 2

Related Work

Past research has attempted to resolve concerns of high false positives rate with non-program factors, most commonly with respect to general problem of bug finding, rather than the more specific purpose of discovering security vulnerabilities. Ranking warning categories by the violated checkers based on historical experience (consider no other factors) and using machine learning models [38], [20], and [19] are two strategies that have been well-explored to date.

2.1 Ranking warning categories

The first technique uses history-based code information to rank warning categories in order to prioritize warnings. Based on industrial experience, some categories of checkers detect more severe warnings [22], [23], [43], and [44]. Kim and Ernst [22] proposed a history-based warning prioritization (HWP) for static analysis tool warning categories by mining software change history and warnings. They used keywords to identify commits for three open source programs. They then marked code lines with software change history and warnings. Afterwards, they calculated each warning category weight based on marked code lines — the more often marked code lines have the heavier weight. Finally, the weight was normalized for each category. This technique is most effective when the categories are relatively fine grained. Their technique shows better results than classification based only on warning categories in two out of three projects. However, one limitation of classifying with warning categories is that it is most effective when the checkers in each category have the same importance and weight.

The second technique uses more than purely categories ranking information to prioritize warnings which include code metrics. Heckman [20] proposed an adaptive ranking model that utilizes feedback from developers to rank the likelihood of false positives. This adaptive ranking model considers factors including developers' feedback, historical data from previous releases, and alert type. Boogerd and Moonen [1] presented a technique for computating the likelihood of the code location being executed to rank warnings. The higher the likelihood the code location would be executed, the more important the warning is. Later, Liang [28] and Wu used code location information for identification of "generic-bug-related" lines to label actionable warnings in a training set.

More studies have been done on the relevant factors of predicting warnings. Gall et al. [14] have identified relationships between maintainability of systems and classes and modules. Graves et al. [15] has pointed out that information from change history plays a more important role than file characteristics, Hanam et al. [37] found warnings with similar pattern based on code characteristics to classify actionable warnings, Williams and Hollingworth [43], [44] automatically rank warnings based on information from history stored in CVS repository (context information) and current version of the software (contemporary context information). They rank functions that produce warnings with *historical context information* and *contemporary context information*. *Historical context information* contains functions that include potential bug fix in a CVS commit. *Contemporary context information* shows the frequency of each function's return value tested. They emphasized warnings from static analysis tools would be more related to bugs identified in the software development change history than in metrics based on the code based on their preliminary manual inspection.

2.2 Machine learning models

Ruthruff et al. [38] exploited an open source static analysis tool, FindBugs, with warning category ranking severity based on Google experience to classify true positives and legitimate warnings that would be acted by developers. They sampled thousands of warnings to validate their model and those models showed high accuracy and efficiency. Two logistic regression models in binary classification were constructed for true positives and legitimate warnings against Google Java codebase. In their models, a set of factors and metrics were chosen for static analysis warning and programs based on their experience in Google.

Originally, they collected code complexity metrics based on Nagappan et al. [34]. Those factors were automatically collected, built, and analyzed from source code in enterprise-wide settings. In total, they selected fourteen factors with their screening process. The screening process used partial warnings and source code of 5%, 25%, 50%, and incrementally up to 100% percent of the training datasets for their classifer. The screening process does not decrease the accuracy while improve the efficiency of training classifiers by decreasing the size of the datasets. Their regression model was built with the screening process with selected factors. It worth notice that the accuracy in model construction is important as it is an incremental process with increasing warnings.

Similar to Ruthruff's work [38], Heckman and Williams [19] used machine learning models to classify warnings from FindBugs on two open source projects against Java codebase. Their models were constructed based on a larger origin feature set. They compared more than ten different classifiers combined with feature reduction algorithms. There is no learner or feature set that work for every project based on their study.

Feature sets previous research studied included:

1. Importance of warning category type
2. Life span of defect related files
3. Similarities between defect related files [25] (defects from the same file are likely to be fixed again); all show a positive effect on bug fixing [16].
4. Textual quality of bug reports
5. Perceived customer/business impact
6. Seniority of the bug opener
7. Interpersonal skills of the bug opener

There are 55 more features, apart from these seven possible features, for static analysis warnings and program rankings from Ruthruff [38] and Heckman [20]. The difference between our work and Ruthruff's [38] work is that our work targets in applying more machine learning techniques to classifying warnings with a security focus. Algorithms used to select relevant factors include: BestFirst, GreedyStepwise, RandSearch, and Chi-squared tests. BestFirst and GreedyStepwise add factors as they increase the predictive power of the set. RankSearch evaluates each factor individually and returns the whole set of factors [19]. Chi-squared tests evaluate deduction of different factors with deviance ranges [38]. Most research was done with benchmarks [19] or comparably small data sets that can be evaluated manually [38].

Machine learning algorithms towards relevant factors are likely to promote the ratio of actionable defects. Widely used classification algorithms such as Bayesian Network (BN) [25], Logistic Regression [38], J4.8, are used to prioritize warnings from static analysis tools. Classified categories could be "false positives" (FP) or "true positives" (TP) [23].

2.3 Security warnings

While there has been some previous research studying how static and dynamic analysis tools target security vulnerabilities in code, more work remains to be done. Chess [7] developed a prototype checker that allows finding security flaws in C program. Their prototype checker detects substantial classes of security vulnerabilities while LCLint [12] can detect limited types of problems in C programs.

Liu and Huuck investigated the strengths and weaknesses of the tool *Cppcheck* and *Goanna* [29] on the Android Kernel along with static security checking. *Goanna* has a rate of over 90% TP rate or higher in eleven warning categories while *Cppcheck* has only six categories have a rate of over 90% TP. V.Benjamin and Monica focused on security vulnerabilities in Web applications such as SQL injections [30]. They did not explore more general types of security vulnerabilities such as Null Pointer. Chess and McGraw summarized the pros and cons of utilizing static analysis tools in security vulnerabilities [6]. But the performance of commercial static analysis tools still remains unexplored.

Chapter 3

Background

In this Chapter, before discussing the research methods and experiment design, we discuss technical background and contextual information that is relevant to the industrial security vulnerability standards, static analysis tools, and terms in this thesis.

There are public industrial security vulnerability databases with effective discussion and description of vulnerabilities in source code. One of the popular security vulnerability scoring standards is Common Vulnerability Scoring System (CVSS) [13]. CVSS assigns a severity score to security vulnerabilities by measuring metrics calculated by CVSS Calculator.

Static analysis tools analyze code without execution and generate warnings for potential bugs. The commercial static analysis tool we used in our work — which we refer to as CSAT, as we cannot name it explicitly — has a security emphasis with security relevant checkers and customized security checkers. Among the generated warnings, some of them needed to be acted on by developers later. Those warnings are called actionable warnings in this thesis.

Commercial Static Analysis Tool (CSAT) is a commercial static analysis tool for major programming languages including C and C++. It has more than 200 checkers for C and C++ code. These checkers are divided into more than twenty categories. Some of these checkers map to the recognition of security vulnerabilities reported in CWE. Our industrial partner in this work, Blackberry, has adopted the use of CSAT enterprise-wide. We perform static analysis on one proprietary BlackBerry project and two open source projects sufficiently to classify actionable warnings to display to engineers.

3.1 Static Analysis Tools

Static analysis tools are designed to identify potential flaws, defects, or otherwise undesirable code patterns such as buffer overflow without having to execute programs. Those undesirable code patterns in software systems that are exploited to detect potential vulnerabilities are called *checkers* in CSAT. Checkers in CSAT are defined differently for several major programming languages such as Java and C++, and are based on design properties of the individual languages. The flexibility provided by CSAT allows users to customize checkers to meet specific needs of the users.. These checkers are used to generate warnings which make up data sets for further machine learning models.

CSAT pinpoints the warnings to accurate line numbers of code in source code files (location). Line numbers of code might change for different versions of source code for the same program. Location details determine specific pieces of code in revision history. They are utilized to collect factors from specific pieces of code.

3.1.1 Commercial Static Analysis Tool

CSAT can look for seven categories of potential critical bugs and security vulnerabilities including: Buffer Overflow, Memory and Resources Management, Web application vulnerabilities such as SQL injection, Dereferencing Null pointers, memory allocation errors, etc., for several major programming languages. It has a security emphasis with over 100 of CWE items mapped to CSAT checkers which differentiate it from a lot of static analysis tools such as Findbugs.

CSAT uses checkers as predefined "rules" for code. Different programming languages have different checkers due to the characteristics of programming languages. If there is any potential in violating the rules, warnings are generated by CSAT. For instance, "ABV.ANY_SIZE_ARRAY" is a checker in C++ with C99 style. Array size is determined later in the code when memory is allocated rather than defining with initialization. This can result in a buffer overflow. When there is a potential violation of any enabled checkers, CSAT generates warnings — also called *issues* — accordingly.

Warning states indicate the history and states of a warning from the time it was generated to the time it is fixed. A warning is in one of the three states: "New", "Existing", and "Fixed". A new generated warning status is "New". If the warning still exists in following builds, it state is changed to "Existing". If the warning is fixed, then the warning is marked as "Fixed" in the next build.

CSAT has a good usability with its user interface and further analysis without any changes to the code. Instead of compiling the code with a compiler, CSAT can compile the code and analyze it at the same time or alternatively analyze the bytecode of the program. The CSAT server makes it convenient to check all warnings through a web browser.

3.1.2 Checker

Each checker has an associated severity which is not related to the code base. Most checkers are scored from 1 to 4 with 1 being the critical ones and 4 being review issues based on CSATs experience. Different from traditional static analysis tools like Findbugs, CSAT gives users the flexibility to enable and disable checkers and additional customized checkers.

Configuration in CSAT projects has a setting of active checkers. BlackBerry uses two settings: Regular and Noisy. Regular set enables most of the critical checkers with some other non-severe checkers, as well as part of the BlackBerry customized checkers. Noisy setting enables almost all the checkers including severe and non-severe checkers as well as all of BlackBerry's checkers.

BlackBerry has created over forty customized checkers for security vulnerability detection. These customized checkers include unsafe usage of String API, memory copy, etc. One of the customized checkers is to detect unsafe usage of String API, the function `sprintf` can write past an array and therefore triggers undefined behavior.

3.1.3 Actionable Warnings

In building our training data sets, we treat actionable warnings as warnings that can be fixed pragmatically, and which also are important enough to be fixed given the warning's severity and team's resources [38]. Instead of using the false positive warnings marked by developers, we labelled the data sets with two developers inside of BlackBerry for all the projects used in the experiment.

The analysis projects are built based on modules or projects level which are typically developed by a few teams in organizations. Within the organization, it is hard to guarantee that all the teams would be using the same tool with the same settings. With different development patterns of open source projects and commercial projects, it is even harder for them to have the same frequency of maintenance. Therefore, the difference in development patterns between open source projects and commencial projects leads to different rates of false positives and datasets. In order to generate an unbiased training dataset, we randomly select warnings from the whole warning set for each project and label their actionability with our best judgement.

3.2 Security Vulnerabilities

Security vulnerabilities are security weakness in software that leave the software with potential of being attacked. For example, buffer overflow allows attackers to access or overwrite data in memory if the associated programming language does not has protections.

3.2.1 Security Vulnerability Enumeration

There are some well-known security enumerations such as the National Vulnerability Database (NVD) [35], Common Vulnerabilities and Exposures (CVE) [32], Common Weakness Enumberation (CWE) [33]. They provide standards of weakness and vulnerability description, management, and usage in source code.

CWE provides a set of security vulnerabilities to the public with detailed description and industrial standard scoring. CVE identifies vulnerabilities for common usage. It developed a preliminary classification of vulnerabilities. However, the CVE grouping is too rough for the growing security assessment industry. CWE enriches the code security assessment industry with better classification, analysis and further needs.

CWE also provides hierarchical analysis for NVD and CVE. NVD is a standards-based security vulnerability database that enables security vulnerability management. NVD is built upon CVE entries using CWE with hierarchies. Every entry in NVD has a CVSS severity score and CVSS vector.

3.2.2 Common Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) is a framework that represents the severity of vulnerabilities with metric groups. There are three metric groups: Base, Temporal, and Environmental. Those three metric groups consist of more detailed metric vectors. Those vectors made up for CVSS vector.

Base metrics represents the characteristics that cause the vulnerability. They include exploitability metrics, scope and attack complexity. Exploitability metrics measured whether the attack is through network or local namely Attack Vector (AV), Attack Complexity (AC) of accessing the attack; Privileges Required (PR) before abusing the vulnerability; User Interface which captures if the attacker can exploit the vulnerability without users' participate (UI). Scope (S) refers to the authority granted by computer to compute resources. Impact metrics refers to Confidentiality (C) of limited information access, Integrity (I) as the veracity of resource, and Availability (A) as the loss of confidentiality and integrity.

Temporal metrics include exploit code maturity, remediation level, and report confidence. Exploit Code Maturity (E) measures the likelihood of attacking the vulnerability with the current status. Remediation Level (RL) reflects the urgency of fixing the vulnerability. Report Confidence (RC) shows the exposures of the vulnerability, whether the report describes a confirmed vulnerability with details.

Environmental metrics include security requirements and modified base metrics. Security Requirements enable the customized CVSS score depending on the needs of users. This is measured with metrics of Confidentiality Requirement (CR), Integrity Requirement (IR), and Availability Requirement (AR).

The detailed metrics of base metric, temporal metric, and environmental metric make up the CVSS vector. With defined equations and scores of different levels in the metrics, the CVSS calculator would calculate the CVSS score of the vulnerability.

3.3 Machine Learning

Machine learning builds statistical models of classification or prediction from studying of data. In our case, the dataset are labeled with classification before training, it is called *supervised learning*. If the training is upon data without labeling then it is called *unsupervised training*. The classification of dataset can be binary or multiple classes.

3.3.1 Feature Selection

In machine learning, a feature is a measurement of observation such as total number of lines of code in our project. Feature selection is an automatic process of selecting features that are more relevant to the results. Feature extraction selects a subset of the features and constructs new features from them. Different from feature extraction, feature selection selects existing features instead of constructing new ensemble data factors. It can decrease the complexity of machine learning models and therefore make the models more understandable without information loss like feature extraction.

3.3.2 Imbalance

In a classification problem, the dataset used to train the model has more than one class. If one of the classes takes much bigger part of the dataset, the ratio of different classes can be extreme. For example, if the two classes are positive (90% of the data) and negative (10% of the data), then it would be easy to train a classifier that always returns positive and has an accuracy of 90%, but the classifier would have little practical use failing to detect true negatives. In that case, machine learning classifiers might not be efficient. We have this imbalance problem in our dataset too.

3.3.3 Machine Learning Adoption

We adopt machine learning approaches to classify actionable warnings from all the warnings generated by CSAT. Machine learning features such as lines of code (expressed with integer) in a file are utilized in classification. These features add more information to classifiers that CSAT does not contain. We have way more actionable warnings than non-actionable warnings. We reconstruct datasets and use specific approaches like cost sensitive learning to solve the imbalance problem.

Chapter 4

Methodology

Classifying static analysis warnings based on actionability for large projects, especially in a large industrial organization, can be costly due to the enormous size of the data set with a high proportion of false positives. Our goal is to classify warnings generated by static analysis tools with machine learning models in a cost-efficient way.

As shown in Figure 4.1, we do static analysis on several versions of the source code of three projects and then collect the generated warnings to create data sets for further training. Factors (features) of those warnings in the data sets were extracted from the source code repository and history. Ground truth refers to the facts of which class any given warning belongs to (e.g., actionable versus non-actionable). Ground truth labeling is performed manually by experts. Labelling is based on the existence of a warning over time in CSAT and code changes in the source code. Some of the warnings might be marked as *Fixed* in CSAT without being fixed. With a large amount of features, feature selection is important for classifier performance. Ensemble classifiers are built on selected features.

Our approach uses feature extraction, feature selection, ground truth labeling, and statistical models to predict whether a warning is an actionable warning or not.

4.1 Commercial Static Analysis Tool Analysis

In our work, we use the CSAT tool to perform analyses on two open source projects (Apache HTTPD and MySQL) and one commercial project (BlackBerry internal project) with the regular setting of checkers. We analyzed multiple versions of three different projects — Apache HTTPD, MySQL, and an internal proprietary project from Blackberry

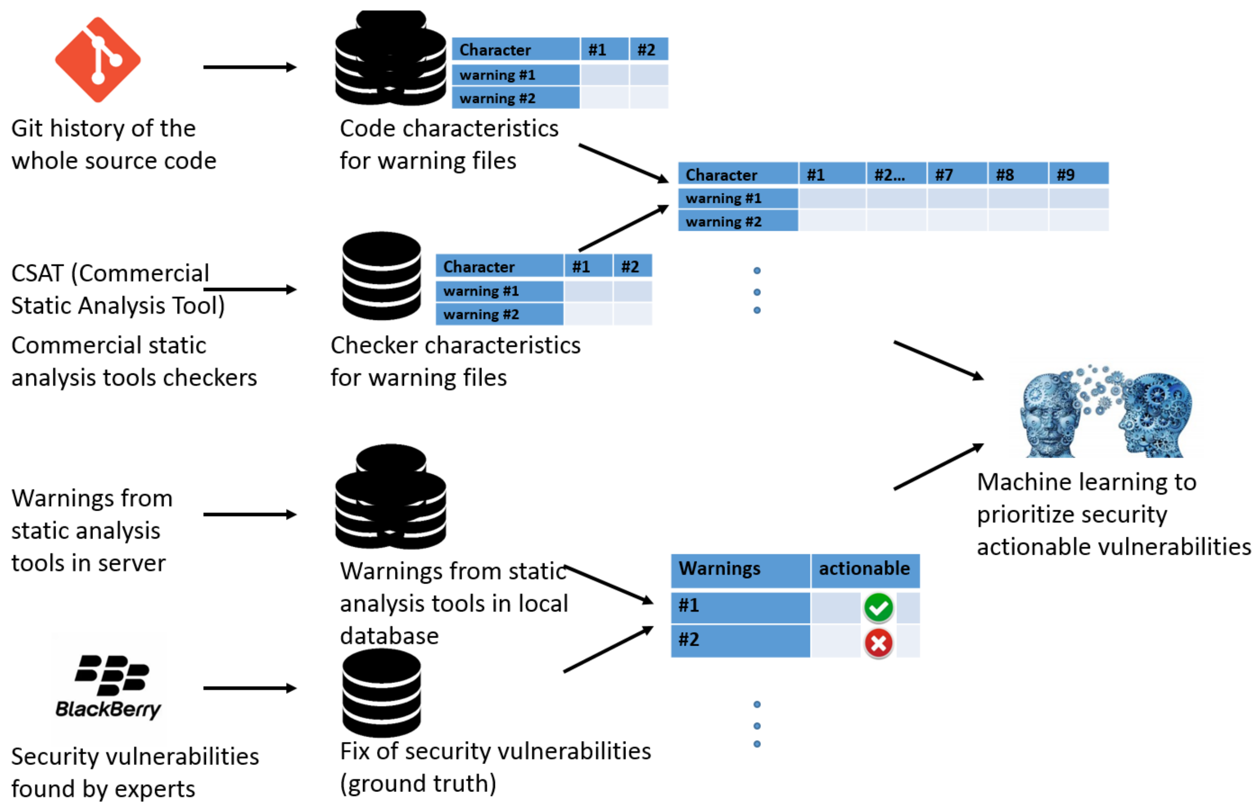

Figure 4.1: Process of feature extraction, feature selection, ground truth labeling, and machine learning in Prioritizer
Source code: commercial projects and open-source projects
Ground truth: label false positives or actionable warnings

— in turn. Five versions of the code from Apache HTTPD and nine versions of the code from MySQL are picked spreading over two years. Three versions of the source code from the proprietary BlackBerry project are picked spreading over half a year. Each of the projects has over 500,000 lines of code. The regular setting satisfies the need for detecting critical vulnerabilities without too many false positives based on experts' experience from BlackBerry. It should be noted that, both MySQL and Apache HTTPD include third party libraries which are not maintained by the project development team. The source code of these third party libraries is included in our analyses for completeness as those third party libraries upgrade with the source code. For the proprietary BlackBerry project, due to the size of the project, only one of the major modules is analyzed by CSAT while all the other modules within the same project are ignored.

4.2 Data Sources and Feature Extraction

We chose our features based on experiences from experts in BlackBerry and other researchers. These features were extracted from four perspectives of the static analysis tools and the code where the warning was detected. First, we extracted some information from the static analysis tool as do the work of Ruthruff [38], Heckman [19], and Kim [22] for warning descriptions. Kim [22] found a strong correlation between warning severity and bugs. Second, work by Bell [36] utilized features from individual files and commit information. Heckman [19] added features from function level, package level, and project level of the warning for predicting actionability. Third, code complexity often leads to more security vulnerabilities in the code [31], [39]. Complicated code has a higher possibility of being attacked. Shin [40] noticed that code complexity is the most important factor among code complexity, code churn (history of code changes), and developer activity in their benchmarks. Fourth, Ruthruff [38] and Heckman [19] both proved relevance of code churn factors with the bug occurrences. Shin [40] proved relevant of code churn factors with security vulnerabilities.

The goal of feature extraction is to remove noisy features and extract more structural features such as code metrics from code repositories and factors from static analysis tools that have good discriminatory value. Code metrics are made up of file characteristics, source code factors, and churn factors. File characteristics include the files age which reflects how long ago the file was created. Source code factors include lines of code which count the total number of lines of code excluding blank lines and comments in a source code file. Churn factors include total changes of code prior to the warning was reported. By replicating previous research and extending it to experiment with more precise machine learning and feature selection algorithms, these features were selected, as shown in Table 4.1.

Commercial Static Analysis Tool issues (warning) descriptor CSAT descriptor has 274 checkers for C/ C++ and 190 checkers for Java.

Code — The code (issue code) refers to the unique abbreviated name assigned to a detected warning type. For instance, *ABV.ANY_SIZE_ARRAY* is the issue code and *buffer overflow-unspecified-sized array index out of bounds* is the description.

Subcode — Subcode refers to the unique abbreviated category name of a code. For instance, *ABV* is the subcode of code *ABV.ANY_SIZE_ARRAY* and stands for buffer overflow for array.

Severity — Each checker has a severity from 1 to 4 with 1 being critical warning and 4 being review warning. The severity shows up with each warning is never changed with warnings status as each checker is bonded with a severity.

File characteristics File characteristics contain the properties of the files where warnings were reported.

File age — The total days of time period between created date of the file and the date the warning was detected.

File programming language — Programming language the file was written in. The programming language is expressed as a number: +1(C/C++)/ -1(Java)

Sha_date_delta_weeks — The total number of the weeks the file was released prior to the warning detection

Directory — The top one, top two, and top three hierarchies of the directories of file path before file name. The top one hierarchy of the directories distinguish source code of open source projects from third party libraries source code. If there are only two directories before the file, then the top three hierarchies would be the same as the top two hierarchies of the directories.

Source code factors Metrics of source code and code characteristics together form source code factors. File length, lines of code (loc), and code indentation [21] show the complexity of code.

Depth — The ratio of the distance from the top of the file to the line of the warning detection compared to the total lines of the file.

File length — Total number of lines in the file including lines of code, empty lines, and comments.

Mean_tab — Mean tabs indented in the beginning of code. If there is not tab in the beginning of code, then mean tab would be zero.

Mean_space — Mean spaces and tabs indented in the beginning of code. If there is not space in the beginning of code, then mean space would be zero.

Churn factors Churn factors consider the history of code changes prior to the warnings detection calculated from development history.

Add_loc — Added lines of code in two weeks, three months, six months, nine months, and twelve months prior to the warning detection.

Del_loc — Deleted lines of code in two weeks, three months, six months, nine months, and twelve months prior to the warning detection.

Fre — Frequency of the file been were touched in two weeks, three months, six months, nine months, and twelve months prior to the warning detection.

Change_total — Number of total lines of code has been changed, which is the sum of added lines of code and deleted lines of code, in two weeks, three months, six months, nine months, and twelve months prior to the warning detection.

Percentage (add_loc_perc, del_loc_perc) — Percentage of added/ deleted lines of code in the past three months compared to loc.

CSAT warnings descriptors	
code	Types of warnings CSAT detect
subcode	Categories of the warnings CSAT detect belong to
severity	Severity of warning types
File characteristics	
file age, create_date_delta_weeks	The length of the time the file existed
file programming language	Programming language of the file
sha_date_delta_weeks	The length of the time the file released
dir	The top hierarchies of the file path
Source code factors	
depth	The ratio of distance of the warning to the total length
indentation (mean_tab, mean_space)	Spaces and tabs indented in the beginning of code
loc	Lines of code
total_lines	Lines of the file
Churn factors: files	
add_loc	Added lines of code prior to the warning detection
del_loc	Deleted lines of code prior to the warning detection
fre	Frequency of modifications prior to the warning detection
change_total	Modified lines prior to the warning detection
percentage (add_loc_perc, del_loc_perc)	Percentage of added/deleted lines of code prior to the warning detection

Table 4.1: Names and explanations of all the features extracted in our project

4.3 Experimental Datasets Labeling

For supervised machine learning models, the datasets require features in machine learning and class that each row (warning) of the datasets belongs to. To have a ground truth to work from, we manually checked the elements in each dataset, and labeled each element with either actionable or non-actionable based on whether the element has been fixed according to the reported warning in later versions of the code. The labeling was performed by two individuals: the author and another employee from BlackBerry. In the case of disagreement, another BlackBerry employee decided on the label. Otherwise, we keep the class of the warnings as labeled.

CSAT marked all the warnings that disappeared in later analysis as “Fixed”, on the assumption that they disappeared from the code because they had been fixed by the development team. However, not all the “Fixed” warnings are addressed by performers regarding the warning reported issue. For instance, if the file where the warning has been deleted, CSAT would still mark the warning as “Fixed”. But in our opinion, this warning is non-actionable. We verified all the “Fixed” warning manually and labelled them with actionability. If a warning is marked as fixed by CSAT, and fixed by performers towards the warning, then we label it as actionable, vice versa.

For each project, a series of CSAT analysis were conducted on 5 to 10 versions of historical code. Datasets generation is very time consuming as we label the actionability of warnings manually. Therefore, we randomly select around 200 warnings from each of the project and label them to form our datasets. For Apache HTTPD, the total amount of warnings is 208 and 199 are selected, the confidence interval is 0.57 when the confidence level is 95% [41]. If the accuracy is 90%, then with 95% confidence level, the accuracy is between 89.43% (90 - 0.57) and 90.57% (90 + 0.57). For MySQL, the total amount of warnings is 1506 and 193 are selected, the confidence interval is 4.43 when the confidence level is 95% [41]. If the accuracy is 90%, then with 95% confidence level, the accuracy is between 85.6% (90 - 4.4) and 94.4% (90 + 4.4). For proprietary BlackBerry project, the total amount of warnings is 3668 and 200 are selected, the confidence interval is 3.07 when the confidence level is 95% [41]. If the accuracy is 90%, then with 95% confidence level, the accuracy is between 86.93% (90 - 3.07) and 93.07% (90 + 3.07).

All the repetitive warnings are removed from the datasets. If a warning is ever marked as fixed by CSAT in any later analysis, we explore the warning by hand and label it as actionable or inactionable. If a warning is never marked as fixed by CSAT in any of the later analysis, we mark the warning as inactionable (not acted on). It is possible that warning is fixed out of our scope of analysis. But it would be a subjective and biased decision to make about actionability by us.

If a warning is addressed due to the reason CSAT reported then it is actionable. Two examples of actionable warning and non-actionable warning from CSAT analysis of MySQL.

```
  /* Don't give sigpipe errors if the client doesn't want them */
  set_sigpipe(mysql);
  if (net->vio != 0)
    len=my_net_read(net);
  reset_sigpipe(mysql);

  if (len == packet_error || len == 0)
  {
    DBUG_PRINT("error",("Wrong connection or packet. fd: %s  len: %d",
                        vio_description(net->vio),len));
#ifdef MYSQL_SERVER
    if (vio_was_interrupted(net->vio))
      return (packet_error);
```

Figure 4.2: An example of a warning that labeled as actionable from CSAT analysis of MySQL

In Figure 4.2, "net -> vio" is checked for "NULL". However, later "net -> vio" is dereferenced without "NULL" checking. Logically, it is possible that "net -> vio" is "NULL" and get dereferenced again. In Figure 4.3, the validity of "net -> vio" is checked before usage. We label it as actionable because the warning is fixed for the potential vulnerability reason as CSAT reported.

```
  if (net->vio != 0)
    len=my_net_read(net);

  if (len == packet_error || len == 0)
  {
    DBUG_PRINT("error",("Wrong connection or packet. fd: %s  len: %lu",
                        vio_description(net->vio),len));
#ifdef MYSQL_SERVER
    if (net->vio && (net->last_errno == ER_NET_READ_INTERRUPTED))
```

Figure 4.3: The fix of the actionable labeled warning from CSAT analysis of MySQL

In Figure 4.4, CSAT reported that "el -> el_line.buffer" is used after have been freed. The function "el_realloc" calls realloc function inherited from C. It doesn not automatically free the variable after calling. Thus the usage of "el -> el_line.buffer" afterwards does not cause the problem CSAT reported. The warning which reported 4.4 is labelled as non-actionable.

```
/*
 * Reallocate line buffer.
 */
newbuffer = el_realloc(el->el_line.buffer, newsz);
if (!newbuffer)
        return 0;

/* zero the newly added memory, leave old data in */
(void) memset(&newbuffer[sz], 0, newsz - sz);

oldbuf = el->el_line.buffer;
```

Figure 4.4: An example of a warning that labeled as non-actionable from CSAT analysis of MySQL

4.4 Machine Learning Approaches

Langley [26] showed that there is no machine learning approach that works in all the fields. Therefore, we experiment with three different approaches so as to get the superior results. We apply machine learning models to the same datasets with three different approaches: machine learning classifiers, feature selection, and machine learning classifiers, as well as machine learning techniques for imbalanced datasets. The datasets consist of all the features from Table 4.1 and labeling of actionable or non-actionable. All the approaches are implemented in Java Weka API [18].

For each of the three approaches, seven machine learning classifiers are trained and evaluated in our study: Random Forest [3], Bayesian Network [2], KStar [8], SVM [4], Conjunctive Rules [18], Decision Table [24], and Logistic Regression [27]. Then the trained classifiers are evaluated with 5-fold cross-validation.

Random Forest [3], BayesNet [2], KStar [8], SVM [4], Conjunctive Rules [18], Decision Table [24], and Logistic Regression [27] are all popular machine learning models for various of tasks.

Random Forest [3] is an ensemble model that combines classification trees with a random bagging step. Originally, the classification trees are constructed by generating nodes that can best split the datasets. In a random forest, instead of using the best split nodes, the random "feature bagging" step generates each node by the best among a subset of nodes. After the construction, the datasets that to be predicted go down each of the trees. Each of the trees gives an result and "vote" for the class.

Bayesian Network [2] is a directed acyclic graphical model that construct the dependencies of a dataset. It can be used for classification especially when features have conditional dependencies.

KStar [8] is a distance measure classification model that uses entropy to get benefits such as missing value. It is an instance-based learner that classify new coming data from classes of previous classified data that are similar to it. The distance measure is the transformation of transforming one instance to another.

SVM [4] mode treats datasets as points in space and generates a trained classification to ensure the gap between classes as far as possible. Four kernel models in SVM are included in our experiments: Linear, Polynomial, RBF (Radial basis function), and Sigmoid. If the trained classification kernel is specified as linear, then the generated trained classification is a linear model that separate different categories.

Conjunctive Rules [18] classifies the class from "AND" rule of features.

Kohavi [24] found that Decision Table works surprisingly well in some cases when datases include continuous features. Decision Table classifier [24] classifies datasets with a simple decision table. It selects a subset of features to generate a schema in the decision table. Then match the new coming data to the exact schema for the class. If no match is found, the majority class is returned.

Logistic Regression [27] measures the relationship of features and class wth a logistic regression. The model is modified in Weka to handle weighted datasets.

4.4.1 Machine Learning Classifiers

In the approach of machine learning classifiers, we train the seven classifiers with datasets that consist all of the extracted features from Table 4.1. Within the datasets, each row of the datasets is labeled with actionable or non-actionable as described in section 4.3. The classifiers are trained for a binary classification and later evaluated with 5-fold cross-validation on the same training datasets.

4.4.2 Feature Selection and Machine Learning Classifiers

In the design of feature selection and machine learning classifiers, we select highly correlated subsets of features first and then apply machine learning classifications to the reduced datasets. There are seven strategies for correlated subset feature selection: CfsSubsetEval [17] and GreedyStepwise, CfsSubsetEval and BestFirst, InfoGainAttributeEval and Ranker, ClassifierSubsetEval and GreedyStepwise, ClassifierSubsetEval and BestFirst, ChiSquaredAttributeEval and Ranker, PrincipalComponent (Principal Component Analysis) and Ranker.

CfsSubsetEval selects features that are correlated with the class with no intercorrelation among them. It excludes redundant features during screening and include the highly correlated features as long as they are not correlated with previous selected features.

InfoGainAttributeEval, Information Gain Evaluation in Weka, measures the features with their contribution to reduce the overall entropy. A good feature reduces the most entropy with the most information gaining. Information, also known as purity, represents the necessary information to classify a row in the datasets.

ClassifierSubsetEval evaluates subset of features on training data and estimates the "merit" of the subset with a classifier.

PrincipalComponent converts a set of features into a set of variables called principal components. In Weka, Principal Component Analysis is in usge with Ranker search. This can reduce the noise and redundancy of features.

GreedyStepwise performs a greedy forward or backward search in the feature set. It starts with no or all features and stops when no more features increase the evaluation. It implements steepest ascent search in Weka. In our experiments, we use greedy forward search.

BestFirst, Best-first search in Weka, searches the subsets of features forward from an empty set or backward from the full set of features. The search algorithm beam search is used in BestFirst in Weka. We use forward in our experiments.

Ranker ranks features by their evaluators such as Entropy.

One of the seven strategies first select a subset of correlated features. Then the new dateset that consist of subset of correlated features and actionablitily labeling is used to train one of the seven classifiers. The classifier is then evaluated with original dataset. In total, there are 49 combinations of seven feature selection strategies multiplied by seven classifiers. The trained classifier is later evaluated with 5-fold cross-validation.

4.4.3 Machine Learning Techniques for Imbalanced Datasets

Within the field of machine learning, there are two major approaches to dealing with imbalanced datasets: tweak the classifiers with cost-sensitive techniques and reconstruct the datasets with resampling techniques. Cost-sensitive techniques, cost-sensitive prediction, and meta-cost are three distinct techniques in cost-sensitive learning. Resampling techniques include resample datasets with over-sampling and under-sampling to balance the ratio of the datasets. Resampling techniques consist of over-sampling with replacement, over-sampling without replacement, under-sampling (such as spread subsample), and Synthetic Minority Oversampling Technique (SMOTE) [5] [11].

In cost-sensitive learning, weights are assigned to different classes in the training datasets. These weights are applied in the machine learning process. Different from cost-sensitive learning, cost-sensitive predicting predicts the class with weights to increase the cost of misclassification. In meta-cost, the training data is reclassified from a bagging approach. The meta-cost uses bagging iteration for reclassifying training datasets and generates one classifier from the training datasets.

In the training of the seven classifiers, the cost matrix used by cost-sensitive learning, cost-sensitive predicting, and meta-cost are the same: one extra cost for false positives and ten extra cost for false negatives. The seven classifiers classify datasets on minimum cost after applying the cost matrix. These trained classifiers are evaluated by 5-fold cross-validation with the same datasets that they are used in training.

Over-sampling technique generates a random subsample of a dataset with or without replacement of instances from the minority class from the original dataset. Under-sampling such as spread subsample reduces the dataset by removing the instances from the majority class (binary classification). In addition, SMOTE shows a combination of over-sampling and under-sampling to achieve a better performance.

The reconstructed datasets from resampling are the same size with the original datasets with 5:1 ratio of majority class compared to minority class. Then the trained classifiers are evaluated with the datasets that are used in training with 5-fold cross-validation.

There is no guarantee that one of the approaches for imbalanced datasets work best for all the projects. Therefore, all the approaches are adopted to experiment and compare for our projects.

4.5 Measures

Three measures are employed in our experiments: accuracy, precision, and recall. Accuracy is preferred because it is the most popular measurement for machine learning models. However, for an imbalanced binary class dataset, if the negative class takes up to 90% of the dataset and the classifier classifies all data into this class, then the accuracy is 90% though there is no true positives. In this case, high accuracy does not stand for high-performance from the perspective of true positives. Thus, precision and recall are both adopted to give a more thorough intepretation of our result.

4.6 Methodology Summary

To classify actionable warnings from a large amount of warnings generated by CSAT, we use statistical models to predict actionable warnings based on code information. For all the warnings generated by CSAT, we extract CSAT warning descriptors, files characteristics, source code factors, and churn factors. We later label these warnings with actionable or non-actionable to have a ground truth to work with. We use these labeled warnings and correlated extracted features to train machine learning models. We adopt machine learning techniques such as feature selection and imbalanced techniques to achieve a better performance in our datasets. The details of our experiements based on our Methodology are shown in Chapter 5

Chapter 5

Experiments

In this chapter, we evaluate our models generated using our approach to detect actionable static analysis warnings on the three target systems described previously. We evaluate each model with accuracy, precision, and recall. We analyze time taken to train the classifiers. Time for running CSAT with the default settings mainly depends on the size of the project to be analyzed. We exclude time for running CSAT from our analysis as that time is a one-time cost for each system version. We compare the models from our methodology against the work of Heckman et al. [19].

5.1 Experimental Data Sets

We generate experiment datasets by selecting around 200 warnings from each project (Apache HTTPD: 199, MySQL: 193, proprietary BlackBerry project: 200). Every instance from these data sets is labeled by employees from BlackBerry. Instances are randomly selected from the first analysis as this would guarantee more fixes from later analysis.

5.2 Experimental Infrastructure

In this chapter, we discuss the infrastructure of our project *Prioritizer*. Prioritizer is made up by performing static analysis, data collection, feature extraction, feature selection and machine learning models as described in Methodology.

5.2.1 Feature Extraction Infrastructure

Our static analyses are performed in C or C++ codebase for two open source project — Apache HTTPD and MySQL — and one commercial project — the proprietary BlackBerry project. We performed static analysis on six versions of Apache HTTPD, ten version of MySQL, and three versions of the proprietary BlackBerry project. More than 200 CSAT checkers and more than twenty BlackBerry customized checkers for C or C++ are utilized in our analysis. More than 3000 unique warnings are generated and stored in our database. All the experiments for three different projects are set up with the same static analysis tools setting. This reveals the difference of commercial projects and open source projects in different performance for static analysis.

CSAT conducts static analysis on source code during compilation and stores warnings in server. It displays all the warnings in server website along with source code. After the analyses, we collect warning information from the CSAT server. To have a better view of the source code, code metrics and change history are measured based on the Git source code repository. File characteristics and source code factors listed in Table 4.1 are extracted from the source code Git repository. Then the datasets are labeled and used as training datasets.

5.2.2 Feature Selection and Machine Learning Infrastructure

It is difficult to predict which features best describe the problem domain. One common strategy is to fit all the features to every classification model and check the model's performance. This design enables models to classify datasets with all the available information, which would help models such as random forest which selects features from the whole dataset. A disadvantage of this design is irrelevant features might provide redundant and bias information for models. However, bias for one project might not be bias for another project. We adopt this strategy to give us a more direct view of our datasets.

Our second design uses a pipelined process of selecting features and training models. In this design, machine learning models are trained with reduced datasets with selected features which might provide a more structural datasets and improve the performance. A weakness of this design is the loss of information from the reduced datasets might make it harder to draw meaningful conclusion from the results. Another weakness is the difficulty of picking up the best feature selection algorithms for different models and projects. There is no universal solution for feature selection algorithms. We use this design and find that some feature selection algorithms work well for one project but not another.

Due to the fact that most of the classifiers are built up based on accuracy, our machine learning classification models suffer from high accuracy with low true positives with extremely imbalanced datasets, namely the ratios of inactionable warnings and actionable warnings are more than 10:1. In order to solve this problem, we adopt cost-sensitive models and resampling techniques in machine learning in our third and fourth designs are designed to solve imbalance problems.

Cost-sensitive learning, cost-sensitive prediction, and metacost are compared separately with all classification models in our third design. Cost-sensitive learning brings cost into the classifiers misclassification during learning. Cost-sensitive prediction, on the other side, use cost to predict class. Making every classifier cost-sensitive is arduous. Instead, metacost [9] wraps a cost-minimizing procedure around the classifiers.

Nonetheless, cost-sensitive models do not always work better than resampling techniques, according to Weiss [42]. Therefore we resample our datasets before training the classifiers in our fourth design; then, we use the resampled datasets to train our classifiers with same settings.

5.3 Threats

Construct validity, internal validity, and external validity are three main threats for our work. The threats to construct validity is the bias in labelling training datasets. The warnings are examined and labelled by two engineers within BlackBerry. Another engineer was involved to make a final decision if there was any disagreement to discriminate the effect of individuals. Apart from this, there is no guarantee that the non-fixed warnings are non-actionable warnings as some of these warnings might be fixed in the future.

In this research, the internal threats concerns the size of datasets. Due to the size of the proprietary BlackBerry project, only three versions of the source code are analyzed. Among all the three projects, around five versions of source code are analyzed which might cause discontinuity among projects. All the scripts and source code for this project are tested manually within the units by the author. Errors within any source code could invalidate some of our results.

We have only selected source code from release versions of open source projects and rather complete versions of proprietary BlackBerry project. There is a high possibility that security vulnerabilities have been mostly found out by tools before and fixed already. Due to these limitations, the goal of this study is to explore more useful techniques in this domain.

Chapter 6

Research Results and Discussion

We first show the selected features and results from our experiments which vary by projects. Then, we compare the results among three different projects especially between open source projects and commercial project. Later we compare our results with Heckman's approach[19] in our datasets. We were unable to re-implement Rutruff's study, as they provided insufficient detail to permit this.

6.1 Results of Machine Learning Approaches

We now describe the results of using three different machine learning approaches in our study of static analysis warnings.

6.1.1 Results of Machine Learning Classifiers

Table 6.1 selects the three projects with best accuracy for each project and non-zero reasonable precision and recall. The precision and recall of the correlated accuracy are given in Table 6.2 and Table 6.3. Accuracies of all the classifiers on all three projects are in Table A.1 in Appendix A. Precisions and recalls of all the classifiers on all three projects are in Table A.2 and Table A.3 in Appendix A.

We use the following definitions:

- *accuracy* is the percentage of all warnings in the dataset that were correctly categorized as actionable or non-actionable, according to our ground truth.

- *precision* is the percentage of actionable warnings that were correctly categorized as such among all the predicted as actionable warnings.

- *recall* is the percentage of warnings correctly categorized as actionable among all the actionable warnings.

As we can see in Table 6.2, the best performing approaches on Apache HTTPD and MySQL are all from classifiers SVM, Random Forest, and KStar (compared in seven classifiers). It should be noted that the kernels for SVM are different for Apache HTTPD (RBF) and MySQL (Linear). The highest accuracy is 89.4% for Apache HTTPD while 85.5% for MySQL.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RK) 89.4% (SK) 89.9%	(LK) 82.4% (SK) 89.6%	(LK) 97.5% (SK) 98.5%
Random Forest	89.4%	88.1%	97.5%
KStar	84.9%	84.5%	98.5%

Table 6.1: Accuracy of highest results of classifiers on Apache HTTPD, MySQL, and proprietary BlackBerry project. The best classifiers are SVM Sigmoid Kernel for all three projects and KStar for the proprietary BlackBerry project. They achieve accuracy of 89.9% for Apache HTTPD, 89.6% for MySQL, and 98.5% for the proprietary BlackBerry project. In the table, RK stands for RBF Kernel, SK stands for Sigmoid Kernel, and LK stands for Linear Kernel.

Figure 6.1, 6.2, and 6.3 give a direct vision of the classifier Random Forest for three different projects. Features that are relevant to the actionability are different in the three projects. For example, *subcode* is relevant to actionability in Apache HTTPD and MySQL but not in the proprietary BlackBerry project.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RK) 42.9% (SK) 0%	(LK) 11.1% (SK) 0%	(LK) 25% (SK) 0%
Random Forest	40%	33.3%	33%
KStar	22.2%	18.8%	50%

Table 6.2: Correlated precision of highest accuracy of best classifiers on Apache HTTPD, MySQL, and proprietary BlackBerry project. The highest precision of actionable warnings classification for Apache HTTPD is 42.9% reached by SVM RBF Kernel. The highest one for MySQL is 33.3% achieved by Random Forest. The highest one for the proprietary BlackBerry project is 50% from KStar. In the table, RK stands for RBF Kernel, SK stands for Sigmoid Kernel, and LK stands for Linear Kernel.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RK) 15% (SK) 0%	(LK) 10% (SK) 0%	(LK) 33.3% (SK) 0%
Random Forest	20%	15%	66.7%
KStar	20%	15%	33.3%

Table 6.3: Correlated recall of highest accuracy of best classifiers on Apache HTTPD, MySQL, and proprietary BlackBerry project. The highest recalls for Apache HTTPD, MySQL, and the proprietary BlackBerry project are 20%, 15%, and 66.7% respectively, achieved by Random Forest. In addition, KStar also gives the highest recall for Apache HTTPD and MySQL. In the table, RK stands for RBF Kernel, SK stands for Sigmoid Kernel, and LK stands for Linear Kernel.

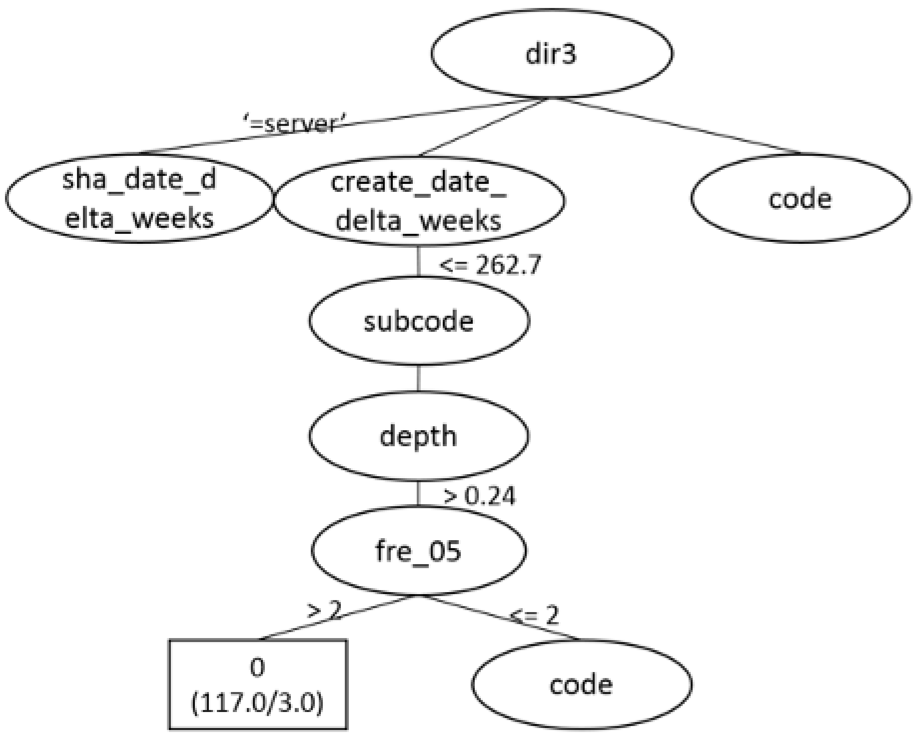

Figure 6.1: Random Forest trained on Apache HTTPD dataset to classify actionable warnings(partial)

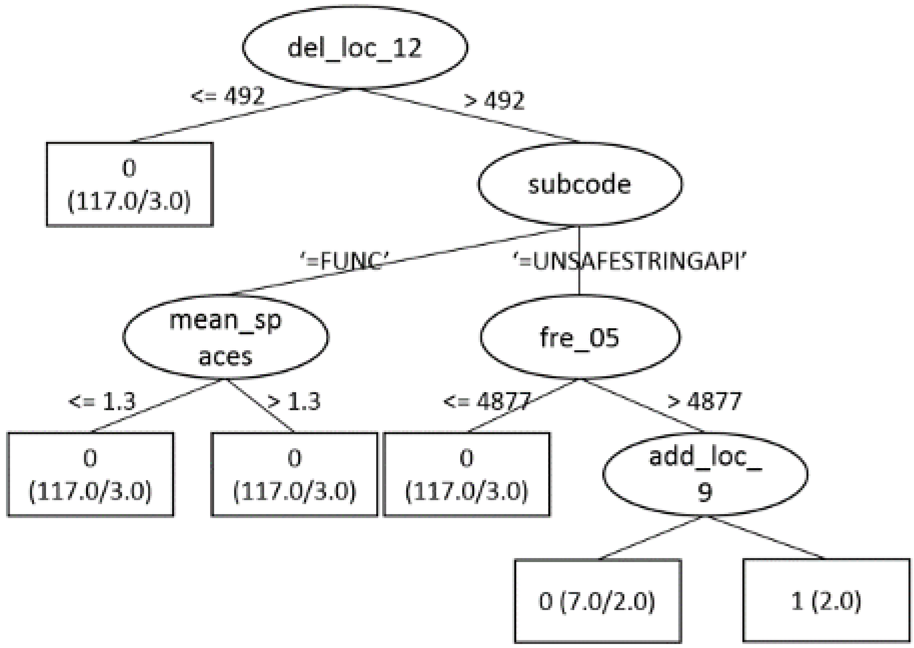

Figure 6.2: Random Forest trained on MySQL dataset to classify actionable warnings(partial)

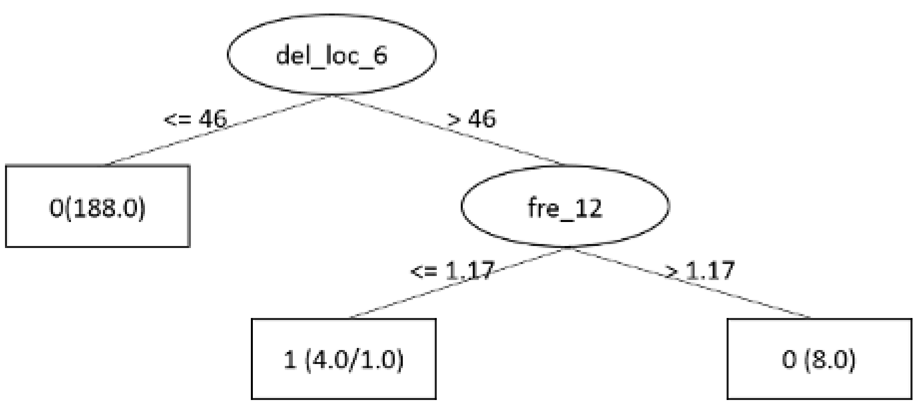

Figure 6.3: Random Forest trained on proprietary BlackBerry project to classify actionable warnings

6.1.2 Results of Feature Selection and Machine Learning Classifiers

We compared seven classifiers with seven feature selection strategies as listed.

Strategy 1 CfsSubsetEval+GreedyStepwise

Strategy 2 CfsSubsetEval+BestFirst

Strategy 3 InfoGainAttributeEval+Ranker

Strategy 4 ClassifierSubsetEval+GreedyStepwise

Strategy 5 ClassifierSubsetEval+BestFirst

Strategy 6 ChiSquaredAttributeEval+Ranker

Strategy 7 PrincipalComponent+Ranker

There are in total 49 combinations (or 70 combinations, if the four SVM kernels counted as four different classifiers) of different classifiers and different feature selection strategies. Showing all of the results would be lengthy and detailed; instead, we present the best performers here. All of the experimental results are listed in Table A.4, A.5, A.6, A.7, A.8, A.9, A.10, A.11, A.12 in Appendix A. For a combination of a feature selection strategy and a classifier, if the accuracy of the evaluation is greater than 75% and the recall is greater than 15%, then the subset of the features is recorded. One subset of features is recorded in one strategy. In total, there are seven subsets of the features. The union of the seven subsets of features are listed in Table 6.4 for all three projects.

Among all the combinations, the highest accuracies of the combinations are listed, as well as the correlated precisions and recalls as shown in Table 6.5, 6.6, and 6.7.

We notice some similarities shared by both open source projects and differences between open source projects and commercial project in our case study. The module and directory where the warnings are found are relevant to their actionability for open source project which is not the case for our proprietary BlackBerry project. Another difference is that the category a warning belongs to plays an important role to actionability for open source projects only. For our proprietary BlackBerry project, the source code change history is crucial to the classification.

Apache HTTPD	MySQL	Proprietary BlackBerry project
dir1	dir1	-
dir2	dir2	-
dir3	dir3	dir3
subcode	subcode	subcode
code	code	code
severity	severity	severity
title	title	-
depth	-	-
-	loc	loc
-	mean_tabs	mean_tabs
-	mean_spaces	mean_spaces
create_date_delta_week	create_date_delta_week	create_date_delta_week
-	add_loc_6	add_loc_3, add_loc_6, add_loc_9, add_loc_12
-	del_loc_6, del_loc_9, del_loc_12	del_loc_3, del_loc_6, del_loc_9, del_loc_12
-	fre_05, fre_6, fre_12	fre_05, fre_3, fre_6, fre_12

Table 6.4: Features that have ever been selected for all three projects with seven feature selection strategies. Checkers' catogeries and severity have a big influence on all three projects. For MySQL and the proprietary BlackBerry project, the code change history is important which is even more so for the proprietary BlackBerry project.

The highest accuracy for Apache HTTPD, MySQL, and proprietary are 89.9%, 89.6%, and 98.5%. However, some of these classifications with the highest accuracy have no true positives classified. The highest accuracy for Apache HTTPD with non-zero true positives is 89.9%, for MySQL is 88.6%, for proprietary BlackBerry project is 98.5%.

Classifier such as Conjunctive Rules have poor performance for all of the three projects with any feature selection approaches. Some classifiers such as KStar and Random Forest work well for all of the three projects and feature selection approaches like ChiSquaredAttributeEval and Ranker. Some feature selection approach combinations such as CfsSubsetEval and GreedyStepwise work well for most classifiers and projects.

6.1.3 Results of Imbalance Techniques

The two most popular approaches to dealing with imbalanced data in machine learning are resampling dataset and cost-sensitive techniques. The results of these imbalance techniques combined with classifiers are shown in this chapter. It is noticeable that classifiers SVM, Random Forest, KStar, and Logistic Regression have high accuracy for our projects.

Classifiers with Resampling dataset

As we can see from Table 6.8, 6.10, and 6.9, Supervised Resampling Replacement dataset works surprisingly well for most of the classifiers for all three projects. It is understandable as our datasets have an extreme ratio of inactionable warnings over actionable warnings. Supervised Resampling resampled actionable warnings to balance out the ratio which helps the classifiers dramatically. All the results of actionable warnings classification with resampled datasets can be found in Table A.13, A.14, A.15, A.16, A.17, A.18, A.19, A.20, and A.21.

Results of Cost-Sensitive Techniques

There are three cost-sensitive techniques: cost-sensitive learning, cost-sensitive prediction, and meta cost. These cost-sensitive techniques are combined with seven different classifiers (classifier SVM has four kernels) on all the datasets. Among them, Table 6.11 lists high accuracy from those combinations for each project. Table 6.12 and 6.13 show the corresponding precision and recall of the combinations in 6.11. All the results of actionable warnings classification with cost-sensitive techniques can be found in Table A.22, A.23, and A.24.

The results are not surprising. There is no combination of classifiers and cost-sensitive technique that work for every project. For the open source projects, the combination of SVM with cost-sensitive prediction has the highest accuracy while the combination of

Classifiers	Apache HTTPD		MySQL		proprietary BlackBerry project	
	Strategy	Accuracy	Strategy	Accuracy	Strategy	Accuracy
BayesNet	3	80.4%	4	89.6%	5	98.5%
	6	80.4%				
DecisionTable	1	89.4%	3	88.1%	4	98.5%
	2	89.4%	6	88.1%		
	3	87.4%				
	6	87.4%				
KStar	1	86.9%	3	85.0%	5	98.5%
	3	84.9%	6	85.0%		
	6	84.9%	7	87.0%		
	7	85.4%				
Logistic Regression	6	80.9%	1	86.5%	3	95.5%
	7	83.9%	2	86.5%	4	98.5%
			7	84.5%	6	95.5%
Random Forest	1	86.9%	3	86.5%	4	98.5%
	3	88.4%	6	86.5%		
	6	88.4%	7	87.0%		
	7	87.9%				
SVM	3 (RK)	89.9%	7 (SK)	88.6%	4 (LK)	98.5%
	3 (PK)	79.9%			1 (PK)	93.5%
Conjunctive Rules	6	89.9%	1	89.6%	5	98.5%

Table 6.5: Accuracy of highest results of feature selection strategies on Apache HTTPD, MySQL, and proprietary BlackBerry project. RK stands for RBF Kernel, SK stands for Sigmoid Kernel, LK stands for Linear Kerneal, and PK stands for Polynomial Kernel. The best feature selection strategies achieve at least 80% accuracy on Apache HTTPD and MySQL (regardless of classifiers) as well as over 95% accuracy for the proprietary BlackBerry project.

Classifiers	Apache HTTPD		MySQL		Proprietary BlackBerry project	
	Strategy	Accuracy	Strategy	Accuracy	Strategy	Accuracy
BayesNet	3	14.8%	4	0.0%	5	0.0%
	6	14.8%				
DecisionTable	1	0.0%	3	28.6%	4	0.0%
	2	0.0%	6	28.6%		
	3	0.0%				
	6	22.2%				
KStar	1	20.0%	3	20.0%	5	0.0%
	3	29.2%	6	20.0%		
	6	29.2%	7	22.2%		
	7	28.6%				
Logistic Regression	6	20.0%	1	20.0%	3	12.5%
	7	20.0%	2	20.0%	4	0.0%
			7	18.8%	6	12.5%
Random Forest	1	25.0%	3	25.0%	4	0.0%
	3	38.5%	6	25.0%		
	6	38.5%	7	27.3%		
	7	30.0%				
SVM	3 (RK)	50.0%	7 (SK)	37.5%	4 (LK)	0.0%
	3 (PK)	25.0%			1 (PK)	8.3%
Conjunctive Rules	6	0.0%	1	0.0%	5	0.0%

Table 6.6: Correlated precision of highest accuracy of best feature selection strategies on Apache HTTPD, MySQL, and proprietary BlackBerry project. RK stands for RBF Kernel, SK stands for Sigmoid Kernel, LK stands for Linear Kerneal, and PK stands for Polynomial Kernel. SVM with different feature selection strategies achieves highest precisions for both Apache HTTPD and MySQL which are 50.0% and 37.5%. Logistic regressio with two different feature selection strategies achieves the highest precision which is 12.5Correlated precisions of the highest accuracies of the results with feature selection strategies combined with classifiers for all three projects

Classifiers	Apache HTTPD		MySQL		Proprietary BlackBerry project	
	Strategy	Accuracy	Strategy	Accuracy	Strategy	Accuracy
BayesNet	3	20.0%	4	0.0%	5	0.0%
	6	20.0%				
DecisionTable	1	0.0%	3	10.0%	4	0.0%
	2	0.0%	6	10.0%		
	3	0.0%				
	6	10.0%				
KStar	1	10.0%	3	15.0%	5	0.0%
	3	35.0%	6	15.0%		
	6	35.0%	7	20.0%		
	7	30.0%				
Logistic Regression	6	30.0%	1	10.0%	3	33.3%
	7	20.0%	2	10.0%	4	0.0%
			7	15.0%	6	33.3%
Random Forest	1	15.0%	3	15.0%	4	0.0%
	3	25.0%	6	15.0%		
	6	25.0%	7	15.0%		
	7	15.0%				
SVM	3 (RK)	25.0%	7 (SK)	15.0%	4 (LK)	0.0%
	3 (PK)	50.0%			1 (PK)	33.3%
Conjunctive Rules	6	0.0%	1	0.0%	5	0.0%

Table 6.7: Correlated recall of highest accuracy of best feature selection strategies on Apache HTTPD, MySQL, and proprietary BlackBerry project. RK stands for RBF Kernel, SK stands for Sigmoid Kernel, LK stands for Linear Kerneal, and PK stands for Polynomial Kernel. SVM with different kernels and different feature selection strategies get the highest recall for Apache HTTPD and the proprietary BlackBerry project at 50.0% and 33.3%. For MySQL, KStar with the feature selection strategy consisting of principal and ranker strategy gets the highest recall at 20.0%.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RBF Kernel) SRWR 91.0%	(RBF Kernel) SRR 89.6%	(Linear, RBF, Polynomial Kernel) SRR 100%
Random Forest	SRR 92.0%	SRR 90.7%	SRR 100%
KStar	SRR **93.5**%	SRR 91.7%	SRR 100%
Logistic Regression	SRR 86.9%	SRR **93.8**%	SRR 99%

Table 6.8: Accuracy of the highest results of resampling techniques for Apache HTTPD, MySQL, and proprietary BlackBerry project. SRR stands for Supervised Resampling with Replacement, SRWR stands for Supervised Resampling without Replacement. KStar and Supervised Resampling with replacement technique generates the highest accuracy for Apache HTTPD at 93.5%. Logistic Regression and supervised resampling with replacement generates the highest accuracy for MySQL at 93.8%. SVM, Random Forest, and KStar along with supervised resampling with replacement generate the highest accuracy for the proprietary BlackBerry project at 100%.

KStar with meta cost classifies most actionable warnings for the proprietary BlackBerry project.

6.2 Comparison

We compare our best results with the replication of Heckman's [19] best results.

In Heckman's work, the best results for her projects come from combinations of five different classifiers and four feature selection processes. The best result for *runtime* was from *IBk* with *Wrapper* and *BestFirst Search*. The best result for *jdom* comes from *KStar* classifier with *ConsisitencySubsetEval* and *BestFirst Search*. We include the best result combinations and other high-accuracy combinations (KStar, Decision table, Conjunctive Rules, Random Forest, and IBk classifiers combined with Cfs, RankSearch, and GainRatio/ Wrapper, RankSearch, and GainRatio) to compare with our work.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RBF Kernel) Supervised Resampling without Replacement 62.5%	(RBF Kernel) SRR 87.2%	(Linear,RBF, Polynomial Kernel) SRR 100%
Random Forest	SRR 86.8%	SRR 86.1%	SRR 100%
KStar	SRR 89.0%	SRR 86.3%	SRR 100%
Logistic Regression	SRR 92.6%	SRR 89.4%	SRR 97.2%

Table 6.9: Correlated precision of the highest accuracy of resampling techniques for Apache HTTPD, MySQL, and proprietary BlackBerry project. SRR stands for Supervised Resampling with Replacement, SRWR stands for Supervised Resampling without Replacement. Logistic Regression and Supervised Resampling with replacement achieve the highest precisions for both Apache HTTPD and MySQL at 92.6% and 89.4%. SVM, Random Forest, and KStar along with Supervised Resampling with replacement achieve 100% precision for the proprietary BlackBerry project.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RBF Kernel) SRWR 25%	(RBF Kernel) SRR 94.1%	(Linear,RBF, Polynomial Kernel) SRR 100%
Random Forest	SRR 100%	SRR 98.0%	SRR 100%
KStar	SRR 100%	SRR 100%	SRR 100%
Logistic Regression	SRR 95.2%	SRR 100%	SRR 100%

Table 6.10: Correlated recall of the highest accuracy of resampling techniques for Apache HTTPD, MySQL, and proprietary BlackBerry project. SRR stands for Supervised Resampling with Replacement, SRWR stands for Supervised Resampling without Replacement. All three projects' recalls could reach 100% from KStar and Supervised Resampling with replacement.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RBF Kernel) cost-sensitive prediction **91.5**%	(Linear Kernel) cost-sensitive prediciton **85.0**%	(Linear Kernel) cost-sensitive learning 97%
Random Forest	cost-sensitive learning 77.4%	cost-sensitive learning 80.8%	cost-sensitive prediction 97.5%
KStar	cost-sensitive prediction 84.4%	cost-sensitive prediction 83.9%	meta cost **98.5**%
Logistic Regression	cost-sensitive learning 78.9%	cost-sensitive prediction 83.9%	cost-sensitive learning 96.5%

Table 6.11: Accuracy of the highest results of cost-sensitive techniques for Apache HTTPD, MySQL, and proprietary BlackBerry project. The highest accuracies for Apache HTTPD and MySQL are achieved by SVM (different kernels) and cost-sensitive prediction at 91.5% and 85.0%. KStar and meta cost classified 98.5% of the warnings correctly for the proprietary BlackBerry project.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RBF Kernel) cost-sensitive prediction 71.4%	(Linear Kernel) cost-sensitive prediciton 23.5%	(Linear Kernel) cost-sensitive learning 28.6%
Random Forest	cost-sensitive learning 20.9%	cost-sensitive learning 28.2%	cost-sensitive prediction 33.3%
KStar	cost-sensitive prediction 26.1%	cost-sensitive prediction 21.1%	meta cost 50.0%
Logistic Regression	cost-sensitive learning 17.6%	cost-sensitive prediction 29.6%	cost-sensitive learning 16.7%

Table 6.12: Correlated precision of the highest accuracy of cost-sensitive techniques for Apache HTTPD, MySQL, and proprietary BlackBerry project. The highest precision for Apache HTTPD is 71.4% from SVM and cost-sensitive prediction. The precision for MySQL is above 20% regardless of the classifiers or cost-sensitive techniques. The highest precision for the proprietary BlackBerry project is 50.0% from KStar and meta cost.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM	(RBF Kernel) cost-sensitive prediction 25.0%	(Linear Kernel) cost-sensitive prediciton 20.0%	(Linear Kernel) cost-sensitive learning 66.7%
Random Forest	cost-sensitive learning 45.0%	cost-sensitive learning 55.0%	cost-sensitive prediction 66.7%
KStar	cost-sensitive prediction 30.0%	cost-sensitive prediction 20.0%	meta cost 33.3%
Logistic Regression	cost-sensitive learning 30.0%	cost-sensitive prediction 40.0%	cost-sensitive learning 33.3%

Table 6.13: Correlated recall of the highest accuracy of cost-sensitive techniques for Apache HTTPD, MySQL, and proprietary BlackBerry project. Random Forest and cost-sensitive learning achieve the highest recall at 45.0% and 55.0% for Apache HTTPD and MySQL. Random Forest and cost-sensitive prediction as well as SVM and cost-sensitive learning both achieve the highest recall for the proprietary BlackBerry project.

Strategy 1 Cfs, RankSearch, and GainRatio

Strategy 2 Consistency, and BestFirst

Strategy 3 Wrapper, RankSearch, and GainRatio

Strategy 4 Wrapper, and BestFirst

It should be noted that there are a few changes in the feature set due to the difference from properties of programming language and static analysis tools listed below. As there are a large number of results from these experiments, only the results with highest accuracy are shown in Table 6.14, 6.15, and 6.16.

Change 1 Comparing to Java, there is no direct match for the concept *package* in C++. Therefore all the features relevant to package in Heckman's work are removed in our replication (*package name*, *number of methods in package level*, *alerts for an artifact in package level*, *staleness in package level*).

Change 2 Not all the files have classes in C++. So the feature *number of classes* is removed.

Change 3 For each project, all the warnings are picked up from the first revision to guarantee there is time for fixing. Thus, there is no difference in those *alert open revision*, *total alerts for revision*, or *total open alerts for revision*. And there is no prior revision to the open revision. No set of *developers* between the open revision and prior revision.

6.3 Execution Time Analysis

We now discuss the execution time of MySQL datasets in our four processes. With almost same amount of data in each dataset, the execution time for Apache HTTPD and proprietary BlackBerry project are similar. Most of the classifiers take less than one second in classification except SVM Linear Kernel and SVM Polynomial Kernel.

As in Table 6.17 and 6.18, the classification itself costs less than one second for all the classifiers. For the feature selection and resampling technique, most of the classifiers cost less than one second, SVM Linear Kernel and SVM Polynomial Kernel costs around 30

Classifiers	Apache HTTPD		MySQL		proprietary BlackBerry project	
	Strategy	Accuracy	Strategy	Accuracy	Strategy	Accuracy
Decision Table	1	89.4%	1	89.6%	1	98.5%
	2	90.5%				
	3	89.4%				
Conjunctive Rule	1	89.9%	1	89.6%	1	98.5%
Random Forest	1	89.9%	1	89.1%	2	98.5%
	2	89.9%	2	89.1%		
	3	91.0%	3	89.6%		
KStar	1	91.0%	1	89.1%	1	99.5%
	2	90.5%				
IBk	1	91.0%	1	89.1%	1	98.5%
	2	86.9%	3	89.6%		
	3	91.0%				

Table 6.14: Accuracy of the highest results of Heckman's methodology on our dataset. Both Random Forest and IBK achieve the highest accuracies in our dataset for Apache HTTPD (91.0%) and MySQL (89.6%). KStar achieves the highest accuracy for the proprietary BlackBerry project at 99.5%.

Classifiers	Apache HTTPD		MySQL		proprietary BlackBerry project	
	Strategy	Accuracy	Strategy	Accuracy	Strategy	Accuracy
Decision Table	1	0.0%	1	0.0%	1	0.0%
	2	66.7%				
	3	0.0%				
Conjunctive Rule	1	0.0%	1	0.0%	1	0.0%
Random Forest	1	50.0%	1	33.3%	2	0.0%
	2	50.0%	2	33.3%		
	3	75.0%	3	0.0%		
KStar	1	60.0%	1	33.3%	1	100.0%
	2	57.1%				
IBk	1	57.1%	1	33.3%	1	50.0%
	2	28.6%	3	0.0%		
	3	62.5%				

Table 6.15: Correlated precision of highest accuracy of Heckman's methodology on our dataset. Random Forest achieves the highest precisions for both Apache HTTPD (75.0%) and MySQL (33.3%). The highest precision is 100% for the proprietary BlackBerry project with KStar.

Classifiers	Apache HTTPD		MySQL		proprietary BlackBerry project	
	Strategy	Accuracy	Strategy	Accuracy	Strategy	Accuracy
Decision Table	1	0.0%	1	0.0%	1	0.0%
	2	10.0%				
	3	0.0%				
Conjunctive Rule	1	0.0%	1	0.0%	1	0.0%
Random Forest	1	20.0%	1	5.0%	2	0.0%
	2	10.0%	2	5.0%		
	3	15.0%	3	0.0%		
KStar	1	30.0%	1	5.0%	1	66.7%
	2	20.0%				
IBk	1	40.0%	1	5.0%	1	66.7%
	2	20.0%	3	0.0%		
	3	25.0%				

Table 6.16: Correlated recall of highest accuracy of Heckman's methodology on our dataset. IBk achieves the highest recalls for all three projects. Apache HTTPD's highest recall is 40%, MySQL's highest recall is 5%, and the proprietary BlackBerry project's highest recall is 66.7%.

Classifiers	Classification	Feature Selection (Average)	Imbalance Technique (Supervised Resampling With Replacement)	Imbalance Technique (Supervised Resampling Without Replacement)	Imbalance Technique (Spread Resample)	Imbalance Technique (SMOTE)
Random Forest	<1s	<1s	<1s	<1s	<1s	<1s
BayesNet	<1s	<1s	<1s	<1s	<1s	<1s
SVM Linear Kernel	<1s	35.7s	38.6s	18.0s	11.5s	46.4
SVM RBF Kernel	<1s	<1s	<1s	<1s	<1s	<1s
SVM Polynomial Kernel	<1s	39.5s	36.9s	19.5s	19.8s	40.7s
SVM Sigmoid Kernel	<1s	<1s	<1s	<1s	<1s	<1s
KStar	<1s	<1s	<1s	<1s	<1s	2.6s
Conjunctive Rule	<1s	<1s	<1s	<1s	<1s	<1s
Decision Table	<1s	<1s	<1s	<1s	<1s	<1s
Logistic Regression	<1s	<1s	<1s	<1s	<1s	2.0s

Table 6.17: Execution time in classification, average time for one process in feature selection and resampling imbalance techniques for MySQL datasets. Supervised Resampling with replacement oversamples (/add) data from the minority class to construct a new dataset with the same size as original dataset. Supervised Resample without replacement resamples the data without adding data from minority repeatatively. Spread Resample generates a random subset of data. There is no big difference in execution time for the other two projects as those two projects have similar size of datasets.

Classifiers	Imbalance Technique (Meta Cost)	Imbalance Technique (Cost-sensitive Prediction)	Imbalance Technique (Cost-sensitive learning)
Random Forest	<1s	<1s	<1s
BayesNet	<1s	<1s	<1s
SVM Linear Kernel	160.3s	20.2s	29.8s
SVM RBF Kernel	2.5s	<1s	<1s
SVM Polynomial Kernel	260.0s	27.8s	33.8s
SVM Sigmoid Kernel	1.5s	<1s	<1s
KStar	37.6s	1.3s	1.2s
Conjunctive Rule	<1s	<1s	<1s
Decision Table	2.3s	<1s	<1s
Logistic Regression	4.5s	<1s	<1s

Table 6.18: Execution time in cost sensitive imbalance techniques for MySQL datasets. Most of the classifiers and imbalance techniques' execution time are under 1s except for SVM with Polynomial or Linear Kernel or meta cost imbalance technique. There is no big difference in execution time for the other two projects.

seconds. These two SVM classifiers also costs much longer time in meta cost which is over 100 seconds. Except SVM classifiers, all the other classifiers take around one second for cost-sensitive prediction and cost-sensitive learning.

Our data collection takes more than 30 minutes due to the searching in git history. If we increase the feature set size, the data collection time would increase accordingly. The time of interaction with static analysis server is ignorable compared to the other feature collection. It should be noted that data generation time is not considered as it highly depends on the size of the project and complexity of checkers, which need to be performed only once.

6.4 Summary of Experiments

We found the results of our experiments and comparison to be surprising. Among all the approaches — classifiers, combination of feature selection techniques and classifiers, imbalance techniques — we would imagine the combination of feature selection techniques and classifiers similar to Heckman's work get the best results. The selected features are more expressive and less noisy in this way. However, due to the specialty of our extremely imbalanced dataset, imbalance techniques give better classification results for both open source and commercial projects.

There are two advantages of our process compared to Heckman's work. The first one is the enlarged feature set. We track back more details of the source code change history and these details in Table 6.4 played an important role. The second one is the imbalance techniques such as cost-sensitive prediction used in our dataset.

In the imbalance techniques, cost-sensitive prediction and supervised resampling work best for all of our projects. Cost-sensitive prediction and SVM classifier with RBF kernel classified 91.5% for Apache HTTPD, supervised resampling with replacement and KStar classifier classifies 93.5% actionable warnings. Supervised resampling with replacement and Logistic Regression classifier classifies 93.8% actionable warnings for MySQL. It is higher than 89.6% from Heckman's work. Both resampling techniques and Heckman's work can classify 100% of actionable warnings for the proprietary BlackBerry project.

Our models provide some insight into the feature set and imbalance techniques for imbalanced dataset. The improvement in accuracy increases the usability of the static analysis tool in software development. In the future, we would like to experiment on more sophisticated static analysis tools with a larger dataset.

Chapter 7

Conclusion

Static analysis tools exhaustively identify all potential instances of a given class of problems. Unfortunately, static analysis tools also generate comparably a lot of false positives. The goal of this work is to classify those warnings worth acting on to improve the usability and save developers' time.

We aim to increase the usability of Commercial Static Analysis Tool with a security focus by classifying actionable warnings using machine learning techniques. We started by trying to replicate Ruthruff's work [38], exploring the domain of this field and more machine learning techniques. Given the characteristics of our dataset, namely the extremely imbalanced ratio of actionable and unactionable warnings, it is necessary to adopt machine learning techniques with an imbalance focus.

In our experiments, we found that the classifiers work best with cost-sensitive prediction and supervised resampling techniques for our projects. They all reach the accuracy of over 90% in our dataset. Furthermore, our models surpass Heckman's work [19] in open source projects.

In future, we will work to deploy these models in BlackBerry with pilot study to adjust our models to increase the number of actionable warnings displayed to the developers. And we will generalize one of our models so that it works for more projects to save the effort of adjusting models for each project.

References

[1] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 149–160. IEEE, 2006.

[2] Remco R Bouckaert. *Bayesian network classifiers in weka*. Department of Computer Science, University of Waikato Hamilton, 2004.

[3] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[4] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[5] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[6] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[7] Brian V Chess. Improving computer security using extended static checking. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 160–173. IEEE, 2002.

[8] John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using an entropic distance measure. In *12th International Conference on Machine Learning*, pages 108–114, 1995.

[9] Pedro Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 155–164. ACM, 1999.

[10] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[11] Nitesh V. Chawla et. al. Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[12] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: A tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes*, 19(5):87–96, 1994.

[13] FIRST. Common vulnerability scoring system v3.0:specification document. https://www.first.org/cvss/specification-document, 2015. [Online; accessed 4-July-2016].

[14] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 13–23. IEEE, 2003.

[15] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[16] Philip J Guo and Dawson R Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX Annual Technical Conference*, pages 285–292, 2009.

[17] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[18] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[19] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing Verification and Validation*, pages 161–170. IEEE, 2009.

[20] Sarah Smith Heckman. Adaptively ranking alerts generated from automated static analysis. *Crossroads*, 14(1):7, 2007.

[21] Abram Hindle, Michael W Godfrey, and Richard C Holt. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, 74(7):414–429, 2009.

[22] Sunghun Kim and Michael D Ernst. Prioritizing warning categories by analyzing software history. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 27. IEEE Computer Society, 2007.

[23] Sunghun Kim and Michael D Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[24] Ron Kohavi. The power of decision tables. In *8th European Conference on Machine Learning*, pages 174–189. Springer, 1995.

[25] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 83–93. ACM, 2004.

[26] Pat Langley and Herbert A Simon. Applications of machine learning and rule induction. *Communications of the ACM*, 38(11):54–64, 1995.

[27] S. le Cessie and J.C. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[28] 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. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 93–102. ACM, 2010.

[29] Tao Liu and Ralf Huuck. Case study: Static security analysis of the android goldfish kernel. In *International Symposium on Formal Methods*, pages 589–592. Springer, 2015.

[30] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, volume 2013, 2005.

[31] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[32] MITRE. Common Vulnerabilities and Exposures The Standard for Information Security Vulnerability Names. http://cve.mitre.org/cve/, 2016. [Online; accessed 4-July-2016].

[33] MITRE. CWE Common Weakness Enumeration A Community-Developed Dictionary of Software Weakness Types. https://cwe.mitre.org/index.html, 2016. [Online; accessed 4-July-2016].

[34] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[35] NIST. National Vulnerability Database automating vulnerability management, security measurement, and compliance checking. https://nvd.nist.gov/, 2016. [Online; accessed 4-July-2016].

[36] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[37] Improving Actionable Alert Ranking. Finding patterns in static analysis alerts. 2014.

[38] Joseph R Ruthruff, John Penix, J David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350. ACM, 2008.

[39] Bruce Schneier. *Beyond fear: Thinking sensibly about security in an uncertain world.* Springer Science & Business Media, 2006.

[40] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[41] Creative Research Systems. Sample size calculator. http://www.surveysystem.com/sscalc.htm, 2012. [Online; accessed 15-September-2016].

[42] Gary M Weiss, Kate McCarthy, and Bibi Zabar. Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs? *DMIN*, 7:35–41, 2007.

[43] Chadd C Williams and Jeffrey K Hollingsworth. Bug driven bug finders. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 70–74, 2004.

[44] Chadd C Williams and Jeffrey K Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

APPENDICES

Appendix A

Complete results for our experiments

A.1 Measurements for classifiers for all three projects

Table A.1: All the accuracies of actionable warnings classification on three projects with classifiers. The best classifiers achieve around 89% accuracy for open source projects and 98.5% for proprietary BlackBerry project.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM (linear kernel)	81.4%	82.4%	97.5%
SVM (polynomial kernel)	76.9%	73.6%	96.5%
SVM (RBF kernel)	89.4%	89.1%	98.5%
SVM (sigmoid kernel)	89.9%	89.6%	98.5%
Random Forest	88.9%	88.1%	97.5%
BayesNet	77.9%	75.6%	93.0%
KStar	84.9%	84.5%	98.5%
Conjunctive Rules	89.9%	89.6%	98.5%
Decision Table	89.4%	89.6%	97.0%
Logistic Regression	77.9%	81.9%	94.0%

Table A.2: All the precisions of actionable warnings classification on three projects with classifiers. Random Forest achieves the highest precisions for all three projects at 40.0%, 33.3%, and 33.3%

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM (linear kernel)	16.0%	11.1%	25.0%
SVM (polynomial kernel)	11.8%	8.1%	16..7%
SVM (RBF kernel)	42.9%	0.0%	0.0%
SVM (sigmoid kernel)	0.0%	0.0%	0.0%
Random Forest	40.0%	33.3%	33.3%
BayesNet	12.5%	17.1%	0.0%
KStar	22.2%	18.8%	50.0%
Conjunctive Rules	0.0%	0.0%	0.0%
Decision Table	33.3%	0.0%	0.0%
Logistic Regression	10.0%	20.0%	0.0%

Table A.3: All the recalls of actionable warnings classification on three projects with classifiers. The best classifiers for Apache HTTPD including Random Forest which achieves 20.0% for recall. The highest recall for MySQL is achieved by BayesNet at 35.0%. The highest recall for proprietary BlackBerry project is achieved by Random Forest at 66.7%.

Classifiers	Apache HTTPD	MySQL	Proprietary BlackBerry project
SVM (linear kernel)	20.0%	10.0%	33.3%
SVM (polynomial kernel)	20.0%	15.0%	33.3%
SVM (RBF kernel)	15.0%	0.0%	0.0%
SVM (sigmoid kernel)	0.0%	0.0%	0.0%
Random Forest	20.0%	15.0%	66.7%
BayesNet	20.0%	35.0%	0.0%
KStar	20.0%	15.0%	33.3%
Conjunctive Rules	0.0%	0.0%	0.0%
Decision Table	5.0%	0.0%	0.0%
Logistic Regression	15.0%	25.0%	0.0%

A.2 Measurements for feature selection and classifiers for all three projects

For the convinience of reading tables from this section, all the involved strategies are listed as following:

Strategy1: Cfs-SubsetEval+GreedyStepwise

Strategy2: CfsSubsetEval+BestFirst

Strategy3: InfoGainAttributeEval+Ranker

Strategy4: ClassifierSubsetEval+GreedyStepwise

Strategy5: ClassifierSubsetEval+BestFirst

Strategy6: ChiSquaredAttributeEval+Ranker

Strategy7: PrincipalComponent+Ranker

Table A.4: All the accuracies of actionable warnings classification on Apache HTTPD with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. The highest accuracy is 90.5% achieved by SVM RBF Kernel.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	89.4%	89.4%	83.9%	89.9%	89.9%	83.9%	84.9%
SVM Polynomial	44.2%	44.2%	79.9%	10.1%	10.1%	79.9%	86.9%
SVM RBF	89.4%	89.4%	89.9%	10.1%	10.1%	89.9%	90.5%
SVM Sigmoid	89.9%	89.9%	89.9%	10.1%	10.1%	89.9%	89.4%
Random Forest	86.9%	86.9%	88.4%	89.9%	89.9%	88.4%	87.9%
BayesNet	86.9%	86.9%	80.4%	89.9%	89.9%	80.4%	88.9%
KStar	86.9%	86.9%	84.9%	89.9%	89.9%	84.9%	85.4%
Conjunctive Rules	88.9%	88.9%	89.9%	89.9%	89.9%	89.9%	89.9%
Decision Table	89.4%	89.4%	87.4%	89.9%	89.9%	87.4%	89.4%
Logistic Regression	84.9%	84.9%	80.9%	89.9%	89.9%	80.9%	83.9%

Table A.5: All the precisions of actionable warnings classification on Apache HTTPD with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. The highest precision is 100.0% achieved by SVM RBF Kernel.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	0.0%	0.0%	20.0%	0.0%	0.0%	20.0%	8.3%
SVM Polynomial	7.5%	7.5%	25.0%	10.1%	10.1%	25.0%	25.0%
SVM RBF	40.0%	40.0%	50.0%	10.1%	10.1%	50.0%	100.0%
SVM Sigmoid	0.0%	0.0%	0.0%	10.1%	10.1%	0.0%	0.0%
Random Forest	25.0%	25.0%	38.5%	0.0%	0.0%	38.5%	30.0%
BayesNet	0.0%	0.0%	14.8%	0.0%	0.0%	14.8%	25.0%
KStar	20.0%	20.0%	29.2%	0.0%	0.0%	29.2%	28.6%
Conjunctive Rules	33.3%	33.3%	0.0%	0.0%	0.0%	0.0%	0.0%
Decision Table	0.0%	0.0%	22.2%	0.0%	0.0%	22.2%	0.0%
Logistic Regression	0.0%	0.0%	20.0%	0.0%	0.0%	20.0%	20.0%

Table A.6: All the recalls of actionable warnings classification on Apache HTTPD with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. SVM Polynomial kernel achieves 100.0% with strategy 4 and 5.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	0.0%	0.0%	20.0%	0.0%	0.0%	20.0%	5.0%
SVM Polynomial	40.0%	40.0%	50.0%	100.0%	100.0%	50.0%	15.0%
SVM RBF	40.0%	40.0%	50.0%	10.1%	10.1%	50.0%	5.0%
SVM Sigmoid	0.0%	0.0%	0.0%	10.1%	10.1%	0.0%	0.0%
Random Forest	15.0%	15.0%	25%	0.0%	0.0%	25%	15.0%
BayesNet	0.0%	0.0%	20.0%	0.0%	0.0%	20.0%	5.0%
KStar	10.0%	10.0%	35.0%	0.0%	0.0%	35.0%	30.0%
Conjunctive Rules	10.0%	10.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Decision Table	0.0%	0.0%	10.0%	0.0%	0.0%	10.0%	0.0%
Logistic Regression	0.0%	0.0%	30.0%	0.0%	0.0%	30.0%	20.0%

Table A.7: All the accuracies of actionable warnings classification on MySQL with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. A few classifiers achieve 89.6% accuracy with different strategies.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	86.5%	84.5%	80.8%	89.6%	89.6%	80.8%	85.5%
SVM Polynomial	36.8%	34.7%	77.7%	10.4%	10.4%	77.7%	86.5%
SVM RBF	88.1%	88.1%	89.1%	10.4%	10.4%	89.1%	88.6%
SVM Sigmoid	86.5%	86.5%	89.6%	10.4%	10.4%	89.6%	88.6%
Random Forest	88.6%	88.6%	86.5%	89.6%	89.6%	86.5%	87.0%
BayesNet	83.9%	83.9%	68.9%	89.6%	89.6%	68.9%	89.6%
KStar	88.1%	88.1%	85.0%	89.6%	89.6%	85.0%	84.5%
Conjunctive Rules	89.6%	89.6%	89.6%	89.6%	89.6%	89.6%	89.6%
Decision Table	89.1%	89.1%	88.1%	89.6%	89.6%	88,1%	89.6%
Logistic Regression	86.5%	86.5%	79.3%	89.6%	89.6%	79.3%	84.5%

Table A.8: All the precisions of actionable warnings classification on MySQL with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. The highest precision is 33.3% achieved by Decision Table.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	20.0%	14.3%	16.0%	0.0%	0.0%	16.0%	10.0%
SVM Polynomial	7.5%	8.6%	7.4%	10.4%	10.4%	7.4%	12.5%
SVM RBF	0.0%	0.0%	0.0%	10.4%	10.4%	0.0%	0.0%
SVM Sigmoid	0.0%	0.0%	0.0%	10.4%	10.4%	0.0%	37.5%
Random Forest	25.0%	25.0%	25.0%	0.0%	0.0%	25.0%	27.3%
BayesNet	0.0%	0.0%	2.4%	0.0%	0.0%	2.4%	0.0%
KStar	0.0%	0.0%	20.0%	0.0%	0.0%	20.0%	22.2%
Conjunctive Rules	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Decision Table	33.3%	33.3%	28.6%	0.0%	0.0%	28.6%	0.0%
Logistic Regression	20.0%	20.0%	22.2%	0.0%	0.0%	22.2%	18.8%

Table A.9: All the recalls of actionable warnings classification on MySQL with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. A few classifiers achieve 100.0% recall.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	10.0%	10.0%	20.0%	0.0%	0.0%	20.0%	5.0%
SVM Polynomial	45.0%	55.0%	10.0%	100.0%	100.0%	10.0%	5.0%
SVM RBF	0.0%	0.0%	0.0%	100.0%	100.0%	0.0%	0.0%
SVM Sigmoid	0.0%	0.0%	0.0%	100.0%	100.0%	0.0%	15.0%
Random Forest	5.0%	5.0%	15.0%	0.0%	0.0%	15.0%	15.0%
BayesNet	0.0%	0.0%	5.0%	0.0%	0.0%	5.0%	0.0%
KStar	0.0%	0.0%	15.0%	0.0%	0.0%	15.0%	20.0%
Conjunctive Rules	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Decision Table	5.0%	5.0%	10.0%	0.0%	0.0%	10.0%	0.0%
Logistic Regression	10.0%	10.0%	40.0%	0.0%	0.0%	40.0%	15.0%

Table A.10: All the accuracies of actionable warnings classification on proprietary BlackBerry project with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. All the classifiers can achieve more than 90.0% accuracy with different strategies.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	96.5%	96.5%	97.0%	98.5%	98.5%	97.0%	96.0%
SVM Polynomial	93.5%	93.5%	97.0%	1.5%	1,5%	97.0%	96.5%
SVM RBF	98.0%	98.0%	98.5%	1.5%	1.5%	98.5%	98.5%
SVM Sigmoid	98.5%	98.5%	98.5%	1.5%	1.5%	98.5%	98.5%
Random Forest	96.5%	96.5%	96.5%	98.5%	98.5%	96.5%	96.0%
BayesNet	97.5%	97.5%	92.5%	98.5%	98.5%	92.5%	98.5%
KStar	98.5%	98.5%	98.0%	98.5%	98.5%	98.0%	97.5%
Conjunctive Rules	98.5%	98.5%	98.5%	98.5%	98.5%	98.5%	98.5%
Decision Table	98.0%	98.0%	98.0%	98.5%	98.5%	98.0%	98.5%
Logistic Regression	96.0%	96.0%	95.5%	98.5%	98.5%	95.5%	96.0%

Table A.11: All the precisions of actionable warnings classification on proprietary BlackBerry project with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. Logistic Regression achieves the highest precision at 12.5%.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
SVM Polynomial	8.3%	8.3%	0.0%	1.5%	1.5%	0.0%	0.0%
SVM RBF	0.0%	0.0%	0.0%	1.5%	1.5%	0.0%	0.0%
SVM Sigmoid	0.0%	0.0%	0.0%	1.5%	1.5%	0.0%	0.0%
Random Forest	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
BayesNet	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
KStar	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Conjunctive Rules	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Decision Table	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Logistic Regression	0.0%	0.0%	12.5%	0.0%	0.0%	12.5%	0.0%

Table A.12: All the recalls of actionable warnings classification on proprietary BlackBerry project with combinations of seven feature selection strategies and seven classifiers. Among the seven classifiers, SVM has four different kernels. SVM achieves 100.0% recall.

Classifiers	S1	S2	S3	S4	S5	S6	S7
SVM Linear	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
SVM Polynomial	33.3%	33.3%	0.0%	100.0%	100.0%	0.0%	0.0%
SVM RBF	0.0%	0.0%	0.0%	100.0%	100.0%	0.0%	0.0%
SVM Sigmoid	0.0%	0.0%	0.0%	100.0%	100.0%	0.0%	0.0%
Random Forest	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
BayesNet	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
KStar	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Conjunctive Rules	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Decision Table	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Logistic Regression	0.0%	0.0%	33.3%	0.0%	0.0%	33.3%	0.0%

Table A.13: All the accuracies of actionable warnings classification on Apache HTTPD project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. KStar and supervised resampling with replacement achieve the highest accuracy at 93.5%.

Classifiers	Supervised resampling with replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	86.4%	84.4%	76.7%	81.4%
SVM Polynomial	78.9%	83.4%	77.5%	70.4%
SVM RBF	87.4%	91.0%	84.2%	88.4%
SVM Sigmoid	52.8%	89.9%	83.3%	89.9%
Random Forest	92.0%	88.4%	81.7%	79.4%
BayesNet	84.4%	77.4%	67.5%	75.9%
KStar	93.5%	83.9%	82.5%	83.9%
Conjunctive Rules	75.9%	89.9%	83.3%	85.9%
Decision Table	77.9%	88.9%	84.2%	87.4%
Logistic Regression	86.9%	82.9%	79.2%	78.4%

Table A.14: All the precisions of actionable warnings classification on Apache HTTPD project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. Decision Table and spread subsample achieve the highest precision at 100.0%.

Classifiers	Supervised resampling with replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	83.6%	21.1%	34.6%	24.2%
SVM Polynomial	83.2%	24.0%	37.9%	14.5%
SVM RBF	83.3%	62.5%	55.6%	38.5%
SVM Sigmoid	52.8%	0.0%	0.0%	0.0%
Random Forest	86.8%	33.3%	43.8%	4.3%
BayesNet	83.6%	16.2%	22.9%	19.6%
KStar	89.0%	25.0%	47.1%	25.0%
Conjunctive Rules	86.1%	0.0%	0.0%	27.8%
Decision Table	72.3%	25.0%	100.0%	36.8%
Logistic Regression	82.6%	26.7%	40.0%	17.1%

Table A.15: All the recalls of actionable warnings classification on Apache HTTPD project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. A few classifiers such as SVM Sigmoid Kernel with supervised resampling with replacement achieve 100.0% recall.

Classifiers	Supervised resampling with replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	92.4%	20.0%	45.0%	40.0%
SVM Polynomial	75.2%	30.0%	55.0%	40.0%
SVM RBF	95.2%	25.0%	25.0%	25.0%
SVM Sigmoid	100.0%	0.0%	0.0%	0.0%
Random Forest	100.0%	15.0%	35.0%	5.0%
BayesNet	87.6%	30.0%	40.0%	45.0%
KStar	100.0%	30.0%	40.0%	30.0%
Conjunctive Rules	64.8%	0.0%	0.0%	25.0%
Decision Table	94.3%	5.0%	5.0%	35.0%
Logistic Regression	95.2%	40.0%	50.0%	30.0%

Table A.16: All the accuracies of actionable warnings classification on MySQL project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. The highest accuracy is achieved by Logistic Regression and supervised resampling replacement.

Classifiers	Supervised resampling replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	76.2%	79.8%	78.3%	73.6%
SVM Polynomial	71.5%	77.2%	67.5%	77.1%
SVM RBF	89.6%	88.6%	83.3%	87.6%
SVM Sigmoid	52.3%	89.6%	83.3%	89.6%
Random Forest	90.7%	85.0%	82.5%	83.9%
BayesNet	82.4%	75.6%	67.5%	72.5%
KStar	91.7%	84.5%	75.0%	83.9%
Conjunctive Rules	75.1%	89.6%	83.3%	82.4%
Decision Table	83.9%	89.6%	81.7%	85.5%
Logistic Regression	93.8%	77.7%	72.5%	75.6%

Table A.17: All the precisions of actionable warnings classification on MySQL project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. The highest precision is achieved by Logistic Regression and supervised resampling with replacement

Classifiers	Supervised resampling replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	78.9%	14.8%	33.3%	18.4%
SVM Polynomial	76.7%	14.7%	19.4%	17.1%
SVM RBF	87.2%	0.0%	0.0%	25.0%
SVM Sigmoid	52.3%	0.0%	0.0%	0.0%
Random Forest	86.1%	26.3%	46.2%	7.7%
BayesNet	79.6%	6.5%	27.9%	22.0%
KStar	86.3%	18.8%	29.2%	21.1%
Conjunctive Rules	68.5%	0.0%	0.0%	11.1%
Decision Table	80.2%	0.0%	33.3%	25.0%
Logistic Regression	89.4%	15.2%	25.9%	17.1%

Table A.18: All the recalls of actionable warnings classification on MySQL project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. Supervised resampling with replacement and KStar achieve 100.0% recall.

Classifiers	Supervised resampling replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	74.3%	20.0%	30.0%	45.0%
SVM Polynomial	65.3%	25.0%	30.0%	30.0%
SVM RBF	94.1%	0.0%	0.0%	10.0%
SVM Sigmoid	100.0%	0.0%	0.0%	0.0%
Random Forest	98.0%	25.0%	30.0%	5.0%
BayesNet	89.1%	10.0%	60.0%	65.0%
KStar	100.0%	15.0%	35.0%	20.0%
Conjunctive Rules	97.0%	0.0%	0.0%	10.0%
Decision Table	92.1%	0.0%	10.0%	20.0%
Logistic Regression	100.0%	25.0%	35.0%	35.0%

Table A.19: All the accuracies of actionable warnings classification on proprietary BlackBerry project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. A few classifiers with supervised resampling with replacement achieve 100.0% accuracy.

Classifiers	Supervised resampling replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	100.0%	97.5%	88.9%	98.5%
SVM Polynomial	100.0%	97.0%	88.9%	96.5%
SVM RBF	100.0%	98.5%	83.3%	98.0%
SVM Sigmoid	53.0%	98.5%	83.3%	98.5%
Random Forest	100.0%	98.0%	88.9%	96.0%
BayesNet	100.0%	92.5%	100.0%	94.5%
KStar	100.0%	98.0%	94.4%	98.5%
Conjunctive Rules	100.0%	98.5%	83.3%	97.0%
Decision Table	97.5%	97.0%	77.8%	97.0%
Logistic Regression	98.5%	95.0%	83.3%	94.5%

Table A.20: All the precisions of actionable warnings classification on proprietary BlackBerry project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. A few classifiers with supervised resampling with replacement achieve 100.0% precision.

Classifiers	Supervised resampling replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	100.0%	25.0%	66.7%	50.0%
SVM Polynomial	100.0%	20.0%	66.7%	0.0%
SVM RBF	100.0%	0.0%	0.0%	0.0%
SVM Sigmoid	53.0%	0.0%	0.0%	0.0%
Random Forest	100.0%	33.3%	66.7%	0.0%
BayesNet	100.0%	0.0%	100.0%	10.0%
KStar	100.0%	0.0%	100.0%	50.0%
Conjunctive Rules	99.1%	0.0%	0.0%	0.0%
Decision Table	95.5%	0.0%	0.0%	0.0%
Logistic Regression	97.2%	0.0%	50.0%	0.0%

Table A.21: All the recalls of actionable warnings classification on proprietary BlackBerry project with resampling techniques. All the classifiers are trained with the reconstructed datasets generated by resampling techniques. All the classifiers with supervised resampling with replacement achieve 100.0% recall.

Classifiers	Supervised resampling replacement	Supervised resampling without replacement	Spread subsample	SMOTE
SVM Linear	100.0%	33.3%	66.7%	33.3%
SVM Polynomial	100.0%	33.3%	66.7%	0.0%
SVM RBF	100.0%	0.0%	0.0%	0.0%
SVM Sigmoid	100.0%	0.0%	0.0%	0.0%
Random Forest	100.0%	33.3%	66.7%	0.0%
BayesNet	100.0%	0.0%	100.0%	33.3%
KStar	100.0%	0.0%	66.7%	33.3%
Conjunctive Rules	100.0%	0.0%	0.0%	0.0%
Decision Table	100.0%	0.0%	0.0%	0.0%
Logistic Regression	100.0%	0.0%	100.0%	0.0%

Table A.22: All the accuracies of actionable warnings classification on Apache HTTPD project with cost-sensitive techniques. SVM RBF Kernel and cost-sensitive prediction achieves highest accuracy at 91.5%.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	83.4%	77.9%	69.3%
SVM Polynomial	79.4%	76.9%	50.0%
SVM RBF	91.5%	83.9%	82.9%
SVM Sigmoid	89.9%	58.3%	89.9%
Random Forest	76.9%	77.4%	75.9%
BayesNet	62.8%	78.4%	55.8%
KStar	84.4%	75.9%	80.9%
Conjunctive Rules	10.1%	71.4%	68.3%
Decision Table	39.7%	72.9%	60.3%
Logistic Regression	76.9%	78.9%	62.3%

Table A.23: All the precisions of actionable warnings classification on Apache HTTPD project with cost-sensitive techniques. The highest precision is achieved by SVM RBF Kernel and cost-sensitive prediction.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	15.8%	22.7%	15.8%
SVM Polynomial	20.0%	19.0%	12.3%
SVM RBF	71.4%	31.3%	26.7%
SVM Sigmoid	0.0%	10.1%	0.0%
Random Forest	15.8%	20.9%	16.7%
BayesNet	13.5%	24.4%	14.6%
KStar	26.1%	18.2%	25.0%
Conjunctive Rules	10.1%	13.7	13.6%
Decision Table	11.5%	20.7%	15.3%
Logistic Regression	19.0%	17.6%	15.2%

Table A.24: All the recalls of actionable warnings classification on Apache HTTPD project with cost-sensitive techniques. The highest recall is achieved by Conjunctive Rules with cost-sensitive prediction.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	15.0%	50.0%	6.0%
SVM Polynomial	35.0%	40.0%	65.0%
SVM RBF	25.0%	50.0%	40.0%
SVM Sigmoid	0.0%	40.0%	0.0%
Random Forest	30.0%	45.0%	35.0%
BayesNet	50.0%	55.0%	70.0%
KStar	30.0%	40.0%	45.0%
Conjunctive Rules	100.0%	35.0%	40.0%
Decision Table	75.0%	60.0%	65.0%
Logistic Regression	40.0%	30.0%	60.0%

Table A.25: All the accuracies of actionable warnings classification on MySQL project with cost-sensitive techniques. The best classifier is SVM which achieves 89.1% accuracy with cost-sensitive prediction.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	85.0%	71.0%	61.1%
SVM Polynomial	77.7%	71.0%	40.9%
SVM RBF	89.1%	79.8%	80.3%
SVM Sigmoid	89.6%	10.4%	89.6%
Random Forest	75.1%	80.8%	77.2%
BayesNet	59.6%	76.2%	62.7%
KStar	83.9%	77.7%	71.0%
Conjunctive Rules	10.4%	47.7%	35.8%
Decision Table	10.4%	69.4%	45.1%
Logistic Regression	83.9%	80.8%	64.2%

Table A.26: All the precisions of actionable warnings classification on MySQL project with cost-sensitive techniques. Most of the precisions for MySQL despite of classifiers are from 10.0% to 20.0%

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	23.5%	9.1%	17.5%
SVM Polynomial	10.3%	7.1%	13.8%
SVM RBF	0.0%	14.8%	5.0%
SVM Sigmoid	0.0%	10.4%	0.0%
Random Forest	18.2%	28.2%	26.0%
BayesNet	17.0%	25.0%	17.5%
KStar	21.1%	15.2%	12.5%
Conjunctive Rules	10.4%	12.8%	10.0%
Decision Table	10.4%	16.9%	14.2%
Logistic Regression	29.6%	24.2%	18.2%

Table A.27: All the recalls of actionable warnings classification on MySQL project with cost-sensitive techniques. Conjunctive Rules and Decision Table both achieve 100.0% recall with cost-sensitive prediction.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	20.0%	20.0%	70.0%
SVM Polynomial	15.0%	15.0%	90.0%
SVM RBF	0.0%	20.0%	5.0%
SVM Sigmoid	0.0%	100.0%	0.0%
Random Forest	40.0%	55.0%	65.0%
BayesNet	75.0%	65.0%	70.0%
KStar	20.0%	25.0%	30.0%
Conjunctive Rules	100.0%	70.0%	65.0%
Decision Table	100.0%	50.0%	85.0%
Logistic Regression	40.0%	40.0%	70.0%

Table A.28: All the accuracies of actionable warnings classification on proprietary BlackBerry project with cost-sensitive techniques. All the accuracies are over 90.0% except from classifier BayesNet despite of cost-sensitive techniques.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	97.5%	97.0%	91.5%
SVM Polynomial	98.0%	96.0%	95.0%
SVM RBF	98.5%	98.0%	98.5%
SVM Sigmoid	98.5%	98.5%	98.5%
Random Forest	97.5%	96.0%	96.0%
BayesNet	88.0%	96.0%	82.5%
KStar	98.0%	98.0%	98.5%
Conjunctive Rules	98.5%	97.5%	98.5%
Decision Table	96.5%	96.5%	96.0%
Logistic Regression	94.0%	96.5%	93.0%

Table A.29: All the precisions of actionable warnings classification on proprietary BlackBerry project with cost-sensitive techniques. The highest precision is achieved by KStar and meta-cost at 50.0%.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	25.0%	28.6%	11.1%
SVM Polynomial	33.3%	14.3%	18.2%
SVM RBF	0.0%	0.0%	0.0%
SVM Sigmoid	0.0%	0.0%	0.0%
Random Forest	33.3%	0.0%	14.3%
BayesNet	4.3%	0.0%	5.6%
KStar	0.0%	0.0%	50.0%
Conjunctive Rules	0.0%	0.0%	0.0%
Decision Table	16.7%	0.0%	14.3%
Logistic Regression	0.0%	16.7%	0.0%

Table A.30: All the recalls of actionable warnings classification on proprietary BlackBerry project with cost-sensitive techniques. The highest recall achieved by a few classifiers is 66.7%.

Classifiers	Cost-sensitive prediction	Cost-sensitive learning	Meta-cost
SVM Linear	33.3%	66.7%	66.7%
SVM Polynomial	33.3%	0.0%	66.7%
SVM RBF	0.0%	0.0%	0.0%
SVM Sigmoid	0.0%	0.0%	0.0%
Random Forest	66.7%	0.0%	33.3%
BayesNet	33.3%	0.0%	66.7%
KStar	0.0%	0.0%	33.3%
Conjunctive Rules	0.0%	0.0%	0.0%
Decision Table	33.3%	0.0%	33.3%
Logistic Regression	0.0%	33.3%	0.0%