Expert Systems With Applications 167 (2021) 114134

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

# Understanding static code warnings: An incremental AI approach

Xueqi Yang[a,*], Zhe Yu[a], Junjie Wang[b], Tim Menzies[a]

[a] Department of Computer Science, North Carolina State University, Raleigh, NC, USA
[b] Institute of Software Chinese Academy of Sciences, Beijing, China

ARTICLE INFO

*Keywords:*
Actionable warning identification
Active learning
Static analysis
Selection process

ABSTRACT

Knowledge-based systems reason over some knowledge base. Hence, an important issue for such systems is how to acquire the knowledge needed for their inference. This paper assesses active learning methods for acquiring knowledge for "static code warnings".

Static code analysis is a widely-used method for detecting bugs and security vulnerabilities in software systems. As software becomes more complex, analysis tools also report lists of increasingly complex warnings that developers need to address on a daily basis. Such static code analysis tools are usually over-cautious; i.e. they often offer many warnings about spurious issues. Previous research work shows that about 35% to 91 % warnings reported as bugs by SA tools are actually unactionable (i.e., warnings that would not be acted on by developers because they are falsely suggested as bugs).

Experienced developers know which errors are important and which can be safely ignored. How can we capture that experience? This paper reports on an incremental AI tool that watches humans reading false alarm reports. Using an incremental support vector machine mechanism, this AI tool can quickly learn to distinguish spurious false alarms from more serious matters that deserve further attention.

In this work, nine open-source projects are employed to evaluate our proposed model on the features extracted by previous researchers and identify the actionable warnings in a priority order given by our algorithm. We observe that our model can identify over 90% of actionable warnings when our methods tell humans to ignore 70 to 80% of the warnings.

## 1. Introduction

Knowledge acquisition problem is a longstanding and challenging bottleneck in artificial intelligence, especially like Semantic Web project (Feigenbaum, 1980). Traditional knowledge engineering methodologies handcraft the knowledge prior to testing that data on some domain (Hoekstra, 2010). Such handcrafted knowledge is expensive to collect. Also, building competent systems can require extensive manually crafting– which leads to a long gap between crafting and testing knowledge.

In this paper, we address these problems with self-adaptive incremental active learning utilizing a human-in-the-loop process. This approach can be leveraged to filter spurious vs serious static warnings generated by static analysis (SA) tools.

Static code analysis is a common operation of detecting bugs and security vulnerabilities in software systems. The wide range of commercial applications of static analysis demonstrates the industrial perception that these tools have a very high economic value. One of the popular SA tools, FindBugs[1] (shown in Fig. 1) has been downloaded over a million times so far. However, large amounts of warnings are falsely suggested by SA tools as bugs to developers, overwhelming the few actionable ones (true bugs). Due to high rates of unactionable warnings, the utility of such static code analysis tools is questionable. Previous research work shows that about 35% to 91 % warnings reported as bugs by SA tools are actually unactionable (Kim & Ernst, 2007b; Heckman & Williams, 2008; Heckman & Williams, 2011).

Experienced developers have the knowledge of filtering out ignorable and unactioable warnings. Our active learning methods incrementally acquire and validate this knowledge. By continuously and incrementally constructing and updating the model, our approach can help SE developers to identify more actionable static warnings with very low inspection costs and provide an efficient way to deal with software mining on the early life cycle.

This paper evaluates the proposed approach with the following four

* Corresponding author.
*E-mail addresses:* xyang37@ncsu.edu (X. Yang), zyu9@ncsu.edu (Z. Yu), wangjunjie@itechs.iscas.ac.cn (J. Wang).
[1] http://findbugs.sourceforge.net/.

https://doi.org/10.1016/j.eswa.2020.114134
Received 30 November 2019; Received in revised form 18 May 2020; Accepted 16 October 2020
Available online 2 November 2020

research questions:

**RQ1.** What is the baseline rate for bad static warnings?

While this is more a systems question rather than a research question, it is a necessary precondition to our work since it documents the problem we are trying to address. For this question, we report results from FindBugs. These results will serve as the baseline for the rest of our work.

**RQ2.** What is the previous state-of-the-art method to tackle the prevalence of actionable warnings in SA tools?

Wang, Wang, and Wang (2018) conduct a systematic evaluation of all the publicly available features (116 features in total) that discuss static code warnings. That work offered a ”golden set of features”; i.e., 23 features that Wang et al. (2018) argued were most useful for extracting serious bug reports generated from FindBugs. Our experiments combining three supervised learning models from the literature with these 23 features.

**RQ3.** Does incremental active learning reduce the cost to identify actionable Static Warnings?

We will show that incremental active learning reduces the cost of identifying actionable warnings dramatically (and obtains performance almost as good as supervised learning).

**RQ4.** How many samples should be retrieved to identify all the actionable Static Warnings?

In this case study, incremental active learning can identify over 90% of actionable warnings by learning from about 20% to 30% of data. Hence, we recommend this system to developers who wish to reduce the time they waste chasing spurious errors.

### 1.1. Organization of this paper

The remainder of this paper is organized as follows. Research background and related work is introduced in Section 2. In Section 3, we describe the detail of our methodology. Our experiment details are introduced in Section 4. In Section 5, we answer proposed research questions. Threats to validity and future work are discussed in Section 6 and we finally draw a conclusion in Section 7.

To facilitate other researchers in this area, all our scripts are data are freely available on-line[2].

### 1.2. Contributions of this Paper

In the literature, active learning methods have been extensively discussed, like finding relevant papers in literature review (Yu, Kraft, & Menzies, 2018; Yu & Menzies, 2019), security vulnerability prediction (Yu, Theisen, Williams, & Menzies, 2019), crowdsourced testing (Wang, Wang, Cui, & Wang, 2016), place-aware application development (Murukannaiah & Singh, 2015), classification of software behavior (Bowring, Rehg, & Harrold, 2004), and multi-objective optimization (Krall, Menzies, & Davies, 2015). The unique contribution of this work lies in the novel application of these methods to resolving problems with static code warnings. To the best of our knowledge, no prior work has tried to tame spurious static code warnings by treating these as an incremental knowledge acquisition problem.

## 2. Related work

### 2.1. Reasoning about source code

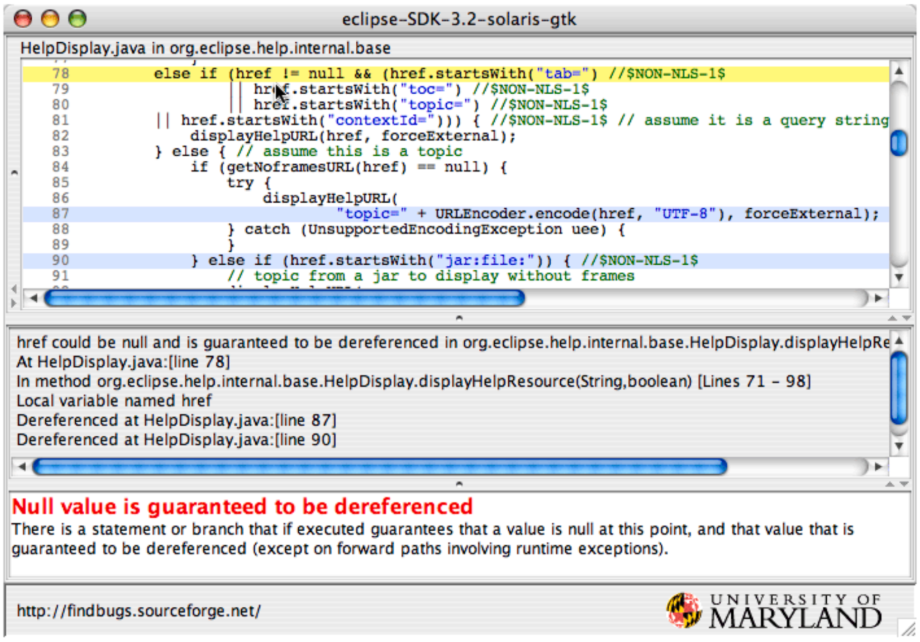
The software development community has produced numerous static code analysis tools such as FindBugs, PMD[3], or Checkstyle[4] that are able to generate various warnings to help developers identifying potential code problems. Such static code analysis tools such as FindBugs leverage *static analysis* (SA) techniques to inspect source code for the occurrence of bug patterns (i.e., the code idiom that is often an error) without actually executing nor considering an exact input. These bugs detected by FindBugs are grouped into a pattern list, (i.e, performance, style, correctness and so forth) and each bug is reported by FindBugs with priority from 1 to 20 to measure the severity, which is finally grouped into four scales either scariest, scary, troubling, and of concern (Ayewah, Pugh, Hovemeyer, Morgenthaler, & Penix, 2008).

Some SA tools learn to identify new bugs using historical data from past problems. This is not ideal since it means that whenever there are chances to tasks, languages, platforms, and perhaps even developers then the old warnings might go out of date and new ones have to be learned. Static warning identification is increasingly relying on complex software systems (Wijayasekara, Manic, Wright, & McQueen, 2012). Identifying static warnings in every stage of the software life cycle is essential, especially for projects in early development stage (Murtaza, Khreich, Hamou-Lhadj, & Bener, 2016).

[2] Download our scripts and data fromhttps://github.com/XueqiYang/incrementally-active-learning_SWID.

[3] https://pmd.github.io/

[4] http://checkstyle.sourceforge.net/


eclipse-SDK-3.2-solaris-gtk

HelpDisplay.java in org.eclipse.help.internal.base

```
78              else if (href != null && (href.startsWith("tab=") //$NON-NLS-1$
79                      || href.startsWith("toc=") //$NON-NLS-1$
80                      || href.startsWith("topic=") //$NON-NLS-1$
81              || href.startsWith("contextId="))) { //$NON-NLS-1$ // assume it is a query string
82                  displayHelpURL(href, forceExternal);
83              } else { // assume this is a topic
84                  if (getNoframesURL(href) == null) {
85                      try {
86                          displayHelpURL(
87                                  "topic=" + URLEncoder.encode(href, "UTF-8"), forceExternal);
88                      } catch (UnsupportedEncodingException uee) {
89                      }
90                  } else if (href.startsWith("jar:file:")) { //$NON-NLS-1$
91                      // topic from a jar to display without frames
```

```
href could be null and is guaranteed to be dereferenced in org.eclipse.help.internal.base.HelpDisplay.displayHelpRe
At HelpDisplay.java:[line 78]
In method org.eclipse.help.internal.base.HelpDisplay.displayHelpResource(String,boolean) [Lines 71 - 98]
Local variable named href
Dereferenced at HelpDisplay.java:[line 87]
Dereferenced at HelpDisplay.java:[line 90]
```

**Null value is guaranteed to be dereferenced**
There is a statement or branch that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions).

http://findbugs.sourceforge.net/

**Fig. 1.** Example of a static code analysis warning, generated via the FindBugs tool.

Arnold et al. (2009) suggests that every project, early in its own lifecycle, should build its own static warning system. Such advice is hard to follow since it means a tedious, time-consuming and expensive retraining process at the start of each new project. To say that in another way, Arnold et al.'s advice suffers from the knowledge acquisition bottleneck problem.

## 2.2. *Static warning identification*

Static warning identification aims at identifying common coding problems early in the development process via SA tools and distinguishing actionable warnings from unactionable ones (Heckman & Williams, 2011; Hovemeyer & Pugh, 2004; Yan et al., 2017).

Previous studies have shown that false positives in static alerts have been one of the most important barriers for developers to use static analysis tools (Thung, Lo, Jiang, Rahman, & Devanbu, 2015; Avgustinov et al., 2015; Johnson, Song, Murphy-Hill, & Bowdidge, 2013). To address this issue, many techniques have been introduced to identify actionable warnings or alerts. Various models have been mentioned in their study, including graph theory (Boogerd & Moonen, 2008; Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012), machine learning (Wang, Liu, & Tan, 2016; Shivaji, Whitehead, Akella, & Kim, 2009) etc. However, most of the studies are plagued by a common issue, choosing the appropriate warning characteristics from abundant feature artifacts proposed by SA studies so far.

Ranking schemes are one way to improve static analysis tool (Kremenek, Ashcraft, Yang, & Engler, 2004). Allier, Anquetil, Hora, and Ducasse (2012) proposed a framework to compare 6 warning ranking algorithms and identified the best one to rank warnings. Similarly, Shen, Fang, and Zhao (2011) employed a ranking technique to rank the true error reports on top so as to reduce false positive warnings. Some other works also prioritize warnings by selecting different categories of impact factors (Liang et al., 2010) or by analyzing software history (Kim & Ernst, 2007a).

Recent work has shown that this problem can be solved by combining machine learning techniques to identify whether a detected warning is actionable or not, e.g., finding alerts with similar code patterns and building prediction models to classify new alerts (Hanam, Tan, Holmes, & Lam, 2014). Heckman and Williams did a systematic literature review revealing that most of these works focus on exploring a reasonable characteristic set, like Alert characteristics (AC) and Code characteristics (CC), to distinguish actionable and unactionable warnings more accurately (Heckman & Williams, 2011; Hanam et al., 2014; Heckman & Williams, 2009). One of the most integrated study explores 15 machine learning algorithms and 51 warning characteristics derived from static analysis tools and achieves good performance with high recall (83–99 %) (Heckman & Williams, 2009). However, in practice, information on bug warning patterns is limited to be obtained, especially for some trivial checkers in SA tools. Also, these tools suffer from conflation issues where similar warnings are given different names in different studies.

Wang et al. (2018) recently conducted a systematic literature review to collect all publicly available features (116 in total) for SA analysis and implemented a tool based on Java for feature extraction. All the values of these collected features are extracted from warning reports generated by FindBugs based on 60 revisions of 12 projects. Six machine learning classifiers were employed to automatically identify actionable static warning. 23 common features were identified as the best and most useful feature combination for Static Warning Identification, since the best performance is always obtained when using these 23 golden features, better than using total feature set or other subset strategies. To the best of our knowledge, this is the most exhaustive research about SA characteristics yet published.

## 2.3. *Active learning*

Labeled data is required by supervised machine learning techniques. Without such data, these algorithms cannot learn predictors. Obtaining good labeled data can sometimes be time consuming and expensive. In the case of this paper, we are concerned with learning how to label static code warnings (spurious or serious). For another example, training a good document classifier might require hundreds of thousands of samples. Usually, these examples do not come with labels, and therefore expert knowledge (e.g., recognizing a handwritten digit) is required to determine the "right" label.

Active learning (Settles, 2009) is a machine learning algorithm that enables the learners to actively choose which examples to label from amongst the currently unlabeled instances. This approach trains on a little bit of labeled data, and then asks again for some more labels for the unlabelled examples that are most "interesting" (e.g. whose labels are most uncertain). This process greatly reduces the amount of labeled data required to train a model while still achieving good predictive performance.

Active learning has been applied successfully in several SE research areas, such as finding relevant papers in literature review (Yu et al., 2018; Yu & Menzies, 2019), security vulnerability prediction (Yu et al., 2019), crowd sourced testing (Wang et al., 2016), place-aware application development (Murukannaiah & Singh, 2015), classification of software behavior (Bowring et al., 2004), and multi-objective optimization (Krall et al., 2015). Overall, there are three different categories of active learning:

- *Membership query synthesis.* In this scenario, a learner is able to generate synthetic data for labeling, which might not be applicable to all cases.
- *Stream-based selective sampling.* Each sample is considered separately in the case of label querying or rejection. There are no assumptions on data distribution, and therefore it is adaptive to change.
- *Pool-based sampling.* Samples are chosen from a pool of unlabeled data for the purpose of labeling. The learner is usually initially trained on a fully labeled fraction of data to generate a preliminary model, which is subsequently used to identify which sample would be most beneficial to be used next in the training set during the next generation of active learning loop. Pool-based sampling scenario is the most widely adopted scheme in literature, which is also applied in our work.

Previous work has shown the successful adoption of active learning in several research areas. Wang et al. (2016) applied active learning to identify the test reports that reveal "true fault" from a large amount of test reports in crowdsourced testing of GUI applications. Within that framework, they proposed a classification technique that labels a fraction of most informative samples with user knowledge, and trained classifiers based on the local neighborhood.

Yu et al. (2018), Yu and Menzies (2019) and Yu, Carver, Rothermel, and Menzies (2019a) proposed a framework called FASTREAD to assist researchers to find the relevant papers to read. FASTREAD works by (1) leveraging external domain knowledge (e.g., keyword search) to guide the initial selection of papers; (2) using an estimator of the number of remaining paper to decide when to stop; (3) applying error correction algorithm to correct human mislabeling. This framework has also been shown effective in solving other software engineering problems (Yu et al., 2018), such as inspecting software security vulnerabilities (Yu et al., 2019), finding self-admitted technical debt (Fahid, Yu, & Menzies, 2019), and test case prioritization (Yu et al., 2019b). In this work, we adopt a similar framework in static warning analysis.

To the best of our knowledge, this work is the first study to utilize incremental active learning to reduce unnecessary inspection of static warnings based on the most effective feature attributes. While Wang et al. is the closest work to this paper, we differ very much from their work.

- In that study, their raw data was screen-snaps of erroneous conditions within a GUI. Also, they spend much effort tuning a feedback mechanism specialized for their images.
- In our work, our raw data is all textual (the text of a static code warning). We found that a different method, based on active learning, worked best for such textual data.

## 3. Methodology

### 3.1. Overview

This work applies an incremental active learning framework to identify static warnings. This is derived from active learning, which has been proved outperformed in solving the total recall problem in several areas, e.g., electronic discovery, evidence-based medicine, primary study selection, test case prioritization, and so forth. As illustrated in Fig. 2, we aim to achieve higher recall with lower effort in inspecting warnings generated by SA tools.

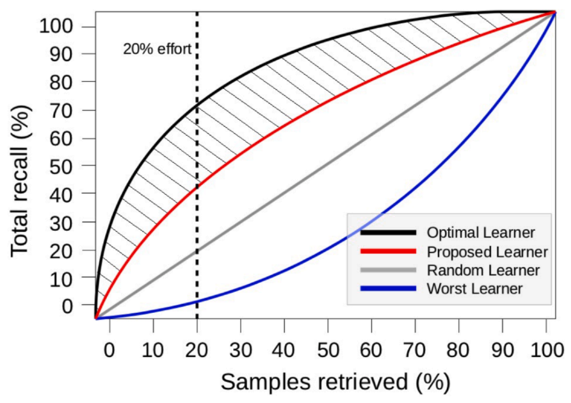


**Fig. 2.** Learning Curve of Different Learners.

### 3.2. Evaluation metrics

Table 1 represents all the variables involved in our study. We evaluated the active learning results in terms of *total recall* and *cost*, which are demonstrated as follows:

*Total recall* addresses the ratio between samples labeled but not revealing actionable warning and total real actionable warning samples. The optimal value of *total recall* is 1, which represents all of the target samples (or actionable warning in our case) have been retrieved and labeled as actionable.

*Cost* considers the set of warning that has currently been retrieved or labeled out of the set of warning reported by the static warning analysis tools. The value of cost varies between the ratio of actionable warnings in the dataset and 1. The lower bound means active learning algorithm prioritizes all targeted samples without uselessly labeling any unactionable warnings. This is a theoretical optimal value (which, in practice, may be unreachable). The upper bound means active learning algorithm successfully retrieves all the real warning samples, but at the cost of labeling them all (which is meaningless because randomly labeling samples will achieve the same goal).

Fig. 2 is an Alberg diagram showing the learning curve of different learners. In this figure, the x-axis and y-axis respectively represent the percentage of warnings retrieved or labeled by learners (i.e. cost) and the percentage of actionable warnings retrieved out of total actionable ones (i.e. total recall). An optimal learner will achieve higher total recall than others when a specific cost threshold is given, e.g., at the cost of 20 % effort as illustrated in Fig. 2. The best performance in Fig. 2 is obtained by optimal learner, followed by proposed learner, random learner and worst learner. This learning curve is a performance measurement at different cost thresholds settings.

**Table 1**
Description of Variables in Incremental Active Learning.

| Variable | Description |
|---|---|
| $E$ | Set of warning that reported by static analysis tools |
| $T$ | Set of actionable warning or target samples |
| $L$ | Set of warning that has been currently retrieved or labeled |
| $L_T$ | Set of warning has been currently labeled and reveals actionable warning |
| Total Recall | $L_T/T$ |
| cost | $L/E$ |

*AUC* (Area under the ROC Curve) measures the area under the Receiver Operator Characteristic (ROC) curve (Witten, Frank, Hall, & Pal, 2016; Heckman & Williams, 2011) and reflects the percentage of actionable warnings against the percentage of unactionable ones so as to overall report the discrimination of a classifier (Wang et al., 2018). This is a widely adopted measurement in Software Engineering, especially for imbalanced data (Liang et al., 2010).

### 3.3. *Active learning model operators*

Several operators are apply to address the challenge of the total recall problem, as listed in Table 2. Specific details about each operator are illustrated as follows:

**Classifier**. We employ three machine learning classifiers as an embedded active learning model, linear SVM with weighting scheme, Random Forest and Decision Tree with default parameters as these classifiers are widely explored in software engineering area and also reported in Wang et al.'s paper. All of the classifiers are modules from Sckit-learn (Pedregosa et al., 2011), a Python package for machine learning.

**Presumptive non-relevant examples**, proposed by Cormack and Grossman (2015), is a technique to alleviate the sample bias of negative samples in unbalanced dataset. To be specific, before each training process, the model samples randomly from the unlabeled pool and assumes that the sampled instance is labeled as negative in training, due to the prevalence of negative samples.

**Aggressive undersampling** (Wallace, Schmid, Lau, & Trikalinos, 2009) is a sampling method to cope with an unbalanced dataset by throwing away majority negative training points close to the decision plane of SVM and aggressively accessing minority positive points until the ratio of these two categories is balanced. It's an effective approach to kill unbalanced bias in datasets. This technique is suggested by Wallace, Trikalinos, Lau, Brodley, and Schmid (2010) after the initial stage of incremental active learning and when the established model becomes stable.

The **querying strategy** is the approach utilized to determine which data instance in an unlabelled pool to query for labelling next. We adopt two of the most commonly used strategies, *uncertainty sampling* (Settles, 2009) and *certainty sampling* (Miwa, Thomas, O'Mara-Eves, & Ananiadou, 2014).

Uncertainty sampling (Settles, 2009) is the simplest and most commonly used query strategy in active learning, where unlabeled samples closest to the decision plane of SVM or predicted to be the least likely positive by a classifier are sampled for query. Wallace et al. (2010) recommended uncertainty sampling method in biomedical literature review and it reduces the cost of manually screening literature efficiently.

Certainty sampling (Miwa et al., 2014) is a kind of greedy algorithm to maximize the utility of incremental learning model by prioritizing the samples which are the most likely to be actionable warnings. Contrary to uncertainty sampling, certainty sampling method gives priority to the instances which are far away from the decision plane of SVM or have the highest probability score predicted by the classifier. It speeds up the process of retrieving and plays the major role of stopping earlier.

**Table 2**
Operators of Active Learning.

| Operator | Description |
|---|---|
| Machine Learning Classifier | Widely-used classification technique. |
| Presumptive non-relevant examples | Alleviate the sampling bias of non-relevant examples. |
| Aggressive Undersampling | Data-balancing technique. |
| Query strategy | Uncertainly sampling and certainty sampling in active learning. |

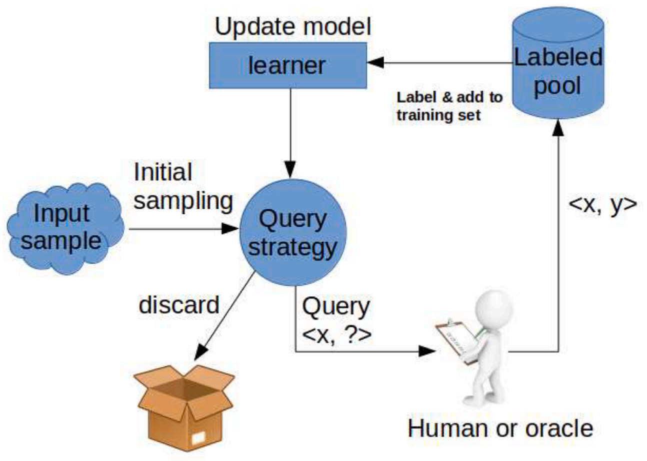


**Fig. 3.** Procedure of Incremental Active Learning.

### 3.4. *Active Learning Procedures*

Fig. 3 presents the procedures of incremental active learning, and a detailed description of each step is demonstrated as follows:

1. Initial Sampling.
   We propose two initial sampling strategies to cope with the scenario that historical information is available or not.
   For software projects in early life cycle without sufficient historical revisions in the version control system, random sampling without replacement is used in the initial stage when the labeled warning pool is NULL.
   For software projects with previous version information, we utilize version N-1 to get a pre-trained model and initialize sampling on version N. This practice can reduce the cost of manually excluding unactionable warnings since the prevalence of false positive in SA datasets.
2. Human or oracle labeling.
   After a warning message is selected by initial sampling or query strategy, manual inspection is required to identify whether the retrieved warning is actionable or not. In our simulation, the ground truth serves as a human oracle and it returns a label once a warning presupposed as unlabeled is queried by active learning model.
   In static analysis, inspecting and tagging the warning being queried is considered as a main overhead of this process. As demonstrated in Table 4, this overhead is denoted as **Cost** and is what software developers strive to reduce.
3. Model Training and updating.
   After a new coming-in warning is labeled by human oracle, this data sample is added to training data. The model is retrained and updated recursively.
4. Query Strategy.
   Uncertainty sampling is leveraged when the actionable samples retrieved and labeled by our model is under a specific threshold. This query strategy mainly applies when targeted data samples are rare in training set and building a stable model faster is required (Yu et al., 2019). Finally, after labeled actionable warning exceed the given threshold, certainty sampling is employed to aggressively searching for true positive and greedily reduce the cost of warning inspection.

## 4. Experiment

### 4.1. *Static warning dataset*

The nine datasets explored in this work are collected from previous research. Wang et al. (2018) performed a systematic literature review to

**Table 4**
Categories of Selected Features. (8 categories are shown in the left column, and 95 features explored in Wang et al. are shown in the right column with 23 golden features in bold.)

| Category | Features |
|---|---|
| Warning combination | size content for warning type; size context in method, file, package; **warning context in method, file,** package; **warning context for warning type**; fix, non-fix change removal rate; **defect likelihood for warning pattern**; variance of likelihood; defect likelihood for warning type; **discretization of defect likelihood**; **average lifetime for warning type**; |
| Code characteristics | method, file, package size; comment length; **comment-code ratio**; **method, file depth**; method callers, callees; **methods in file**, package; classes in file, **package**; indentation; complexity; |
| Warning characteristics | **warning pattern, type, priority,** rank; warnings in method, file, **package**; |
| File history | latest file, package modification; file, package staleness; **file age**; **file creation**; deletion revision; **developers**; |
| Code analysis | call name, class, **parameter signature**, return type; new type, new concrete type;operator; field access class, field; catch; field name, type, visibility, is static/final; **method visibility**, return type, is static/ final/ abstract/ protected; class visibility, is abstract/ interfact/ array class; |
| Code history | added, changed, deleted, growth, total, percentage of LOC in file in the past 3 months; **added**, changed, deleted, growth, total, percentage of LOC in file in the last 25 revisions; **added**, changed, deleted, growth, total, percentage of LOC in package in the past 3 months; added, changed, deleted, growth, total, percentage of LOC in package in the last 25 revisions; |
| Warning history | warning modifications; warning open revision; **warning lifetime by revision**, by time; |
| File characteristics | file type; file name; package name; |

gather all publicly available features (116 in total) for SA analysis. For this research, all the values of this collected feature set were extracted from warnings reported by FindBugs on the 60 successive revisions of 12 projects. Using the Static Warning (SA) tool, we applied FindBugs to 60 revisions from 12 projects' revision history. By collected performance statistics from three supervised learning classifiers on 12 datasets, a golden feature set (23 features) is found via a greedy backward elimination algorithm. We utilize the best feature combination as the warning characteristics in our research.

On closer inspection of these datasets, we found three projects with obvious data inconsistency issues (such as data features mismatch with data labels). Hence, our study only explored the remaining nine projects.

Table 3 lists the summary of projects surveyed in our paper. For each project, there are 5 versions collected from starting revision time after a specific revision interval. We train the model on version 4 and test on version 5.

Previous research (Wang et al., 2018) collected 116 static warning features with a systematic literature review. These features fall into eight categories, and 95 features are left after eliminating unavailable ones as shown in Table 4. We employ the 23 golden features as the independent variables proposed by Wang et al., which are highlighted in bold in Table 4. In our study, the dependent variable is actionable or unactionable. These labels were generated via a method proposed by previous researches (Heckman & Williams, 2008; Hanam et al., 2014; Liang et al., 2010). That is, for a specific warning, if it is closed in later revision after a revision interval when the project was collected, it will finally be labeled as actionable. For warning still existing after later revision interval, it will be labeled as unactionable. Otherwise, for some minority warnings which are deleted after later interval, they will be removed and ignored in our study.

Table 5 shows the number of warnings and distribution of each warning type (as reported by FindBugs) in nine software projects. Note that our data is highly imbalanced with the ratio of targeted samples from 3 to 34 percent.

### 4.2. *Machine learning algorithms*

We choose three machine learning algorithms, i.e., Support Vector Machine (SVM), Random Forest (RF), Decision Tree (DT). These classifiers are selected for their common use in the software engineering literature. All these three algorithms are studied in Wang et al.'s paper (Wang et al., 2018) and the best performance is obtained by Random Forest, followed by Decision Tree. Regarding to SVM, it obtains the worst perform reported in six algorithms by Wang et al. (2018), but due to its wide combination with active learning and promising performance in many research areas like image retrieval (Pasolli, Melgani, Tuia, Pacifici, & Emery, 2013) and text classification (Tong & Koller, 2001), especially imbalanced problems (Ertekin, Huang, & Giles, 2007), we also include this algorithm in our work. We now give a brief description of these algorithms and their application in this work.

All our learners come from the Python toolkit Scikit-Learn (Pedregosa et al., 2011). For the most part, we use the default parameters from that toolkit. Exception for support vector machines, we followed the advice of a previous publication (Krishna, Yu, Agrawal, Dominguez, & Wolf, 2016) which suggested using a linear, and a not radial, kernel.

**Support Vector Machine.** Support Vector Machine (SVM) (Cortes & Vapnik, 1995) is a supervised learning model for binary classification and regression analysis. The optimization objective of SVM is to maximize the margin, which is defined as the distance between the separating hyperplane (i.e., the decision boundary) and the training samples (i.e., support vectors) that are closest to the hyperplane. Support vector machine is a powerful linear model, it also can tackle nonlinear problems through the kernel trick, which consists of multiple hyperparameters that can be tuned to make good predictions.

**Random Forest.** Random forests (Liaw et al., 2002) can be viewed as an ensemble of decision trees. The idea behind ensemble learning is to combine weak learners to build a more robust model or a strong learner, which has a better generalization error and is less susceptible to overfitting. Such forests can be utilized for both classification and regression problems, and also employed to measure the relative importance of each feature on the prediction (by counting how often attributes are used in each tree of the forest).

**Decision Tree.** Decision tree learners are known for their ability to decompose complex decision processes into small and simple subsets

**Table 3**
Summary of Projects Surveyed.

| Project | Period | Revision- Interval | Domain |
|---|---|---|---|
| Lucence-solr | 01/2013–01/2014 | 3 month | Search engine |
| Tomcat | 01/2013–01/2014 | 3 month | Server |
| Derby | 01/2013–01/2014 | 3 month | Database |
| Phoenix | 01/2013–01/2014 | 3 month | Driver |
| Cassandra | 01/2013–01/2014 | 3 month | Big data manage |
| Jmeter | 01/2012–01/2014 | 6 month | Performance manage |
| Ant | 01/2012–01/2014 | 6 month | Build manage |
| Commonslang | 01/2012–01/2014 | 6 month | Java utility |
| Maven | 01/2012–01/2014 | 6 month | Project manage |

**Table 5**
Number of Samples on Version 5.

| Project | Open/Unactionable | Close/Actionable | Delete |
|---|---|---|---|
| ant | 1061 | 54 | 0 |
| commons | 744 | 42 | 0 |
| tomcat | 1115 | 326 | 0 |
| jmeter | 468 | 145 | 7 |
| cass | 2245 | 356 | 64 |
| phoenix | 2046 | 343 | 13 |
| mvn | 790 | 28 | 44 |
| lucence | 2257 | 1168 | 440 |
| derby | 2386 | 121 | 0 |

(Safavian & Landgrebe, 1991). In this process an associated multistage decision tree is hierarchically developed. There are several tree-based approaches widely used in software engineering areas like ID3, C4.5, CART and so forth. Decision tree is computationally cheap to use, and is easy for developers or managers to interpret.

Algorithm 1: Pseudo Code for Supervised Learning.

```
Input   : $V_{n-1}$, previous version for training
          $V_n$, current version for prediction
          $C$, common set of features shared by five releases
Output  : Total Recall, total recall for version n
          cost, samples retrieved by percent

// Keep reviewing until stopping rule satisfied
while $|L_R| < 0.95|R|$ do
    // Start training or not
    if $|L_R| \geq 1$ then
        $CL \leftarrow Train(L)$;
        // Query next
        $x \leftarrow Query(CL, \neg L, L_R)$;
    else
        // Random Sampling
        $x \leftarrow Random(\neg L)$;
    end
    // Simulate review
    $L_R, L \leftarrow Include(x, R, L_R, L)$;
    $\neg L \leftarrow E \setminus L$;
end
return $L_R$;
Function Train($V_{n-1}$)
    // Classifier: Linear-SVM,decision tree,
       random forest
    clf$\leftarrow$ $Classifier$;
    $training_x, training_y \leftarrow V_{n-1}$
    clf$\leftarrow$ $clf.fit(training_x, training_y)$
    return clf;
end
Function PredictProb($V_n$, clf)
    // predict Probability
    $pos_{at} \leftarrow list(clf.classes).index("yes")$
    $testset_x, testset_y \leftarrow V_n$
    $prob \leftarrow clf.PredictProb(testset_x)[:, pos_{at}]$
    return prob, $testset_y$;
end
Function Retrieve(prob, $testset_y$)
    // retrieve by descending-sorted probability
    $sum = 0$
    $order \leftarrow np.argsort(prob)[:: -1][:]$
    $pos_{all} \leftarrow number - of - positive - samples$
    $num_{all} \leftarrow length - of - testset_y$
    while $i \in order$ do
        // Sort label by descending order
        $label_{real} \leftarrow testset_{y[i]}$
        $sorted_{label}.append(label_{real})$
        // Retrieve
        while $label \in sorted_{label}$ do
            if $label == "yes"$ then
                sum + = 1
                $pos_{get} \leftarrow sum$
            else
                continue
            end
        end
        $total_{recall}.append(pos_{get}/pos_{all})$
        cost.append(len($sorted_{label}$) / $num_{all}$)
    end
    return $total_{recall}$, cost;
end
```

## 5. Experiments

In this section, we answer the four research questions formulated in Section 1.

### 5.1. Research method

Static warning tools like FindBugs, Jlint and PMD are widely used in static warning analysis. Previous research has shown that FindBugs is more reliable than other SA tools regarding to its effective discrimination between true and false positives (Wang et al., 2018; Rahman, Khatri, Barr, & Devanbu, 2014). FindBugs is also known as a cost-efficient SA tool for detecting warnings by the combination of line-level, method-level and class-level granularity, thus reports much fewer warnings with obviously more lines (Rahman et al., 2014; Panichella, Arnaoudova, Di Penta, & Antoniol, 2015). Due to all the merits mentioned above, FindBugs has gained widespread popularity among individual users and technology-intensive companies, like Google[5].

In terms of a baseline result, we used the default priority ranking reported by FindBugs. Since FindBugs generates warnings and classifies them into seven categories of patterns (Shen et al., 2011), in which warnings with the same priority in random order have the same severity to be fixed. And the higher priority denotes that the warning report is more likely to be actionable suggested by FindBugs. This randomly ranking strategy provides a reasonable probabilistic bounded time for software developers to find bugs and implements the scene without any information to prioritize warning reports (Heckman & Williams, 2011; Kremenek et al., 2004).

### 5.2. Research results

As is shown in Fig. 4, the dark blue dashed line denotes the learning curve of random selection generated from Findbugs reports. The curve grows diagonally, indicating that an end-user without any historical warning information or auxiliary tool has to inspect 2507 warnings to identify only 121 actionable ones in Derby dataset.

### 5.3. Research method

Wang et al. (2018) implements a Java tool to extract the value of 116 total features collected from exhaustive systematic literature review and employs the machine learning utility Weka[6] to build classification models. An optimal SA feature set with 23 features is identified as the golden features by obtaining the best AUC values evaluated with 6 machine learning classifiers. We reproduce the experiments with three most outperforming supervised learning models in the previous research study, e.g., weighted linear SVM, random forest and decision tree with default parameters in Python3.7. The detailed process to replicate the baseline is demonstrated in Algorithm 1.

The specific process is as follows: For each project, a supervised model (either weighted SVM, Random Forest or Decision Tree) is built by training on Version 4. After the training process, we test on Version 5 for the same project and get a list of probability for each bug reported by FindBugs to be actionable. Sort this list of probability from most likely to be real actionable to least likely and retrieve these warnings in a descending order to report the *total recall*, *cost* and *AUC* as evaluation metrics.

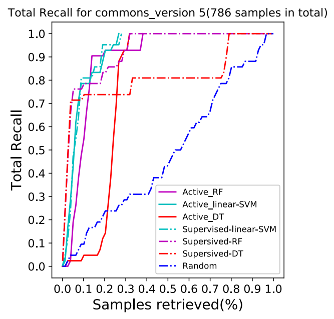
### 5.4. Research results

As shown in Table 6, the median and IQR of AUC scores of ten runs on nine projects are reported in our paper. *Median* and *IQR* are commonly used robust measures of a set of observations. *IQR* (the interquartile range) is a measure of statistical dispersion. It evaluates the variability of distribution by dividing a data set into quartiles and

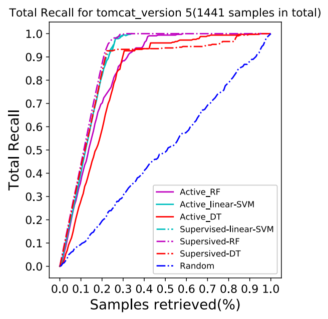
[5] In 2009, Google held a global fixit for UMD's FindBugs tool and aimed at gathering feedback for the 4,000 highest confidence reported by FindBugs. It has been downloaded for more than a million times so far.

[6] https://www.cs.waikato.ac.nz/ ml/weka/

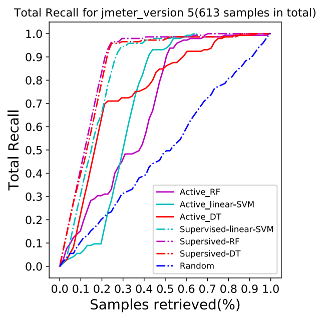
**RQ1.** What is the baseline rate for bad static warnings?



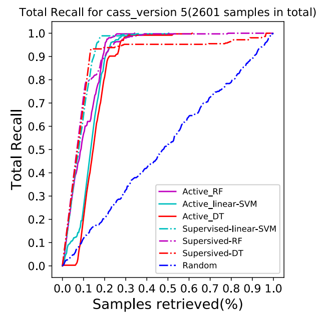
(a) commons



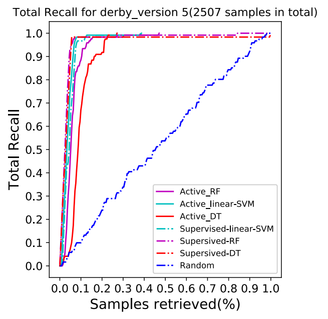
(b) tomcat



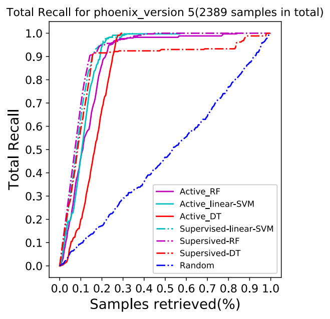
(c) jmeter



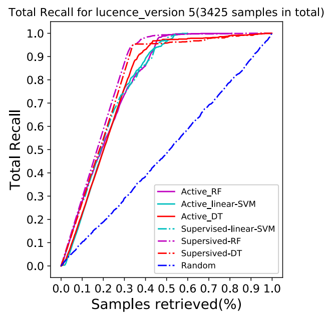
(d) cass



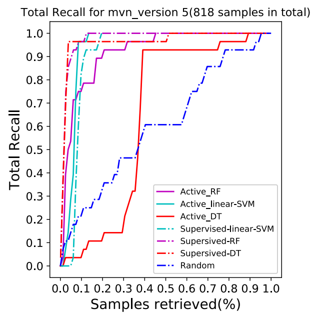
(e) derby



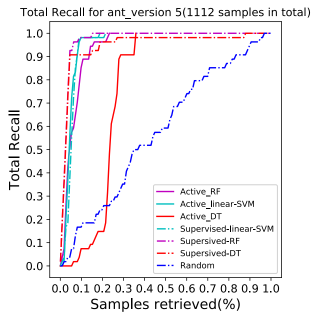
(f) phoenix



(g) lucence



(h) mvn



(i) ant

**Fig. 4.** Test Results of incremental active learning, supervised learning and randomly selection.

**RQ2.** What is the previous state-of-the-art method to tackle the prevalence of actionable warnings in SA tools?

reflecting the difference between 75th and 25th percentiles.

For three supervised learning methods explored, Linear weighted Support Vector Machine and Random Forest both outperform Decision Tree. For incremental active learning algorithms, the best combination is *Active Learning + Support Vector Machine*, followed by *Active Learning + Random Forest* and *Active Learning + Decision Tree*.

It's observed that incremental active learning can obtain high AUC, no worse than supervised learning on most of datasets. The pink shadow highlights the median results of active learning methods which are better or no less 0.05 than the median AUC of the state-of-the-art methods.

The column "Prior Work" shows results reported in Wang et al.'s prior research (Wang et al., 2018). Note that our AUC scores for supervised models replicated with Python3.7 are higher than that prior work implemented by Weka. This difference is explained by two factors.

**Table 6**
AUC % on 9 projects for 10 runs. Our results are better than prior results (shown in blue) since they used default parameters in Weka while we adjusted (e.g.) the SVM kernel (as well as a more recent implementation of these tools).

| Project | Active+SVM | | Supervised_SVM | | Active+RF | | Supervised_RF | | Active+DT | | Supervised_DT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Median** | IQR | **Median (IQR)** | Median of Prior work | **Median** | IQR | **Median (IQR)** | Median of Prior work | **Median** | IQR | **Median (IQR)** | Median of Prior work |
| Derby | 98 | 1 | 97(2) | 50 | 96 | 7 | 97(4) | 43 | 93 | 2 | 94(4) | 44 |
| Mvn | 94 | 3 | 96(7) | 50 | 93 | 2 | 97(3) | 45 | 67 | 3 | 91(2) | 45 |
| Lucence | 95 | 1 | 97(3) | 50 | 85 | 9 | 99(2) | 98 | 94 | 2 | 93(4) | 98 |
| Phoenix | 97 | 2 | 97(3) | 62 | 90 | 7 | 97(3) | 71 | 90 | 2 | 91(7) | 70 |
| Cass | 96 | 5 | 99(3) | 67 | 96 | 4 | 98(5) | 70 | 90 | 1 | 94(4) | 69 |
| Jmeter | 94 | 1 | 95(2) | 50 | 90 | 4 | 97(2) | 86 | 86 | 2 | 91(12) | 82 |
| Tomcat | 98 | 1 | 97(3) | 50 | 92 | 5 | 96(2) | 80 | 94 | 2 | 92(6) | 64 |
| Ant | 95 | 2 | 98(2) | 50 | 94 | 1 | 98(3) | 44 | 84 | 3 | 94(7) | 44 |
| Commons | 91 | 3 | 98(3) | 50 | 93 | 1 | 92(2) | 57 | 80 | 8 | 85(14) | 56 |

- We found that better results could be obtained by adjusting some of the learner parameters; e.g. we use a linear (not radial) kernel for our SVM.
- The implementation tools employed by our study and previous work are different. Prior work used a Java implementation of these tools (in Weka) while our replication utilizes a more recent Python toolkit (Scikit-Learn) that is being used and updated by a larger and more developed community.

**RQ3.** Does incremental active learning reduce the cost to identify actionable Static Warnings?

The purpose of this research question is to compare incremental active learning with random selection and traditional supervised learning models.

### 5.5. Research method

Considering a real-world scenario when a software project in different stages of the life cycle, RQ3 is answered in two parts: We first contrast incremental active learning, denoted as solid lines in Fig. 4 with random ranking (default ranking reported from FindBugs, denoted as dark blue dashed line in Fig. 4). Then, we compare active learning results with supervised learning (denoted as purple, lighted blue and red dashed lines in Fig. 4).

### 5.6. Research results

Results of supervised learning methods are denoted as light blue, purple and red dashed lines. As revealed in Fig. 4, Random Forest outperforms the other classifiers, followed by Linear SVM and Decision Tree.

Fig. 4 provides an overall view of the experiment results to address Research Question 3. These nine subplots are the results of a ten-time repeated experiment on fourth and fifth versions of nine projects and we only report the median values here. The latest version 5 is selected to construct incremental active learning, while for the supervised learning model, we choose the two latest versions, learning patterns from version fourth for model construction and testing on version fifth for evaluation to make the experimental results comparable.

Fig. 5 summarizes the ratio of real actionable warnings in version 5 of each project and the corresponding median of cost when applying incremental active learning to identify all these actionable warnings.

As illustrated in Fig. 4, incremental active learning outperforms random selection, which simulates real-time cost bound when an end-user recurs to warning reports prioritized by FindBugs. While, the learning curve of incremental active learning without historical version is almost as good as supervised learning in most of nine projects based on version history. Also, the test results on nine datasets suggest that *Linear SVM + incremental active learning* is the best combination of all active learning schemes, and *Random Forest* is the winner in supervised learning methods.

Overall, the above observations suggest that applying an incremental active learning model in static warning identification can help to retrieve actionable warnings in higher priority and reduce the effort to eliminate false alarms for software projects without adequate version history.

How many samples to be retrieved is a critical problem when implementing an active learning model in the scenario of static warning identification. Stopping too early or too late will incur the issue of missing important actionable warnings or wasting unnecessary running time and CPU resources.

In the following part, we introduce the research method and analysis of the experimental results to answer Research Question 4.

### 5.7. Research method

Fig. 5 employs the box-plot to describe the costs required or percentage of samples retrieved by three classifiers, Linear weighted SVM, Random Forest and Decision Tree combined with our incremental active learning algorithm. Horizontal coordinate of the box charts represents the thresholds of recall, a mechanism to stop retrieving new potential actionable warnings when the proportion of related samples found reached the specific given thresholds. The vertical axis means the corresponding effort required to obtain the given recall, measured by the proportion of warnings retrieved.

### 5.8. Research results

Based on the results shown in Fig. 5, it can be observed that the growth of effort required is in a gentle and slow fashion when the threshold of relevant warnings visited increasing from 70 % to 90 %. However, for reaching 100 % threshold, the effort needed is almost or over twice compared with the cost of threshold equal to 90 %. A very

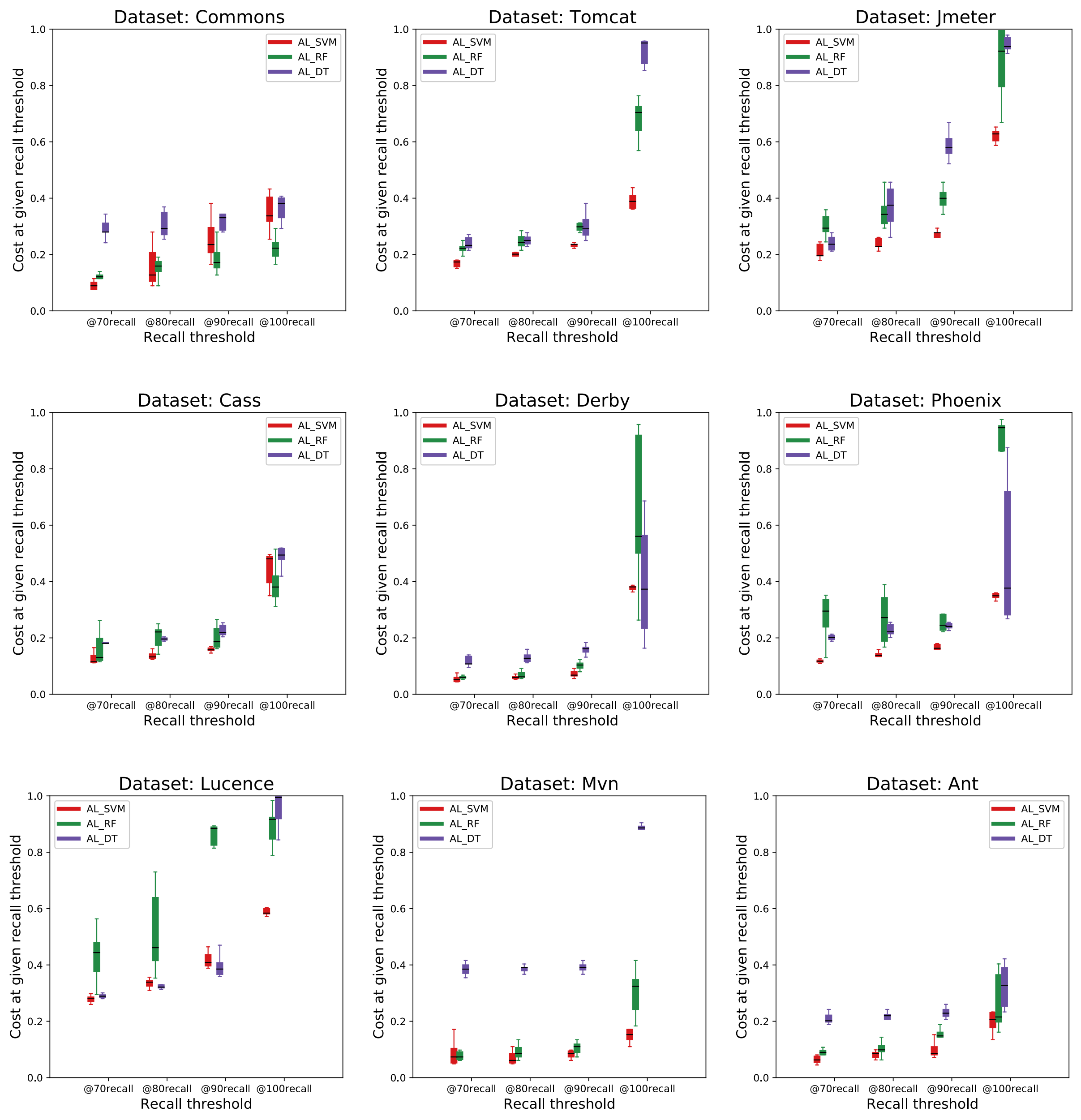


**Fig. 5.** Cost Results at different thresholds for Incremental Active Learning.

**RQ4.** How many samples should be retrieved to identify all the actionable Static Warnings?

intuitive suggestion can be obtained from Fig. 5 is learning from 20 % or 30 % warnings for each of these nine projects, in which case the active learning models can identify over 90 % of actionable warnings.

However, there is an exception. Results of lucence reveal that our model has to inspect more than 40 % of data to identify 90 % actionable warnings. Revisiting Table 5, it indicts that most of our projects have data imbalanced issues (ratio of target class is less than 20 % for derby, mvn, phoenix, cass, commons and ant, and for jmeter and tomcat it's slightly over 20 %) while ratio of lucence (about 35 %) is relatively higher. Our study attempts to provide a solid guideline but there is no general conclusion about the specific percent of data that should be fed into the learner. It highly depends on the degree of data imbalance and the trade-off between missing target samples and reducing costs. Since the cost can only be reduced at the expense of a lower threshold, which means missing some real actionable warnings.

In summary, our model has been proven to be an efficient methodology to deal with information retrieval problem for SA identification of extremely unbalanced data sets, moreover, it is also a good option for engineers and researchers to apply active learning model in general problems because it has a lower building cost, a wider application range, and a higher efficiency compared with state-of-the-art supervised learning methods and random selection.

## 6. Discussion

### 6.1. Threats to validity

As to any empirical study, biases can affect the final results. Therefore, conclusions drawn from this work must be considered with threats to validity in mind. In this section, we discuss the validity of our work.

**Learner bias.** This work applies three classifiers, weighted linear SVM, Random Forest and Decision Tree, which are the best setting according to previous research work (Wang et al., 2018). However, this doesn't necessarily guarantee the best performance in other domains or other static warning datasets. According to the No Free Lunch Theorems (Wolpert et al., 1997), applying our method framework to other areas would be needed before we can assert that our methods are also better in those domains.

**Sampling bias.** One of the most important threats to validity is sampling bias since several sampling methods, random sampling, uncertainty sampling and certainty sampling, are used in combination. However, there are also many sampling methods in the active learning area we can utilize. And different sampling strategies and combinations may result in better performance. This is a potential research direction.

**Ratio bias.** In this paper, we propose an ideal scale value for our learner to retrieve on nine static warning datasets to effectively solve the prevalence of false positive in warnings reported by SA tools. obvious improvement is observed for this unbalanced problem. But it doesn't necessarily apply to balanced datasets.

**Measurement bias.** To evaluate the validity of the incremental active learning method proposed in this paper, we employ two measurement metrics: total recall and cost. Several prior research work has demonstrated the necessity and effectiveness of these measurements (Yu et al., 2019; Yu et al., 2018; Yu & Menzies, 2019). Nevertheless, many studies are still based on some classic and traditional metrics, eg. confusion matrix or also known as error matrix (Landgrebe & Duin, 2008). There exist many popular terminology and derivations from confusion matrix, false positive, F1 score, G measure, and so on. We cannot explore and include all the options in one article. Also, even for this same research methodology, conclusions drawn from different evaluation matrices may differ. However, in this research scenario, it is more efficient to report recall and cost for an effort-aware model.

### 6.2. Future work

**Estimation.** In real-world problems, labeled data may be scarce or expensive to be obtained, while data without labels may be abundant. In this case, the query process of our incremental learning model cannot safely stop to obtain a given targeted threshold without knowing the actual number of actionable warnings in the data set beforehand. Therefore, estimation is required to guarantee the algorithm stopping detection at an appropriate stage: stopping too late will cause unnecessary cost to explore unactionable warnings and increase false alarms; while stopping too early may incur missing potential and important true warnings.

**Ensemble of classifiers.** Ensemble learning is a methodology of making decision based on inputs of multiple experts or classifiers (Zhang & Ma, 2012). It's a feasible and important scheme to reduce the variance of classifiers and improve the reliability and robustness of the decision system. The famous No Free Lunch Theorems proposed by Wolpert et al. (1997) gives us an instinct guidance to recur to ensemble learners. This will be promising to make the best of incremental active learning by precisely making predictions and pinpoint real actionable warnings with a generalized decision system.

## 7. Conclusion

Previous research work shows that about 35% to 91% warnings reported as bugs by static analysis tools are actually unactionable (i.e., warnings that would not be acted on by developers because they are falsely suggested as bugs). Therefore, to make such systems usable for programmers, some mechanism is required to reduce those false alarms.

Arnold et al. (2009) warn that knowledge about what is an ignorable static code warning may not transfer from project to project. Here, they advise that methods for managing static code warnings should be tuned to different software projects. While we agree with that advice, it does create a knowledge acquisition bottleneck problem since acquiring that knowledge can be a time-consuming and tedious task.

This explored methods for acquiring knowledge of what static code warnings can be ignored. Using a human-in-the-loop active learner, we conducted an empirical study with 9 software projects and 3 machine learning classifiers to verify how the performance of current SA tools could be improved by an efficient incremental active learning method. We found about 90 % of actionable static warnings can be identified when only inspecting about 20 % to 30 % warning reports without using historical version information. Our study attempts to bridge the research gap between supervised learning and effort-aware active learning models by an in-depth analysis of reducing the cost of static warning identification problems.

Our method significantly decreases the cost of inspecting falsely reported warnings generated by static code analysis tools for software engineers (especially in the early stage of software project's life cycle) and provides a meaningful guideline to improve the performance of current SA tools. Acceptance and adoption of future static analysis tools can be enhanced by combining with SA feature extraction and self-adaptive incremental active learning.

## CRediT authorship contribution statement

**Xueqi Yang:** Methodology, Software, Writing - original draft, Visualization. **Zhe Yu:** Conceptualization, Resources. **Junjie Wang:** Data curation. **Tim Menzies:** Supervision, Writing - review & editing, Funding acquisition.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgement

This work was partially funded by NSF grant #1908762.

## References

Allier, S., Anquetil, N., Hora, A., & Ducasse, S. (2012). A framework to compare alert ranking algorithms. In *2012 19th Working Conference on Reverse Engineering* (pp. 277–285). IEEE.

Arnold, J., Abbott, T., Daher, W., Price, G., Elhage, N., Thomas, G., & Kaseorg, A. (2009). Security impact ratings considered harmful. arXiv preprint arXiv:0904.4058.

Avgustinov, P., Baars, A. I., Henriksen, A. S., Lavender, G., Menzel, G., de Moor, O., Schäfer, M., & Tibble, J. (2015). Tracking static analysis violations over time to capture developer characteristics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (pp. 437–447). IEEE Press.

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. (2008). Using static analysis to find bugs. *IEEE Software, 25*, 22–29.

Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M. (2012). Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 419–429). IEEE.

Boogerd, C., & Moonen, L. (2008). Assessing the value of coding standards: An empirical study. In *2008 IEEE International Conference on Software Maintenance* (pp. 277–286). IEEE.

Bowring, J. F., Rehg, J. M., & Harrold, M. J. (2004). Active learning for automatic classification of software behavior. In *ACM SIGSOFT Software Engineering Notes* (pp. 195–205). ACM.

Cormack, G. V. & Grossman, M. R. (2015). Autonomy and reliability of continuous active learning for technology-assisted review. arXiv preprint arXiv:1504.06868.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning, 20*, 273–297.

Ertekin, S., Huang, J., & Giles, C. L. (2007). Active learning for class imbalance problem. In *SIGIR* (pp. 823–824).

Fahid, F. M., Yu, Z., & Menzies, T. (2019). Better technical debt detection via surveying. arXiv preprint arXiv:1905.08297.

Feigenbaum, E. A. (1980). Knowledge engineering: the applied side of artificial intelligence. Technical Report. Stanford Univ CA Dept of Computer Science.

Hanam, Q., Tan, L., Holmes, R., & Lam, P. (2014). Finding patterns in static analysis alerts: improving actionable alert ranking, in. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 152–161). ACM.

Heckman, S. & Williams, L. (2008). On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, ACM. pp. 41–50.

Heckman, S., & Williams, L. (2009). A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing Verification and Validation* (pp. 161–170). IEEE.

Heckman, S., & Williams, L. (2011). A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology, 53*, 363–387.

Hoekstra, R. (2010). The knowledge reengineering bottleneck. *Semantic Web, 1*, 111–115.

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *Acm sigplan Notices, 39*, 92–106.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 672–681). IEEE Press.

Kim, S. & Ernst, M.D. (2007a). Prioritizing warning categories by analyzing software history, in: Proceedings of the Fourth International Workshop on Mining Software Repositories, IEEE Computer Society. p. 27.

Kim, S., & Ernst, M. D. (2007b). Which warnings should i fix first?, in. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 45–54). ACM.

Krall, J., Menzies, T., & Davies, M. (2015). Gale: Geometric active learning for search-based software engineering. *IEEE Transactions on Software Engineering, 41*, 1001–1018.

Kremenek, T., Ashcraft, K., Yang, J., & Engler, D. (2004). Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes* (pp. 83–93). ACM.

Krishna, R., Yu, Z., Agrawal, A., Dominguez, M., & Wolf, D. (2016). The'bigse'project: Lessons learned from validating industrial text mining. In *2016 IEEE/ACM 2nd International Workshop on Big Data Software Engineering (BIGDSE)* (pp. 65–71). IEEE.

Landgrebe, T. C., & Duin, R. P. (2008). Efficient multiclass roc approximation by decomposition via confusion matrix perturbation analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 30*, 810–822.

Liang, G., Wu, L., Wu, Q., Wang, Q., Xie, T., & Mei, H. (2010). Automatic construction of an effective training set for prioritizing static analysis warnings, in. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (pp. 93–102). ACM.

Liaw, A., Wiener, M., et al. (2002). Classification and regression by randomforest. *R News, 2*, 18–22.

Miwa, M., Thomas, J., O'Mara-Eves, A., & Ananiadou, S. (2014). Reducing systematic review workload through certainty-based screening. *Journal of Biomedical Informatics, 51*, 242–253.

Murtaza, S. S., Khreich, W., Hamou-Lhadj, A., & Bener, A. B. (2016). Mining trends and patterns of software vulnerabilities. *Journal of Systems and Software, 117*, 218–228.

Murukannaiah, P. K., & Singh, M. P. (2015). Platys: An active learning framework for place-aware application development and its evaluation. *ACM Transactions on Software Engineering and Methodology (TOSEM), 24*, 19.

Panichella, S., Arnaoudova, V., Di Penta, M., & Antoniol, G. (2015). Would static analysis tools help developers with code reviews? In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE. pp. 161–170.

Pasolli, E., Melgani, F., Tuia, D., Pacifici, F., & Emery, W. J. (2013). Svm active learning approach for image classification using spatial information. *IEEE Transactions on Geoscience and Remote Sensing, 52*, 2217–2233.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research, 12*, 2825–2830.

Rahman, F., Khatri, S., Barr, E. T., & Devanbu, P. (2014). Comparing static bug finders and statistical prediction, in. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 424–434). ACM.

Safavian, S. R., & Landgrebe, D. (1991). A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics, 21*, 660–674.

Settles, B. (2009). *Active learning literature survey. Technical Report.* University of Wisconsin-Madison Department of Computer Sciences.

Shen, H., Fang, J., & Zhao, J. (2011). Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (pp. 299–308). IEEE.

Shivaji, S., Whitehead, E. J., Jr, Akella, R., & Kim, S. (2009). Reducing features to improve bug prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 600–604). IEEE.

Thung, F., Lo, D., Jiang, L., Rahman, F., Devanbu, P. T., et al. (2015). To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. *Automated Software Engineering, 22*, 561–602.

Tong, S., & Koller, D. (2001). Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research, 2*, 45–66.

Wallace, B. C., Schmid, C. H., Lau, J., & Trikalinos, T. A. (2009). Meta-analyst: software for meta-analysis of binary, continuous and diagnostic data. *BMC Medical Research Methodology, 9*, 80.

Wallace, B. C., Trikalinos, T. A., Lau, J., Brodley, C., & Schmid, C. H. (2010). Semi-automated screening of biomedical citations for systematic reviews. *BMC Bioinformatics, 11*, 55.

Wang, J., Wang, S., Cui, Q., & Wang, Q. (2016). Local-based active classification of test report to assist crowdsourced testing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 190–201). IEEE.

Wang, J., Wang, S., & Wang, Q. (2018). Is there a golden feature set for static warning identification?: an experimental evaluation, in. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (p. 17). ACM.

Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 297–308). IEEE.

Wijayasekara, D., Manic, M., Wright, J. L., & McQueen, M. (2012). Mining bug databases for unidentified software vulnerabilities. In *2012 5th International Conference on Human System Interactions* (pp. 89–96). IEEE.

Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2016). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wolpert, D. H., Macready, W. G., et al. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation, 1*, 67–82.

Yan, M., Zhang, X., Xu, L., Hu, H., Sun, S., & Xia, X. (2017). Revisiting the correlation between alerts and software defects: A case study on myfaces, camel, and cxf. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (pp. 103–108). IEEE.

Yu, Z., Carver, J. C., Rothermel, G., & Menzies, T. (2019a). Searching for better test case prioritization schemes: a case study of ai-assisted systematic literature review. arXiv preprint arXiv:1909.07249.

Yu, Z., Fahid, F., Menzies, T., Rothermel, G., Patrick, K., & Cherian, S. (2019b). Terminator: Better automated ui test case prioritization. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 883–894). New York, NY, USA: ACM. https://doi.org/10.1145/3338906.3340448.

Yu, Z., Kraft, N. A., & Menzies, T. (2018). Finding better active learners for faster literature reviews. *Empirical Software Engineering, 23*, 3161–3186.

Yu, Z., & Menzies, T. (2018). Total recall, language processing, and software engineering, in. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering* (pp. 10–13). ACM.

Yu, Z., & Menzies, T. (2019). Fast2: An intelligent assistant for finding relevant papers. *Expert Systems with Applications, 120*, 57–71.

Yu, Z., Theisen, C., Williams, L., & Menzies, T. (2019). Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering, 1–1*. https://doi.org/10.1109/TSE.2019.2949275

Zhang, C., & Ma, Y. (2012). *Ensemble machine learning: methods and applications*. Springer.