# FRUGAL: Unlocking Semi-Supervised Learning for Software Analytics

Huy Tu, Tim Menzies
Com Sci, NCState, USA
hqtu@ncsu.edu, timm@ieee.org

## ABSTRACT

Standard software analytics often involves having a large amount of data with labels in order to commission models with acceptable performance. However, prior work has shown that such requirements can be expensive, taking several weeks to label thousands of commits, and not always available when traversing new research problems and domains. Unsupervised Learning is a promising direction to learn hidden patterns within unlabelled data, which has only been extensively studied in defect prediction. Nevertheless, unsupervised learning can be ineffective by itself and has not been explored in other domains (e.g., static analysis and issue close time).

Motivated by this literature gap and technical limitations, we explore the performance variations seen in several simple optimization schemes. We present FRUGAL, a tuned semi-supervised method that builds on a simple optimization scheme that does not require sophisticated (e.g., deep learners) and expensive (e.g., 100% manually labelled data) methods. Our method optimizes the unsupervised learner's configurations in the grid search manner while validating the picked settings on only 10% of the labelled train data before predicting. FRUGAL outperforms the state-of-the-art actionable static code warning recognizer and issue closed time predictor with less information, reducing the cost of labelling by 90%.

Hence we assert that FRUGAL can save considerable efforts in data labelling especially in validating prior work or researching new problems. Also, proponents of complex and expensive methods should always baseline such methods against simpler and cheaper alternatives. For instance, a semi-supervised learner like FRUGAL can serve as a baseline to the state-of-the-art software analytics tools.

## 1 INTRODUCTION

Software analytics can guide improvements to software quality, maintenance and security. For example, analytics can discover which static code warnings are actionable [79, 88]; whether the new issues can be easily fixed [50, 91]; where software defects are likely to occur [1, 63]; which comments likely to contain technical debts [48, 92]; what the current health conditions of these open-source projects [82]; or how to distinguish security bug reports [71].


*ASE 2021, 15 - 19 November, 2021, Melbourne, Australia*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, models that perform these software analytics tasks typically learn from *labelled data*. Generating such labels can be extremely slow and expensive. For instance, Tu et al. [76] reported that manually reading and labelling 22, 500+ commits required 175 person-hours (approximately nine weeks), including cross-checking among labellers. Due to the labor-intensive nature of the process, researchers often reuse datasets labelled from previous studies. For instance, Lo et al. [87], Yang et al. [89], and Xia et al. [86] certified their methods using data generated by Kamei et al. [36]. While this practice allows researchers to rapidly test new methods, it leaves the possibility for any labelling mistake to propagate to other related works. In fact, in technical debts identification, before reusing prior work's data [48], Yu et al. [92] discovered that more than 98% of the false positives were actually true positives, casting doubt on work that used the original dataset. Hence, it is timely to ask:

*Can we reduce the labelling effort associated with building models for software analytics?*

Unsupervised learning techniques that learns patterns from unlabelled data is a promising direction for software analytics. Such learning has been used for buggy/non-buggy classification [83–85, 89, 90, 96]. The state-of-the-art (SOTA) unsupervised learner is Nam and Kim [60]'s CLA($C$) method. CLA is based on the binary split of the output space at the aggregated median ($C = 50\%$) of all features' median in the data. However, other areas and different datasets may not share the same data characteristics for the default CLA ($C = 50\%$) to perform well. To address this gap, our study adopts and extends CLA from defect prediction to other software analytics like static code warnings and issues close time.

Promising extensions for unsupervised learning involves finding different control settings to configure the system (hyperparameter tuning) and validating on small labelled data regions (e.g., 10%) before applying the best setting to the test data. Recent software engineering (SE) research shows many domains' SOTA can be improved with hyperparameter tuning [1, 2, 12, 21, 23, 59]. DODGE is one prominent optimizer which shows the output space of the models on low-dimensional data can be easily surveyed through dodging away from (1) prior options or options that resulted (2) in statistically similar performance. Simply, the central function of CLA (binary split of the output space via aggregated $C$) is synonymous to SE's SOTA optimizer DODGE with less information required, a reduction of 90% train data's labels. Specifically, our work proposed FRUGAL($C$) = three different modes of CLA (as shown in Figure 1) with $C = \{5\%$ to $95\%$, increments by $5\%\}$ where all the combinations can be easily executed in a grid search manner.

To understand and validate the FRUGAL system, we investigate the following research questions:

**RQ1: How much labelled data ($L$%) that FRUGAL requires?**

Huy Tu, Tim Menzies
Com Sci, NCState, USA
hqtu@ncsu.edu, timm@ieee.org

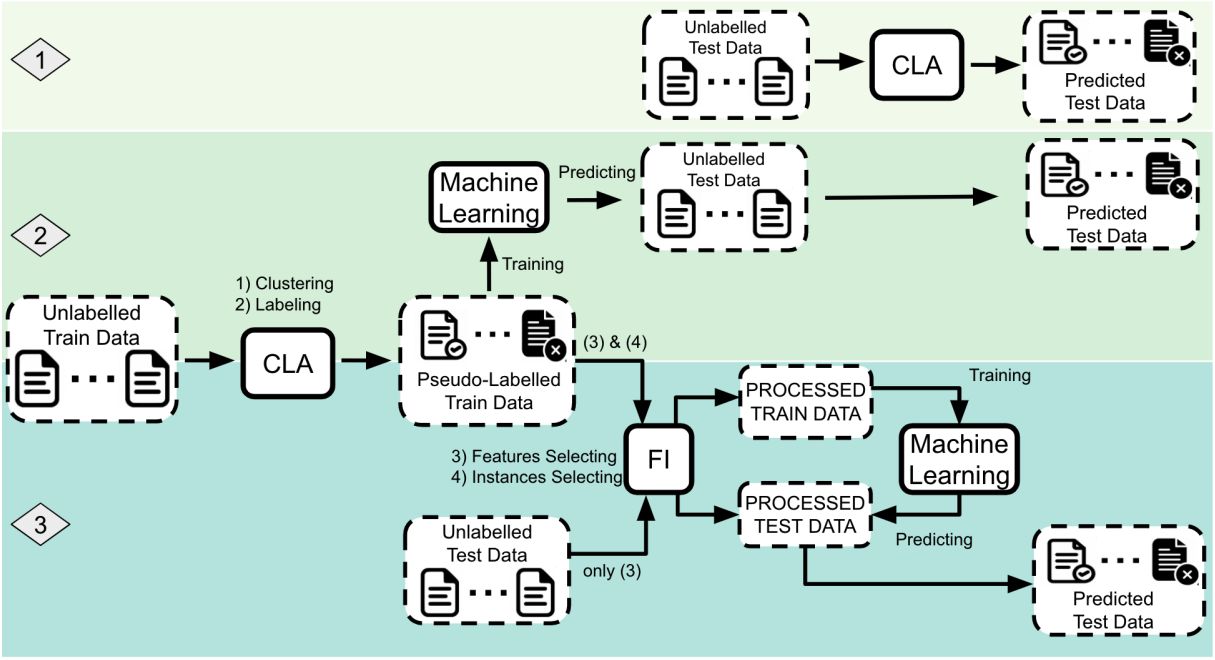


**Figure 1: Three different modes of CLA devised from Nam and Kim [60] for defec t prediction.**

> **Result:**
> From our investigation of various $L$ values, FRUGAL's performance plateaus when $L \geq 10\%$ and FRUGAL's success is not altered by large changes to $L$.

**RQ2: How does FRUGAL perform in actionable static code warnings identification?**

> **Result:**
> When comparing to the SOTA solution in EMSE'20 [88], FRUGAL wins on recall, loses in AUC, and draws in FAR with only 10% of the labelled train data.

**RQ3: How does FRUGAL perform in issue close time prediction?**

> **Result:**
> When comparing to the SOTA solution in EMSE'20 [91] (which was compared to ICSE'10 [26], PROMISE'11 [51], MSR'16 [37], COMAD'19 [50]), FRUGAL outperforms in FAR, recall, and AUC while performing similarly in accuracy with only 10% of the labelled train data.

In summary, our work's contributions to the field of software analytics are as follows:

(1) This work is the first to assess the usage of unsupervised learning to reduce the labelling efforts to commission models building in actionable static code warnings identification and issues close time prediction.
(2) FRUGAL surpasses the SOTA issues close time predictor and performs similarly to the SOTA actionable static code warning identifier with 90% less information.
(3) FRUGAL reduces the labelling efforts to commission new models building by 90%. In another word, FRUGAL is 10 times cheaper than SOTA methods in issue close time and static code warning analysis areas.
(4) The performance of our framework suggests that many more domains in SE could benefit from unsupervised learning beyond defect prediction [22, 60, 62, 83–85, 89, 90, 96, 97].
(5) To better support other researchers our scripts and data are on-line at https://github.com/SE-Efforts/SE_SSL.

The rest of this paper is structured as follows. *Section 2* and *3* discusses the background, related works, and motivation of this work. *Section 4* describes our methodologies. *Section 5* analyzes the results while *Section 6* discusses our short-comings. Finally, *7* concludes the work and states the venues for future work.

## 2 BACKGROUND AND RELATED WORK

### 2.1 Studying Static Code Warnings

*2.1.1 Background.* Static code warning tools detect potential static code defects in source code or executable files at the stage of software product development. This covers a range of potential defects such as common programming errors, code styling, in-line comments common programming anti-patterns, style violations, and questionable coding decisions. The distinguishing feature of these tools is that they make their comments without reference to a particular input. Nor do they use feedback from any execution of the code

being studied. Examples of these tools include PMD[1], Checkstyle[2] and the FindBugs[3] tool.

One issue with static code warnings is that they generate a large number of false positives. Many programmers routinely ignore most of the static code warnings, finding them irrelevant or spurious [55]. Such warnings are considered as "unactionable" since programmers never take action on them. Between 35% and 91% of the warnings generated from static analysis tools are known to be unactionable. This high false alarm rate is one of the most significant barriers for developers to use these tools [4, 35, 74]. Hence it is prudent to learn to recognize what kinds of warnings programmers usually act upon so the tools can be made more useful by first pruning away the unactionable warnings. Various approaches have been tried to reduce these false alarms including graph theory [6, 7], statistical models [14], and ranking schemes [41].

*2.1.2 Data and Algorithms.* The data for this paper comes from a recent study by Wang et al. [79]. They conducted a systematic literature review to collect all public available static code features generated by widely-used static code warning tools (116 in total):

- All the values of these collected features were extracted from warning reports generated by FindBugs based on 60 revisions of 12 projects.
- To ensure the difference between prior and later revision intervals of a project is adequate for the solid conclusions to be drawn, Wang et al. [79] set revision intervals for different projects, e.g., 3 months for *Lucene* and 6 months for *Mvn*. Each project in this study has at least two-years commit history.
- To eliminate ineffective features to the results of those learners, a greedy backward selection algorithm is applied. Then they isolated 23 features as the most useful ones for identifying actionable static code warnings.
- They called these features the "golden set"; i.e. the features most important for recognizing actionable static code warnings.

To the best of our knowledge, this is the most exhaustive research about static warning characteristics yet published. As shown in Table 1, the "golden set" features fall into eight categories. These features are the independent variables used in this study. To assign dependent labels, we applied the methods of Liang et al. [44]. They defined a specific warning as actionable if it is closed after the later revision interval.

By analyzing FindBugs output from two consecutive releases of nine software projects, collecting the features of Table 1, and then applying the Liang et al.'s definitions, we created the data of Table 2. In this table, the "training set" refers to release $i-1$ and the "test set" is release $i$. In this study, we only employ two latest releases. One of many extensive studies exploring the usage of Machine Learning (ML) in this area is Heckaman et al. [29]. They applied 15 ML algorithms to recognize the actionable warnings (programmers can act upon) based on 51 features derived from static analysis tool, they achieved recalls of 83-99 % (average across 15 data sets). The SOTA system that we will compare against is from Yang et al. [88] where they took advice from Ghotra et al. [25] to compare several representative non-neural learners (Table 9 of [25]) in software analytics with various popular neural-network models. They found that all treatments performed similarly to each other but non-neural learners did that with less time than deep learners.

Note that, for any particular data set, the 23 categories of Table 1, can grow to more than 23 features. For example, consider the "return type" feature in the "code analysis" category. This can include numerous return types extracted from a given project, which could be void, int, URL, boolean, string, printStream, file, and date (or a list of any of these periods). Hence, as shown in Table 2, the number of features in our data varied from 39 to 60.

**Table 1: Categories of Wang et al. [79]'s 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;<br>size context in method, file, package;<br>**warning context in method, file,** package;<br>**warning context for warning type**;<br>fix, non-fix change removal rate;<br>**defect likelihood for warning pattern**;<br>variance of likelihood;<br>defect likelihood for warning type;<br>**discretization of defect likelihood**;<br>**average lifetime for warning type**; |
| Code characteristics | method, file, package size;<br>comment length;<br>**comment-code ratio**;<br>**method, file depth**;<br>method callers, callees;<br>**methods in file**, package;<br>classes in file, **package**;<br>indentation;<br>complexity; |
| Warning characteristics | **warning pattern;**<br>**type, priority**;<br>rank, warnings in method, file,<br>**package**; |
| File history | latest file, package modification;<br>file, package staleness;<br>**file age**; **file creation**;<br>deletion revision; **developers**; |
| Code analysis | call name, class, **parameter signature**,<br>return type;<br>new type, new concrete type;<br>operator;<br>field access class, field;<br>catch;<br>field name, type, visibility, is static/final;<br>**method visibility**, return type,<br>is static/ final/ abstract/ protected;<br>class visibility,<br>is abstract / interface / array class; |
| Code history | added, changed, deleted, growth, total, percentage<br>of LOC in file in the past 3 months;<br>**added**, changed, deleted, growth, total, percentage<br>of LOC in file in the last 25 revisions;<br>**added**, changed, deleted, growth, total, percentage<br>of LOC in package in the past 3 months;<br>added, changed, deleted, growth, total, percentage<br>of LOC in package in the last 25 revisions; |
| Warning history | warning modifications;<br>warning open revision;<br>**warning lifetime by revision**, by time; |
| File characteristics | file type;<br>file name;<br>package name; |

[1]https://pmd.github.io/latest/index.html
[2]https://checkstyle.sourceforge.io/
[3]http://findbugs.sourceforge.net

Huy Tu, Tim Menzies
Com Sci, NCState, USA
hqtu@ncsu.edu, timm@ieee.org

**Table 2: Summary of Yang et al. [88]'s data distribution. The gray cells are median values for the corresponding columns.**

| | | training set | | test set | |
|---|---|---|---|---|---|
| Dataset | Features | Instance Counts | Actionable Ratio(%) | Instance Counts | Actionable Ratio(%) |
| commons | 39 | 725 | 7 | 786 | 5 |
| phoenix | 44 | 2235 | 18 | 2389 | 14 |
| mvn (maven) | 47 | 813 | 8 | 818 | 3 |
| jmeter | 49 | 604 | 25 | 613 | 24 |
| cass (cassandra) | 55 | 2584 | 15 | 2601 | 14 |
| ant | 56 | 1229 | 19 | 1115 | 5 |
| lucence | 57 | 3259 | 37 | 3425 | 34 |
| derby | 58 | 2479 | 9 | 2507 | 5 |
| tomcat | 60 | 1435 | 28 | 1441 | 23 |

## 2.2 Predicting Bugzilla Issue Close Time

*2.2.1 Background.* When programmers work on repositories, predicting issue close time has multiple benefits for the developers, managers, and stakeholders since it is helpful for (1) end-users who are directly affected by the product; (2) developers prioritize work; (3) managers allocate resources and improve consistency of release cycles; and (4) stakeholders understand changes in project timelines and budgets:

- Although bugs have an assigned severity, this is not a sufficient predictor for the lifetime of the issue. For example, the author who issued the bug may be significant contributors to the project.
- Alternatively, an issue deemed more *visible* to end-users may be given higher priorities. It is therefore insufficient simply to consider the properties of the issue itself (the *issue metrics*), but also of its environment ( *context metrics*). This is similar to the recent work on how *process metrics* are better defect predicting measurements than *product metrics* [47].

An example of such issue close times estimator can notify involved parties if the recently created issue is an easy fix.

*2.2.2 Data and Algorithms.* The state-of-the-art system for predicting issue close time comes from a recent study by Yedida et al. [91]. They conducted a literature review of 99 research papers that are comprised of (1) from Watson's literature reviews; and (2) top venues listed in Google Scholar metrics for Software Systems, Artificial Intelligence, and Computational Linguistics in the last three years with at least 10 citations per years:

- Traditional or non-neural approaches include (1) Guo et al. [27]'s study on a large closed-source project (Microsoft Windows) to predict whether or not a bug will be fixed; and (2) Marks et al. [51] used ensemble method of decision trees, i.e., random forests, on Eclipse and Mozilla data.
- As to deep learning or neural network approach are DASENet [42] and DeepTriage[50].
- Only a minority of deep learning papers (39.4%) performed any sort of hyper-parameter optimization, i.e., varied few numbers of parameters, such as the number of layers of the deep learner, to edge out the best performance of deep learning. Even fewer papers (18.2%) applied hyper-parameter optimization in a non-trivial manner; i.e., not using a hold-out set to assess the tuning before assessing the separate test set).

To obtain a fair comparison with the prior state-of-the-art, we use the same data as used in the Lee et al. [42], Mani et al. [50], Yedida et al. [91]'s studies. The data was collected from the three projects of Firefox, Chromium, and Eclipse:

**Table 3: An overview of the data used in the Lee et al. [42], Mani et al. [50], and Yedida et al. [91] studies. Note that because of the manner of data collection, i.e., using bin-sequences for each day for each report, there are many more data samples generated from the number of reports mined.**

| Project | Observation Period | # Reports | # Train | # Test |
|---|---|---|---|---|
| Eclipse | Jan 2010–Mar 2016 | 16,575 | 44,545 | 25,459 |
| Chromium | Mar 2014–Aug 2015 | 15,170 | 44,801 | 25,200 |
| Firefox | Apr 2014–May 2016 | 13,619 | 44,800 | 25,201 |

- Preprocessing involves standard text mining to remove special characters or stack traces, tokenization, and pruning the corpus to a fixed length.
- The activities per day were collected into two bins including user activity (e.g., comments), system records (e.g., added/removed labels), and metadata (e.g., the user was the reporter, days from opening, etc).
- Along with the numerical metadata, user and system records are transformed to machine-readable data for the models to execute through word2vec [54, 55].

In the same manner as prior work, the target class is discretized into two bins (so that each bin has roughly the same number of samples). This yields datasets that are near-perfectly balanced (e.g., in the Chromium dataset, we observed a 49%-51% class ratio).

## 2.3 Evaluation

*2.3.1 Measures of Performance.* Since we wish to compare our approach to prior work, we take the methodological step of adopting the same performance scores as that seen in prior work. Let TP, TN, FP, FN are the true positives, true negatives, false positives, and false negatives (respectively), then Yang et al. [88] used AUC, recall, and false-alarm while Yedida et al. [91] using only accuracy for their studies:

- **AUC** (Area Under the ROC Curve) measures the two-dimensional area under the Receiver Operator Characteristic (ROC) curve [30, 81]. It provides an aggregate and overall evaluation of performance across all possible classification thresholds to overall report the discrimination of a classifier [79].
- **Recall** = $TP/(TP + FN)$ represents the ability of one algorithm to identify instances of positive class from the given data set
- **False Alarms (FAR)** = $TN/(TN + FP)$ measures the instances that are falsely classified by an algorithm as positive which are actually negative. This is an important index used to measure the efficiency of a model.
- **Accuracy** = $(TP + TN)/(TP + TN + FP + FN)$ is the percentage of correctly classified samples.

In the effort-aware theme of this paper, we are also interested in the labelling effort to commission new models building which is **Cost** = $\frac{|\{\text{human verified comments}\}|}{|\{\text{comments}\}|}$. Except for FAR and Cost metrics, the rest of the metrics (Accuracy, Recall, and AUC), the higher the better the performance.

*2.3.2 Statistical Analysis.* With the deterministic nature, we employed Cohen'*d* effect size test to determine which results are similar by calculating *medium_step2* across Recall, False Alarm,

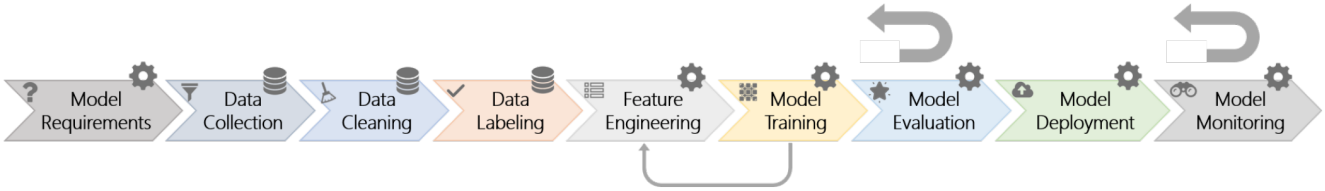


**Figure 2: Nine stages of the machine learning workflow from a case study at Microsoft by Zimmermann et al. [3]. Some stages are data-oriented (e.g., data collection, cleaning, and labelling) and others are model-oriented (e.g., model requirements, features engineering, model training, evaluation, evaluation, deployment and monitoring).**

AUC, Accuracy, and cost. As to what $d$ to use for this analysis, we take the advice of a widely accepted Sawilowsky et al.'s work [68]. That paper asserts that "small" and "medium" effects can be measured using $d = 0.2$ and $d = 0.5$ (respectively). Splitting the difference, we will analyze this data looking for differences larger than $d = (0.5 + 0.2)/2 = 0.35$:

$$Medium_{step2} \text{ or } M = 0.35 \cdot StdDev(\text{All results}) \quad (1)$$

The SOTA actionable code warnings identifier and the SOTA issue close time predictor also validated their results with this test but with $d = 0.35$ and $d = 0.3$ respectively.

## 3 LABELLING

One of the goals of industrial analytics is that new conclusions can be quickly obtained from new data just by applying data mining algorithms. As shown in Figure 2, there are at least nine separate stages that must be completed before that goal be reached [3]. Each of these stages offers unique and separate challenges, each of which deserves extensive attention. Many of these steps have been extensively studied in the literature [16–18, 34, 45, 45, 48, 49, 65, 92, 95]. However, the labelling work of step 4 has been receiving scant attention. In literature, there are several approaches for executing the labelling process:

(1) Manual labelling;
(2) Crowdsourcing;
(3) Reuse of labels;
(4) Automatic labelling;
(5) Active learning (a special kind of semi-supervised learning)

All of these approaches have their drawbacks; e.g. they are error-prone or will not scale. In response to these shortcomings, this study will take two directions:

- First, we will try a *label-free* approach using a plain *unsupervised learning* technique to label the data;
- If the *label-free* approach fails, then we will try a hybrid of a *tuned semi-supervised learning* approach, called FRUGAL, which optimizes the unsupervised learner's configurations in the grid search manner while validating the results on only 10% of the labelled data.

### 3.1 Manual Labelling

In manual labelling, a team of (e.g.) graduate students assigns labels then (a) cross-checks their work via say, a Kappa statistic; then (b) use some skilled third person to resolve any labelling disagreements [46, 75, 76].

Manual labelling can be very slow. Tu et al. recently studies a corpus of 678 Github projects [75, 76]. A random selection of 10 projects from that corpus had 22, 500 commits, which took 175 hours to manually label the commits *buggy, non-buggy* (time includes cross-checking). That is, manual labelling of those 500 projects would have required 90 weeks of work.

### 3.2 Crowdsourcing

Tu et al. [76] offers a cost estimate of what resources would be required to sub-contract that effort to dozens of crowdsourced workers via tools like Mechanical Turk (MT). Applying best practices in crowdsourcing [13], assuming (a) at least USA minimum ages [72]; and (b) our university taking a 50% overhead tax on grants; then crowd sourcing the labelling of the issues from 500 projects would require $320,000 of grant reserve.

### 3.3 Reusing Labels

Since manual labelling is time consuming and crowdsourcing is too expensive, researchers often reuse labels from previous studies. For instance, Lo et al., Yang et al., and Xia et al. certified their methods by using data generated by Kamei et al. [86, 87, 89]. This approach is unsatisfactory for two reasons. One, if exploring a new domain, there may be no relevant old labels to reuse. Two, reusing labels might allow unsatisfactory instances to seep into other works. For example, Yu et al. [92] were exploring self-admitted technical debt and found that their classifiers had an alarming high false-positive rate. But when they manually checked the labels of their data (which they taken from a prior study by Maldonado et al. [48]), they found that over 98% of the reused false-positive labels were incorrect.

### 3.4 Automatic Labelling

If labels cannot be generated manually or reused from other papers, using automatic labelling processes is an attractive alternative. For example, defect prediction papers [10, 31, 36, 38, 56, 61, 66] can label a commit as "bug-fixing" when the commit text contains certain keywords (e.g. "bug", "fix", "wrong", "error", "fail" etc [76]). Vasilescu et al. [77, 78] noted that these keywords are used in a somewhat ad hoc manner (researchers peek at a few results, then tinker with regular expressions that combine these keywords). Tu et al. [76] had found that these simplistic keyword approaches can introduce many errors, perhaps due to the specialization of the project nature or the ad-hoc nature of their creation. [77]). In technical debts identification, Yu et al. proposed a pattern-based method that automatically identified 20-90% of SATDs by finding patterns associated with high precision from the labelled training sets (close to 100%). This approach does need extensively labelled training data to find quality patterns that are associated with technical debt because it relies on precision. Another automatic approach is ML which involves supervised learning models to train on existing labelled datasets to learn the underlying rules of the data. However, this also requires having access to a substantial amount of labelled data

Huy Tu, Tim Menzies
Com Sci, NCState, USA
hqtu@ncsu.edu, timm@ieee.org

(especially for deep learners) which is not always available in new domains (e.g., the success of open-source projects).

### 3.5 Active Learning

A third approach is to (a) only label a representative sample of the data; then (b) build a classifier from that sample; then (c) use that classifier to label the remaining data [80]. To find that representative example, some unsupervised learners like an associations rule learner or a clustering algorithm or an instance selection algorithm is used to find repeated patterns in the data [38]. Then a human oracle is asked to label just one exemplar from each pattern. More sophisticated versions of this scheme include *active learners*, where an AI tool rushes ahead of the human to fetch the most information next examples to be labelled [40, 70]. If humans first label most informative examples, then better models can be built faster. This means, in turn, that humans have to label fewer examples.

The more general term for *active learning* is *semi-supervised learning*. Both terms mean "do what you can with a small sample of the labels" while *active learning* adds a feedback loop that checks new labels, one at a time, with some oracle. Moreover, semi-supervised learning relies on partially labelled data and mostly unlabelled data.

Since 2012, active learning approaches have been received scarce attention in SE [39, 76, 93, 94]. Initially, it seems to be a promising method for addressing the cost of label checking and generating. For self-admitted technical debt identification, only 24% on the median of the training corpus had to be labelled [94]; Also, using active learning, effort estimation for $N$ projects only needed labels on 11% of those projects [39]; Further, while seeking 95% of the vulnerabilities in 28,750 Mozilla Firefox C and C++ source code files, humans only had to inspect 30% of the code [93]. That said, after much work, it must be reported that active learning still produces disappointing results. It is still daunting to "only" label (say) 5% to 10% of the projects in the 1,857,423 projects in RepoReapers [57] or the 9.6 million links explored by Hata et al. [28]. Also, consider the Firefox study mentioned in the last paragraph. The human effort of inspecting 28,750×30% = 8,625 source code files (needed to find identify 95% of the vulnerabilities) it is beyond the resources of most analysts (but it might be justified for mission-critical projects).

It is opportune that in this actionable static code warnings identification and issue close time prediction work, we aim to reduce the reviewing cost of these labelling methods by two methods (1) label-free approach with unsupervised learning, or (2) tuned semi-supervised learning to optimize the unsupervised learner's configurations in the grid search manner while validating the results on a small amount of the labelled data, i.e., 10%.

## 4 METHODOLOGY

### 4.1 General Framework

In order to overcome the previously documented limitation in labelling, where the previous labelling methods are very expensive, the "label-free" approach via unsupervised learning is a promising direction. However, unsupervised learning may not be effective by itself so we propose the FRUGAL framework that is based on the integration of semi-supervised learning and tuning. Nam et al.'s CLA is the SOTA unsupervised learner for defect prediction, which is also confirmed by Xu et al. [83]'s large-scale study. As shown in Figure 1, CLA consists of three modes: CLA, CLA+ML, and CLAFI+ML. This study shall adopt and extend CLA with tuning in the grid search manner of (1) three modes of CLA while varying (2) the $C$% percentile parameter. Simply, FRUGAL finds the best combination of unsupervised learners = {CLA, CLA+ML, CLAFI+ML} and $C$ = {5% to 95% increments by 5%}. The author only proposed CLA and CLAFI+ML but CLA+ML is a natural medium that can be useful during the tuning process. We thoroughly explain in details these unsupervised learners in §4.2, §4.3, and §4.4.

### 4.2 CLA

In the SOTA comparative study of unsupervised models in defect prediction, CLA starts with two steps of (1) **C**lustering the instances and (2) **LA**beling those instances accordingly to the cluster. In the setting with no train data available, we can label or predict all new/test instances, as shown in the first block of Figure 1.

**Clustering**:

(1) Find the median of feature $F_1, F_2, ..., F_n$ ($percentile(F_i, C)$) where $C = 50\%$ across the whole dataset.
(2) For each data instance $X_i$, go through each feature value of the respective data instance to count the time when the feature $F_i > percentile(F_i, C)$ as $K_i$.

**Labelling**: label the instance $X_i$ as the positive class if $K_i > median(K)$, else label it as the negative class.

The intuition of such methods is based on the defect proneness tendency that is often found in defect prediction research, that is *the higher complexity is associated with the proneness of the defects* [60]. Simply, there is a tendency where the problematic instance's feature values are higher than the non-problematic ones. This tendency and CLA's/CLAFI+ML's effectivenesses are confirmed via the recent literature and comparative study of 40 unsupervised models in defect prediction across 27 datasets and three types of features by Xu et al. [83]. They found CLA's/CLAFI+ML's performances are superior to other unsupervised methods while similar to supervised learning approaches. Therefore, this study investigated and found that the hypothesized tendency is also applicable in issues close time prediction and actionable static code warning identification data but not with $C$ at the median ($C = 50\%$). This opens opportunities for hyperparameter tuning.

### 4.3 CLA + ML

If there is an abundant train data in the wild but without labels, CLA can pseudo-label the train data before applying any machine learner in the "supervised" manner (as shown in the second block in Figure 1). For this step, we take Nam and Kim [60]'s advice to incorporate Random Forest [9] (RF, described in §4.5.1), an ensemble of tree learners method, as the machine learner of choice.

### 4.4 CLAFI + ML

CLAFI is an extension of CLA which is a fullstack framework that also include (3) **F**eatures selection and (4) **I**nstances selection. The setting is similar to CLA+ML, as shown in the third block of Figure 1, the pseudo-labelled train data (from CLA) and unlabelled test data will be processed with **FI** and **F** respectively. Finally, machine

learner can train the processed pseudo-labelled train data and then predict on the processed test data.

**Feature Selection**: Calculate the violation score per feature, called metric in the original proposal of Nam et al. [60]. The process is done on both the train and the test dataset.

(1) For each $F_i$, go through all instances of $X_j$, a violation happens when $F_i$ at $X_j$ is higher than the $percentile(K_i, C)$ where $C$ = 50% but $Y_j = 1$ and vice-versa.
(2) Sum all the violations per feature across the whole dataset and sort it in ascending order.
(3) Select the feature with the lowest violation score, if multiple of them have the same score then pick all of them.

**Instance Selection**:

(1) With the selected features, go through each instance $X_i$ and check if the respective $F_j$ values violated the proneness assumption then remove that instance $X_i$.
(2) If the dataset do not have instances with both classes at the end then pick the next minimum violation score to select metrics.
(3) This process is only done on the train dataset.

After selecting features with the minimum violation scores and removing the instances that violated the proneness tendency, a practitioner can train an RF model on the processed train data to identify the target classes from the processed test dataset.

## 4.5 Machine Learning Models

*4.5.1 Random Forest (RF).* is an ensemble learning method that operates by constructing a multitude of decision trees, each time with different subsets of the data rows $R$ and columns $C$[4]. Each decision tree is recursively built to find the features that reduce most of *entropy*, where a higher entropy indicates less ability to draw conclusions from the data being processed [8]. Test data is then passed across all $N$ trees and the conclusions are determined (say) a majority vote across all the trees [9]. Holistically, RF is based on bagging (bootstrap aggregation) which averages the results over many decision trees from sub-samples (reducing variance).

*4.5.2 Support Vector Machine (SVM).* is a discriminative classifier formally defined by a separating hyperplane [73]. Soft-margin linear SVMs are commonly used in text classification given the high dimensionality of the feature space. This was recommended by Yang et al. [88] as the state of the art for our actionable static code warning identification case study. A soft-margin linear SVM looks for the decision hyperplane that maximizes the margin between training data of two classes while minimizing the training error (hinge loss):

$$\min \lambda \|w\|^2 + \left[ \frac{1}{n} \sum_{i=1}^{n} \max\left(0, 1 - y_i(w \cdot x_i - b)\right) \right] \quad (2)$$

where the class of $x$ is predicted as $sgn(w \cdot x - b)$.

Both SVM and RF are popular in the field of ML and implemented in the popular open-source toolkit Scikit-learn by [64].

*4.5.3 Feedforward Neural Networks.* is the first and simplest technology devised from artificial neural network [69]. The information moves in the forward direction only, starting from the input nodes through the hidden nodes and to the output nodes. At each node of these networks, the inputs are multiplied with weights that are learned, and then an activation function is applied. The weights are learned by the backpropagation algorithm [67]. This uses just a few layers while the "deep" learners use many layers. Also, the older methods use a threshold function at each node, while feedforward networks typically use the Rectified Linear Unit function [58] of $f(x) = \max(0, x)$. This is the base learner for our second case study's state of the art where Yedida et al. [91] proposed a framework combining different preprocessors and different configurations of the simple feedforward neural network.

[4]Specifically, using $\log_2 C$ of the columns, selected at random.

# 5 RESULTS

In order to make sure our proposed method's effectiveness is not affected by the bias between deterministic and non-deterministic models or the bias of uncertainty, we shuffle both the train/test set in random order and incorporate stratified sampling with five bins (ensuring that the class distribution of the whole data is replicated within each bin). The process is repeated for the train data but also includes an extra 10% validating partition for each 90% tuning partition. During the simulation, the tune partition will not review labels for our unsupervised learning and semi-supervised learning candidates. FRUGAL does have access to the corresponding 10% labelled validation partition while deciding on the best configurations. For each 20% of the test data, the process learns a model on five stratified samples of the train data.

**RQ1: How much labelled data ($L$%) that FRUGAL requires?**

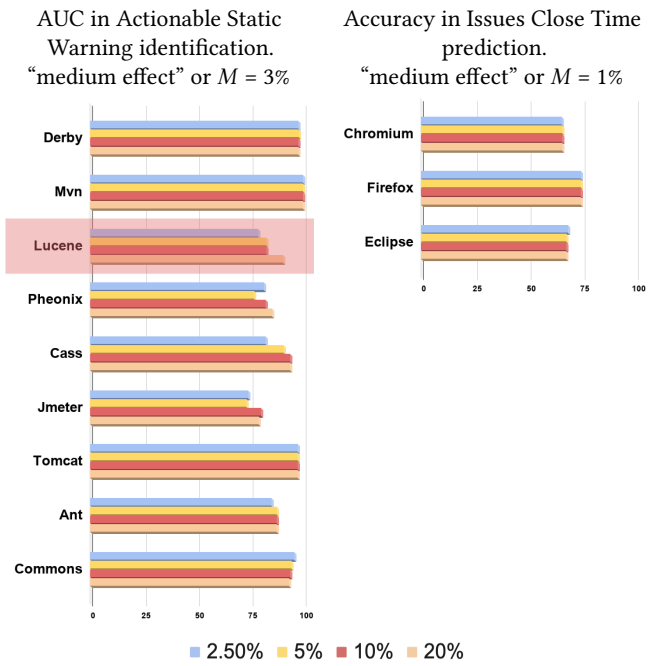


**Figure 3: RQ1 results on two case studies with FRUGAL($L \in \{2.5\%, 5\%, 10\%, 20\%\}$). Changing $L$ does not change the results for issues close time prediction. However, for actionable static warning identification, FRUGAL($L$ = 10% and $L$ = 20%) performed the best across 8 datasets except in Lucene (the highlighted one) where FRUGAL(L=20%) outperformed FRUGAL(L=10%).**

Huy Tu, Tim Menzies
Com Sci, NCState, USA
hqtu@ncsu.edu, timm@ieee.org

**Table 4: Comparison between CLA[60], SVM[88], and FRUGAL in terms of FAR, Recall, and AUC for identifying actionable static warning. Except for FAR, the higher the results the better the performance of the treatment. Medians and IQRs (delta between 75th and 25th percentile, lower the better) are calculated for easy comparisons. Here, the highlighted cells show best performing treatments.**

| Metrics | Treatment | Derby | Mvn | Lucene | Phoenix | Cass | Jmeter | Tomcat | Ant | Commons | Median | IQR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FAR ($M$ = 6%) | CLA | 43.3 | 44.9 | 30.8 | 39.1 | 42.1 | 39.4 | 43 | 45.8 | 43 | 43 | 4.2 |
| | CLAFI+RF | 0.4 | 18.1 | 0.8 | 29.4 | 0.5 | 4.1 | 0.4 | 22.5 | 13.6 | 4.1 | 18.5 |
| | FRUGAL | 0.4 | 3.2 | 0.8 | 1.5 | 0.5 | 3.3 | 0.4 | 1.7 | 3.5 | 1.5 | 2.5 |
| | SVM [88] | 1.3 | 1.2 | 6.9 | 3.5 | 1.4 | 2.1 | 3.2 | 0.5 | 5.8 | 2.1 | 2.7 |
| Recall ($M$ = 5%) | CLA | 45.8 | 100 | 64.5 | 66.7 | 98.6 | 75.9 | 63.1 | 80 | 100 | 75.9 | 32.8 |
| | CLAFI+RF | 57.2 | 93.3 | 67.1 | 62.1 | 83.7 | 73.1 | 64.4 | 77.8 | 77.5 | 73.1 | 12.9 |
| | FRUGAL | 98.9 | 95.5 | 100 | 98.1 | 97 | 96.4 | 93.3 | 97.8 | 100 | 97.8 | 2.4 |
| | SVM [88] | 97.8 | 97 | 87.1 | 96.1 | 90.3 | 93.3 | 98.2 | 95 | 99.5 | 96.1 | 4.8 |
| AUC ($M$ = 6%) | CLA | 54.9 | 88.9 | 66.8 | 64.7 | 83.8 | 72 | 69.7 | 68.5 | 81.1 | 69.7 | 13.7 |
| | CLAFI+RF | 66.8 | 89 | 77.3 | 69.4 | 78.4 | 78 | 75.2 | 75.9 | 84.3 | 77.3 | 4.2 |
| | FRUGAL | 97.3 | 99.7 | 82.5 | 82.2 | 93.7 | 79.8 | 97 | 87.4 | 93.5 | 93.5 | 12 |
| | SVM [88] | 99.5 | 99.6 | 97.3 | 98.8 | 99.7 | 98.8 | 99.6 | 99.7 | 99 | 99.5 | 0.8 |

Our hypothesis is "there are few key data regions where extra data would lead to indistinguishable results". We test the different amounts of the train data's labels that are required for FRUGAL's performance to plateaus. Let $L$ be 2.5%, 5%, 10%, or 20%, Figure 3 reports FRUGAL's performance on both actionable static warning identification (in AUC) and issues close time prediction (in accuracy). Both metrics are derived from the SOTA's evaluation metrics. Specifically, we pick AUC as the representation metric for actionable static warning analysis since AUC measures the area under the curve whereas other metrics only calculate a single point on the curve. From Figure 3:

- For actionable static warning identification, FRUGAL's performance improves initially and plateaus when $L$ = 10% or 20% across 8 datasets. FRUGAL($L$ = 10%) loses to FRUGAL($L$ = 20%) in only *Lucene* project.
- For issues close time prediction, FRUGAL surprisingly performs statistically similar across all $L \in \{2.5\%, 5\%, 10\%, 20\%\}$.

The same effect is absent in issue close time prediction, this is highly likely due to the balanced nature of the data's class distribution. However, the data in static warning analysis is more imbalanced (with a median of 15% for the actionable static warning class ratio). This is consistent with the motivations for oversampling and undersampling techniques for imbalanced data [2, 11].

**In summary, our answer to RQ1 is:**
From our investigation of various $L$ values, FRUGAL's performance plateaus when $L \geq 10\%$ and FRUGAL's success is not altered by large changes to $L$.

**RQ2: How does FRUGAL perform in actionable static warnings identification?**

Wang et al. [79] proposed the "golden set" features along with the ML study where they employed RF, Decision Tree, and SVM (with the RBF kernel) with median AUC performances at 70%, 64%, and 50%. Yang et al. [88] extensively investigated different deep learners (DNN, CNN, RF, Decision Tree, and SVM) that pushed Wang et al.'s results to new higher watermarks in the area with median AUC performances at 99.5%, 95.9%, and 99.5% with almost 45%, 55%, and 100% relative improvements to the same learner choices as Wang et al. [79]. The default parameters in Weka (used by Wang et al. [79]) are different to those used in SciKit-Learn (used by Yang et al. [88]). For instance, Wang et al.'s SVM used RBF kernel while Yang et al. [88]'s SVM used linear kernel.

Yang et al. [88] proposed the standard linear SVM as the SOTA's actionable static warning identifier. Table 4 reports the comparison of our proposed method FRUGAL, the SOTA's SVM, and the baseline unsupervised learners (CLA & CLAFI+RF) across FAR, recall, and AUC. In those results:

- Standard unsupervised learner CLA/CLAFI+RF performs the worst as their default behavior is clustering based on the median of the data which may not apply for all the data and especially in the static warning analysis. However, CLA's recalls are almost 100% in a few cases (*Mvn, Cass,* and *Commons*) and CLAFI+RF's FAR are almost 0% in more than half cases (*Derby, Lucene, Cass, Jmeter* and *Tomcat*). This indicates promising areas for tuning configurations of unsupervised learners.
- FRUGAL performs similarly to the SOTA's SVM as FRUGAL wins in Recall, loses in AUC, and draws in FAR.
- In term of labelling efforts, CLA is label-free, FRUGAL costs 10%, and Yang et al.'s method costs 100% because FRUGAL and the SOTA require 10% and 100% of the data to be labelled.

**In summary, our answer to RQ2 is:**
FRUGAL improves significantly from standard unsupervised learner CLA with 10% of data labelled as a tradeoff while performing similarly to the SOTA with 90% fewer information. A simple method that explores small regions of data does no worse than methods that extensively learn the whole space.

**RQ3: How does FRUGAL perform in issue close time prediction?**

Mark et al.[50] proposed DeepTriage as SOTA deep learning solution extended from bidirectional LSTMs with an "attention mechanism" to predict issue close time. A Long Short-Term Memory (LSTM) [33] is a form of recurrent neural network that has

**Table 5: Comparison between CLA[60], SIMPLE[91], and FRUGAL in terms of Accuracy, FAR, Recall, and AUC for predicting issue close time. Except for FAR, the higher the results the better the performance of the treatment. Medians and IQRs (delta between 75th and 25th percentile, lower the better) are calculated for easy comparisons. Here, the highlighted cells show best performing treatments.**

| Metrics | Treatment | Chromium | Firefox | Eclipse | Median | IQR |
|---|---|---|---|---|---|---|
| Accuracy ($M$ = 3%) | CLA | 53.6 | 57.1 | 57.6 | 57.4 | 2 |
| | CLAFI+RF | 50.2 | 53.9 | 51.3 | 52.6 | 1.9 |
| | FRUGAL | 65.5 | 74.1 | 67.3 | 70.7 | 4.3 |
| | SIMPLE [91] | 70.3 | 68.3 | 68.8 | 68.6 | 1 |
| FAR ($M$ = 5%) | CLA | 34.9 | 26.9 | 37.3 | 32.1 | 5.2 |
| | CLAFI+RF | 35.1 | 3.1 | 32.9 | 18 | 16 |
| | FRUGAL | 2.2 | 2 | 2 | 2 | 0.1 |
| | SIMPLE [91] | 33.1 | 32.1 | 22.5 | 27.3 | 5.3 |
| Recall ($M$ = 8%) | CLA | 54.9 | 88.9 | 66.8 | 77.9 | 17 |
| | CLAFI+RF | 38.3 | 45.4 | 38.1 | 41.8 | 3.7 |
| | FRUGAL | 99.9 | 97.2 | 96.9 | 97.1 | 1.5 |
| | SIMPLE [91] | 71.7 | 74.1 | 54 | 64.1 | 10.1 |
| AUC ($M$ = 3%) | CLA | 58.8 | 66.4 | 62.2 | 64.3 | 3.8 |
| | CLAFI+RF | 53.4 | 60.6 | 54.2 | 57.4 | 3.6 |
| | FRUGAL | 71.9 | 80.1 | 75.3 | 77.7 | 4.1 |
| | SIMPLE [91] | 67.3 | 70.4 | 65.6 | 68 | 2.4 |

additional “gate" mechanisms to allow the network to model connections between long-distance tokens in the input. Bidirectional variants of recurrent models, such as LSTMs, consider the token stream in both forward and backward directions; this allows for the network to model both the previous and the following context for each input token. Attention mechanisms[5] use learned weights to help the network “pay attention" to tokens that are more important than others in a context.

Yedida et al. [91]’s SIMPLE extended basic 1980s’ style feedforward neural network with state-of-the-art SE’s optimizer DODGE [1] to automatically select the preprocessors (normalizer, binarizer, etc) and the neural network model’s hyperparamters (num_layers, num_units_in_layer, batch_size). SIMPLE outperformed DeepTriage and other non-neural network methods from Marks et al. [51] and Guo et al. [27].

SIMPLE is employed as the SOTA solution for predicting issue close time. Yedida et al. [91] only compared solutions by the accuracy metric. In order to ensure the generalizability of our proposed solution, we also compared different methods with metrics from static warning analysis in RQ2 (FAR, recall, and AUC). Hence, Table 4 reports the comparison of our proposed method’s FRUGAL, the SOTA’s SIMPLE, and the baseline unsupervised learners (CLA & CLAFI+RF) across accuracy, FAR, recall, and AUC. We observe:

- Standard unsupervised learners CLA/CLAFI+RF performed the worst as it’s default behavior is clustering based on the median of the data which may not apply for all the data and especially in the prediction issue close time domain. Surprisingly, CLAFI+RF performs better than CLA in static code warnings but that effect is not seen here which demonstrates what works for one domain may not work for another. Additionally, on average, CLA only underperformed SIMPLE by approximately 11%, 5%, 14%, and 4% in accuracy, FAR, recall, and AUC without access to the train data. Altogether, both points indicate promising areas for tuning configurations of unsupervised learners.
- FRUGAL performs similarly to the SOTA’s SIMPLE as FRUGAL wins in recall, AUC, and FAR while drawing in accuracy. FRUGAL, on average, improves relative SOTA’s accuracy, AUC, and recall by 3%, 14%, and 52% respectively while reducing FAR by 90% relatively.
- In term of labelling efforts, CLA is label-free, FRUGAL costs 10%, and the Yedida et al. [91]’s method costs 100% because FRUGAL and the SOTA need 10% and 100% of the data labelled to execute.

**In summary, our answer to RQ3 is:**

FRUGAL outshined both standard unsupervised learner CLA and the SOTA SIMPLE (EMSE’20 [91] which outperformed a decade of research including ICSE’10 [26], PROMISE’11 [51], MSR’16 [37], COMAD’19 [50]) in predicting issues close time. FRUGAL requires only 10% of the train data to be labelled when being compared against unsupervised learning while using 90% less information than the SOTA tuned deep learning method. Hence, FRUGAL is not only effective in static warning analysis, but also in issue close time prediction. The success in both areas let this study hypothesizes that other areas of SE may also benefit from FRUGAL.

## 6 THREATS OF VALIDITY

There are several validity threats [20] to the design of this study. Any conclusion made from this work must be considered with the following issues in mind:

**Conclusion validity** focuses on the significance of the treatment. To enhance conclusion validity, we run experiments on 12 different target projects across stratified sampling (25 runs) and find that our proposed method always performed better than the state-of-the-art approaches. More importantly, we apply a similar statistical testing of Cohen’d as the SOTA work [88, 91] from the two case studies to obtain fair comparison. In addition, we have taken into generalization issues of single evaluation metrics (e.g., recall and precision) into consideration and instead evaluate our methods on metrics that aggregate multiple metrics like AUC while being effort-aware via cost. As future work, we plan to test the proposed methods with additional analyses that are endorsed within SE literature (e.g., P-opt20 [76]) or general ML literature (e.g., MCC [15]).

One of the possible explanations for the simple effectiveness of both binary split of the output space (CLA/CLAFI+ML/FRUGAL’s centrality) and 10% labelled train data requirement is highly due to the intrinsic dimensionality. Levina and Bickel [43] argued that many datasets embedded in high-dimensional spaces can be compressed without significant information loss (similar to the PCA method [52]). To compute Levina’s intrinsic dimensionality, a 2-d plot is created where the x-axis shows $r$; i.e. the radius of two configurations while the y-axis shows $C(r)$ as the number of configurations after spreading out some distance $r$ away from any of $n$ data instances:

Huy Tu, Tim Menzies
Com Sci, NCState, USA
hqtu@ncsu.edu, timm@ieee.org

**Table 6: Summary of intrinsic dimensions ($D$) of this study's 12 datasets from Levina and Bickel [43].**

| | Static Code Warnings | | | | | | | | | Issue Close Time | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Derby | Mvn | Lucene | Phoenix | Cass | Jmeter | Tomcat | Ant | Commons | Chromium | Firefox | Eclipse |
| $D$ | 0.78 | 1.10 | 0.15 | 0.62 | 1.94 | 1.54 | 0.73 | 0.82 | 1.04 | 1.95 | 2.10 | 1.9 |

$$y = C(r) = \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} I\left[\|x_i, x_j\| < r\right] \quad (3)$$

The maximum slope of $\ln C(r)$ vs. $\ln r$ is then reported as the intrinsic dimensionality, $D$. Note that $I[\cdot]$ is the indicator function (i.e., $I[x] = 1$ if $x$ is true, otherwise it is 0); $x_i$ is the $i$th sample in the dataset. Applying this calculation to the 12 datasets of two study cases (reports in Table 6), we found the intrinsic or latent dimensionality ($D$) of our data is very low (median around one, no more than three). Agrawal et al.'s DODGE [1] is the SOTA optimizer for SE, DODGE executes by binary splitting the tuning space, each chop moves in the bounds for numeric choices by half the distance from most distant value to the value that produced the "best" performance. According to Agrawal et al., DODGE's effectiveness roots in how the performance score generated from SE data can be divided into a few regions (low dimensional). FRUGAL's central function of binary splitting is similar to DODGE as FRUGAL compresses the data dimensions (features) via aggregated percentile $C$ and survey the whole space by varying $C$ ({5% to 95% increments by 5%}). Menzies et al. [53] and Hindle et al. [32] also reported on how several SE data are low dimensional and the benefits from building effective tools from such data. This work extends those findings: the labelling efforts to commission to tools building can be reduced greatly because of the low dimensionality of SE data.

**Internal validity** focuses on how sure we can be that the treatment caused the outcome. To enhance internal validity, we heavily constrained our experiments to the same dataset, with the same settings, except for the treatments being compared.

**Construct validity** focuses on the relation between the theory behind the experiment and the observation. To enhance construct validity, we compared solutions with and without our strategies in Table 4 and 5 while showing that both components (unsupervised learning with CLA/CLAFI+ML [60] and tuned semi-supervised method of FRUGAL) and in various amounts of labelled data required for the proposed mehtod to improve the overall performance. However, we only show that with our default parameters settings of random forest learner. The performance can get even better by tuning the parameters, employing different learners (e.g., deep learners), and introducing a variety of data preprocessors (e.g., synthetic minority over-sampling or SMOTE that is known to help with imbalanced datasets [2, 11] like our static code warnings study case). We aim to explore these in our future work.

**External validity** concerns how widely our conclusions can be applied. In order to test the generalizability of our approach, we always kept a project as the holdout test set and never used any information from it in training. Moreover, we have validated our proposed method on two important software analytics case studies: actionable static code warnings identification and issues close time prediction. Our experiments with default CLA/CLAFI+ML [60] demonstrates the danger of treating all data with the state-of-the-art method, especially when switching domain (from defect prediction to issue close time prediction and actionable static code warning identification).

## 7 CONCLUSION AND FUTURE WORK

There is much recent advance for software analytics research with automated and semi-automated methods. However, these methods are built on a sufficiently large amount of data labelled. Generating such labels can be labor-intensive and expensive (as discussed in §3). Such requirement can introduce barrier for entering new research domains (e.g., the success of open-source projects). In order to reduce the label famine and human effort, FRUGAL is recommended. FRUGAL tunes the state-of-the-art unsupervised learner from defect prediction (CLA/CLA+ML/CLAFI+ML) and it's corresponding percentile parameter $C$ in the grid search manner while validating on only 10% of the labelled data. Our findings include:

(1) Unsupervised Learners without access to the train data's labels performed only 10% less than the SOTA methods on average. The results are promising but still not effective enough.
(2) FRUGAL performed similarly to the SOTA actionable static code warning identifier while surpassing the SOTA issue close time predictor with 90% less information.
(3) FRUGAL reduced the labelling efforts needed for the software analytics tools by 90%. Simply, FRUGAL is 10 times cheaper than the SOTA methods in issue close time and static code warnings analysis areas.
(4) The success of FRUGAL for the two case studies here suggests that many more domains in software analytics could benefit from unsupervised learning. As mentioned above, those benefits include the ability to commission new models with less human efforts and costs. By restricting human involvement in the process, we also reduced erroneous labels that can cascade to the whole research community since human are still error-prone (Yu et al. [92] found 98% of the false-positive labels within Maldonado and Shihab [48] were actually true-positive labels).
(5) Overall, our proposed method restated the benefit in exploring low dimensional SE data [1, 32, 53, 88, 91] and extended their findings that the labelling efforts can be reduced greatly because of the low dimensionality of SE data.

That said, FRUGAL still suffers from the validity threats discussed in §6. To further reduce those threats and to move forward with this research, we propose the following future work:

- Test whether replacing the Random Forest model in FRUGAL with a deep learning model will further improve its performance.
- Explore non-SE or high-dimensional SE data with FRUGAL to see if our current conclusions still hold.
- Apply non-trivial hyper-parameter tuning (e.g., DODGE [1] or FLASH [59]) on various data preprocessors and machine learners with FRUGAL to test whether tuning can further improve the performance.
- Extend the work to other software engineering domains (e.g., security [24], technical debts [48], software configurations [19], etc) and compare it with other state-of-the-art methods which continue to appear.

## ACKNOWLEDGEMENTS

This work was partially funded by blinded for review.

## REFERENCES

[1] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies. How to "dodge" complex software analytics. *Preprint, IEEE Transactions on Software Engineering*, 2019. Available on-line at http://arxiv.org/abs/1902.01838.

[2] Amritanshu Agrawal and Tim Menzies. Is" better data" better than" better data miners"? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1050–1061. IEEE, 2018.

[3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *ICSE*, 2019.

[4] Pavel Avgustinov, Arthur I Baars, Anders S Henriksen, Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. Tracking static analysis violations over time to capture developer characteristics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 437–447. IEEE Press, 2015.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429. IEEE, 2012.

[7] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *2008 IEEE International Conference on Software Maintenance*, pages 277–286. IEEE, 2008.

[8] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. *Cytometry*, 1987.

[9] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[10] G. Catolino. Just-in-time bug prediction in mobile applications: The domain matters! In *MOBILESoft*, 2017.

[11] 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.

[12] D. Chen, W. Fu, R. Krishna, and T. Menzies. Applications of psychological science for actionable analytics. In *FSE*, 2018.

[13] Di Chen, Kathyrn T Stolee, and Tim Menzies. Replication can improve prior results: A github study of pull request acceptance. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 179–190. IEEE, 2019.

[14] Wei-Chou Chen, Shian-Shyong Tseng, and Ching-Yao Wang. A novel manufacturing defect detection method using association rule mining techniques. *Expert systems with applications*, 29(4):807–815, 2005.

[15] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 2020. doi: 10.1186/s12864-019-6413-7.

[16] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, André Batista da Silva, and Rodrigo Oliveira Spínola. A contextualized vocabulary model for identifying technical debt on code comments. In *MTD*, 2015.

[17] Mário André de Freitas Farias, José Amâncio Santos, Marcos Kalinowski, Manoel Mendonça, and Rodrigo Oliveira Spínola. Investigating the identification of technical debt through code comment analysis. In *ICEIS*, 2016.

[18] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *IST*, 2020.

[19] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 2005.

[20] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research-an initial survey. In *SEKE*, 2010.

[21] W. Fu and T. Menzies. Easy over hard: A case study on deep learning. In *FSE*, 2017.

[22] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 72–83. ACM, 2017.

[23] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135 – 146, 2016. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2016.04.017. URL http://www.sciencedirect.com/science/article/pii/S0950584916300738.

[24] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 11–20, 2010. doi: 10.1109/MSR.2010.5463340.

[25] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 37th ICSE*.

[26] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56, 2010.

[27] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 495–504, 2010.

[28] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 9.6 million links in source code comments: Purpose, evolution, and decay. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1211–1221. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00123. URL https://doi.org/10.1109/ICSE.2019.00123.

[29] 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.

[30] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

[31] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. MSR, 2008.

[32] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[34] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying self-admitted technical debt in open source projects using text mining. *EMSE*, 2018.

[35] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[36] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *TSE*, 2013.

[37] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using dynamic and contextual features to predict issue lifetime in github projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 291–302, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901751. URL https://doi.org/10.1145/2901739.2901751.

[38] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans SE*, 2008.

[39] Ekrem Kocaguneli, Tim Menzies, Jacky Keung, David Cok, and Ray Madachy. Active learning and effort estimation: Finding the essential content of software effort estimation data. *IEEE Transactions on Software Engineering*, 39(8):1040–1053, 2012.

[40] Ekrem Kocaguneli, Tim Menzies, Jacky Keung, David Cok, and Ray Madachy. Active learning and effort estimation: Finding the essential content of software effort estimation data. *IEEE Transactions on Software Engineering*, 39(8):1040–1053, 2013.

[41] 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.

[42] Youngseok Lee, Suin Lee, Chan-Gun Lee, Ikjun Yeom, and Honguk Woo. Continual prediction of bug-fix time using deep learning-based activity stream embedding. *IEEE Access*, 8:10503–10515, 2020.

[43] Elizaveta Levina and Peter Bickel. Maximum likelihood estimation of intrinsic dimension. *Advances in neural information processing systems*, 17:777–784, 2004.

[44] 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.

[45] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li. Satd detector: a text-mining-based self-admitted technical debt detection tool. In *ICSE*, 2018.

[46] Robyn R Lutz and Inés Carmen Mikulski. Empirical analysis of safety-critical anomalies during operations. *IEEE Transactions on Software Engineering*, 30(3): 172–180, 2004.

[47] Suvodeep Majumder, Pranav Mody, and Tim Menzies. Revisiting process versus product metrics: a large scale analysis. *arXiv preprint arXiv:2008.09569*, 2020.

[48] E. da S Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *MTD*, 2015.

[49] E. da S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *TSE*, 2017.

[50] Senthil Mani, Anush Sankaran, and Rahul Aralikatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *COMAD'19: ACM India Joint International Conference on Data Science and Management of Data*, pages 171–179,

2019.

[51] Lionel Marks, Ying Zou, and Ahmed E Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, pages 1–8, 2011.

[52] Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (pca). *Computers & Geosciences*, 1993. ISSN 0098-3004. doi: https://doi.org/10.1016/0098-3004(93)90090-R. URL https://www.sciencedirect.com/science/article/pii/009830049390090R.

[53] Tim Menzies, David Owen, and Julian Richardson. The strangest thing about software. *Computer*, 40(1):54–60, January 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.37. URL https://doi.org/10.1109/MC.2007.37.

[54] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[55] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[56] A. Mockus and L. Votta. Identifying reasons for software changes using historic databases. In *ICPC*, 2000.

[57] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6): 3219–3253, Dec 2017.

[58] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[59] Vivek Nair, Zhe Yu, and Tim Menzies. Flash: A faster optimizer for sbse tasks. *arXiv preprint arXiv:1705.05018*, 2017.

[60] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets. In *ASE 2015*, 2015.

[61] M. Nayrolles and A. Hamou-Lhadj. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *MSR*, 2018.

[62] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi: 10.1109/TSE.2020.3001739.

[63] 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.

[64] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 2011.

[65] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *ICSME*, 2014.

[66] C. Rosen, B. Grawi, and E. Shihab. Commit guru: Analytics and risk prediction of software commits. ESEC/FSE 2015, 2015.

[67] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[68] S. Sawilowsky. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods*, 8:26, 2009.

[69] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 2015. doi: https://doi.org/10.1016/j.neunet.2014.09.003. URL https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[70] Burr Settles. Active learning literature survey. 2009.

[71] Rui Shu, Tianpei Xia, Jianfeng Chen, Laurie Williams, and Tim Menzies. Improved recognition of security bugs via dual hyperparameter optimization. *EMSE*, 11 2019.

[72] M Six Silberman, Bill Tomlinson, Rochelle LaPlante, Joel Ross, Lilly Irani, and Andrew Zaldivar. Responsible research with crowds: pay crowdworkers at least minimum wage. *Communications of the ACM*, 61(3):39–41, 2018.

[73] J. AK Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 1999.

[74] Ferdian Thung, David Lo, Lingxiao Jiang, Foyzur Rahman, Premkumar T Devanbu, et al. To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. *Automated Software Engineering*, 22 (4):561–602, 2015.

[75] Huy Tu, Rishabh Agrawal, and Tim Menzies. The changing nature of computational science software, 2020.

[76] Huy Tu, Zhe Yu, and Tim Menzies. Better data labelling with emblem (and how that impacts defect prediction). *IEEE Transactions on Software Engineering*, 2020.

[77] B. Vasilescu. Personnel communication at fse'18, 2018.

[78] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *FSE*, 2015.

[79] Junjie Wang, Song Wang, and Qing Wang. Is there a golden feature set for static warning identification?: an experimental evaluation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 17. ACM, 2018.

[80] I Witten, Eibe Frank, M Hall, and C Pal. : Data mining: practical machine learning tools and techniques. elsevier inc. 2017.

[81] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[82] Tianpei Xia, Wei Fu, Rui Shu, and Tim Menzies. Predicting project health for open source projects (using the decart hyperparameter optimizer), 2020.

[83] Zhou Xu, Li Li, Meng Yan, Jin Liu, Xiapu Luo, John Grundy, Yifeng Zhang, and Xiaohong Zhang. A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software*, 172: 110862, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110862. URL https://www.sciencedirect.com/science/article/pii/S0164121220302521.

[84] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*, pages 344–353. IEEE, 2017.

[85] Jun Yang and Hongbing Qian. Defect prediction on unlabeled datasets by using unsupervised clustering. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 465–472. IEEE, 2016.

[86] X. Yang, D. Lo, X. Xia, and J. Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *IST*, 2017. URL http://www.sciencedirect.com/science/article/pii/S0950584917302501.

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

[88] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. How to recognize actionable static code warnings (using linear svms). In *EMSE*, 2020.

[89] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. FSE, 2016. URL http://doi.acm.org/10.1145/2950290.2950353.

[90] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168. ACM, 2016.

[91] Rahul Yedida, Xueqi Yang, and Tim Menzies. When simple is better than complex: A case study on deep learning for predicting bugzilla issue close time. In *EMSE*, 2021.

[92] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies. Identifying self-admitted technical debts with jitterbug: A two-step approach. *TSE*, 2020. doi: 10.1109/TSE.2020.3031401.

[93] Zhe Yu, Christopher Theisen, Laurie Williams, and Tim Menzies. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering*, 2019.

[94] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Transactions on Software Engineering*, 2021.

[95] F. Zampetti, A. Serebrenik, and M. Di Penta. Automatically learning patterns for self-admitted technical debt removal. In *SANER*, 2019.

[96] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.

[97] Yuming Zhou, Yibiao Yang, Hongmin Lu, L. Chen, Yanhui Li, Y. Zhao, J. Qian, and B. Xu. How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Trans. Softw. Eng. Methodol.*, 27:1:1–1:51, 2018.