Noname manuscript No.
(will be inserted by the editor)

Simplifying Software Defect Prediction (via the "early bird" Heuristic)

Shrikanth N.C. · **Tim Menzies**

Received: date / Accepted: date

Abstract Before researchers rush to reason across all available data or try complex methods, perhaps it is prudent to first check for simpler alternatives. Specifically if the historical data has most information in some small region, then perhaps a model learned from that region would suffice for the rest of the project.

To support this claim, we offer a case study with 240 GitHub projects, where find that the information in those projects "clumped" towards the earliest parts of the project. A defect prediction model learned from just the first 150 commits works as well, or better than state-of-the-art alternatives. Using just this early life cycle data, we can build models very quickly, very early in the software project lifecycle. Using this method we have can also show that a very simple model (with just two features) generalizes to hundreds of software projects.

Based on this experience, we doubt that prior work on generalizing software engineering defect prediction models may have needlessly complicated an inherently simple process. Further, prior work that focused on later-life cycle data now needs to be revisited since their conclusions were drawn from relatively uninformative regions.

Replication note: all our data and scripts are online at https://github.com/snaraya7/simplifying-software-analytics

Keywords software analytics · quality assurance · defect prediction · early · early-data-lite

Shrikanth N.C., Tim Menzies
Department of Computer Science, North Carolina State University, Raleigh, NC, USA.
E-mail: nc.shrikanth@gmail.com, and timm@ieee.org

1 Introduction

In defect prediction, researchers often fail to try simpler methods before trying complex analytics [103, 140]. For example:

- "For many new projects we may not have enough historical data to train prediction models" [91]
- "At least for defect prediction, it is no longer enough to just run a data miner and present the result without conducting a tuning optimization study." [25]
- "Many research studies have shown that ensemble learning can achieve much better classification performance than a single classifier" [129]

While we do not doubt those results in that context, we show in this work that we can simplify software defect prediction if we focus more on the quality of the data than on applying various software analytic techniques. Furthermore, this paper shows that much of the prevalent methods used to build defect predictors are needless and can be 'simplified' to a large extent with knowledge-rich early project data.

By simplified we mean:

- Stable conclusions (no need to update defect predictors for every new project release).
- Less run-time (With fewer project data we can find and build relevant predictors faster).
- Better explanations (With fewer features we can communicate the oracles (predictors) decisions to stakeholders effectively).

In our work, we have sometimes seen unsatisfactory results from such a complex approach. For example, once we tried learning what we could from 700,000+ commits. The web slurping required for that process took nearly 500 days of CPU, using five machines with 16 cores, over seven days (since the data from those projects had to be cleaned, massaged, and transformed

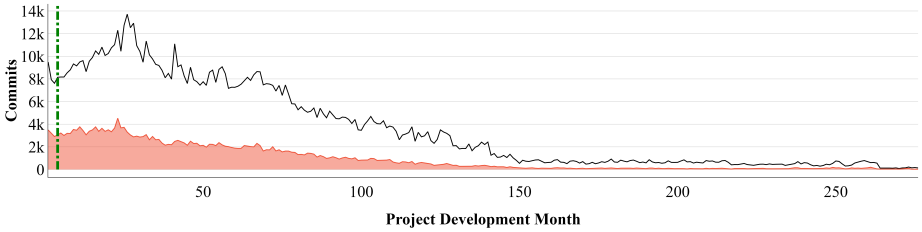


Figure 1.A: 155 popular GitHub projects (#stars> 1000). Data from 1.2 million commits.

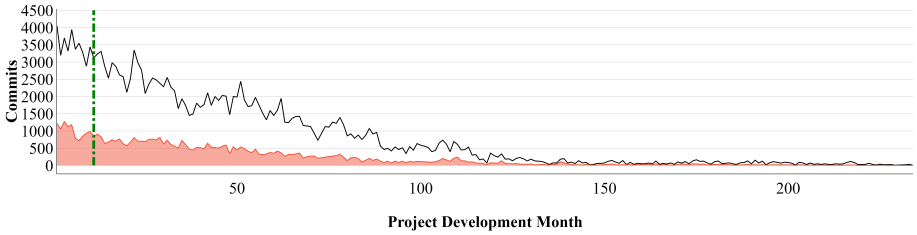


Figure 1.B: 85 unpopular GitHub projects (#stars< 1000). Data from 253,289 commits.

Fig. 1: Most defective commits occur early in the life cycle. Black:Red = Clean:Defective commits. In this paper, we compare (a) models learned up to the vertical green (dotted) line to (b) models learned using more data.

into some standard format). Within that data space, we found significant differences in the models learned from different parts of the data. So even after all that work and over a year of CPU, we were unable to report a stable conclusion to our business users.

If “too much data” can be too much, when is enough data just enough to build effective defect predictors? Our answer to this question comes from a previously unreported effect is shown in Figure 1. As shown in this figure, when we look at the percent of buggy commits in GitHub projects, a remarkable pattern emerges. Specifically, most of the buggy commits occur earlier in the life cycle.

This observation prompted an investigation of a “early-data-lite” approach that just uses early life cycle data to predict for defects. The literature review of §4 shows that, surprisingly, this approach to defect prediction has been overlooked by prior work [103]. This is a significant oversight since, in 240 GitHub projects, we can show that defect predictors learned from the first 150 commits work as well, or better, than state-of-the-art alternatives.

Overall, the contributions of this paper are to show:

- The information within projects may not be evenly distributed across the life cycle. For such data, it can be very useful to adopt a “early-data-lite” approach.
- For example, using early life cycle data, we found simple models (with only two features) that generalize across hundreds of projects Such models can be built much faster than traditional methods (weeks versus months of CPU time).
- So before researchers use all available data, they need to first check that their (e.g.) buggy commit data occurs at equal frequency across the life cycle. We say this since much prior work on methods for learning from multiple projects [9, 14, 15, 19–23, 26, 32–35, 37, 38, 41–43, 52, 53, 57–60, 63, 64, 73, 74, 77, 81, 86–89, 91, 95–98, 115, 119, 120, 123, 124, 130, 135–138, 140] needlessly complicated an inherently simple process.

The rest of this paper explains our method; and offers experimental evidence that this early life cycle method works better than sophisticated methods like classifier tuning, ensemble and notably made transfer learning algorithms work simpler, faster, and more stable. This paper uses the abbreviations of Table 1.

Table 1: Table of Important Acronyms

Acronym	Abbreviation
AUC	Area under the receiver operating characteristic curve
CFS	Correlation-based Feature Selection
DODGE	Optimizer proposed by Agrawal et al. in [1]
DT	Decision Tree
HPO	Hyperparameter optimization
HYPEROPT	Optimizer proposed by Bergstra et al. [7]
IFA	Initial Number of False Alarams
KNN	k-nearest neighbors algorithm
LA and LT	Refer to features list in Table 4
LR	Logistic Regression
MCC	Matthews correlation coefficient
NB	Naive Bayes classifier
PF	False Alaram Rate
RF	Random forest
SMOTE	Synthetic Minority Over-sampling Technique [16]
SVM	Support vector machines
SZZ	Sliwerski Zimmerman Zellar Algorithm [106]
TCA	Transfer Component Analysis [74]
TLEL	Two-layer Ensemble Learning Algorithm [129]
TPE	Tree-structured Parzen Estimator [7]

Before all that, we digress to address the obvious objections to our conclusion. The results presented here are only for defect prediction. In future work, we need to test if our results hold for other domains.

2 Connection to Prior Work

Previously, we have reported the Figure 1.A results at ICSE'21 [103][1]. We calculate that only Figure 1.A and 5 pages of this text (e.g. §8.1) come from that prior work.

As to the semantic difference of this paper to prior work, that prior study was more limited since it did not have the additional data of Figure 1.B. Also, that prior study only ran some within-project sampling methods on the Figure 1.A data. This paper takes the additional step of comparing our methods to transfer learning, classifier tuning and ensemble methods. Further, previously, we did not report the model learned via this method. We show here that a very simple model (that uses just a few variables) can be learned from a few early life cycle samples. Lastly, as shown by the new results of this paper in Table 7, Table 8, Table 9 and Table 10 we can out-perform that results of that prior work.

3 Background

3.1 About Defect Prediction

The case studies of this paper are based on defect prediction. Hence, before doing anything else, we need to introduce that research area.

Fixing software defects is not cheap [28]. Accordingly, for decades, SE researchers have been devising numerous ways to predict software quality before deployment. One of the oldest studies was made in 1971 by Akiyama using size-based defect predictions for a system developed at Fujitsu, Japan [4]. This approach remains popular today. A 2018 survey of 395 practitioners from 33 countries and five continents [20] found that over 90% of the respondents were willing to adopt defect prediction techniques [117].

Defect prediction uses data miners to input static code attributes and output models that predict where the code probably contains most bugs [65, 78]. The models learned in this way are very effective and relatively cheap to build:

- *Effective*: Misirili et al. [68], and Kim et al. [45] report considerable cost savings when such predictors are used in guiding industrial quality assurance processes.
- *Relatively simpler to implement* Also, Rahman et al. [90] show that such predictors are competitive with more elaborate approaches. For example, they note that static code analysis tools can have expensive licenses that need to be updated after any major language upgrade. Defect predictors, on the other hand, can be quickly implemented via some lightweight parsing of project data.

Our prior work [103] proposed numerous extensions that improve the validity and scope of the "early-data-lite" method. Therefore, in this work we continue to assess the efficacy and applicability of "early-data-lite" method broadly on two active areas in defect prediction, specifically:

- **Data:** Transfer Learning (Cross-project defect prediction)
- **Technique:** Tuning and Ensemble methods

[1] For reviewers, we note that that paper is available on-line at https://arxiv.org/pdf/2011.13071.pdf

3.2 Data: Transfer Learning

Defect predictors are learned from project data. What happens if there is not enough data to learn those models? This is an especially acute problem for newer projects (and in the absence of historical data in some legacy projects [13]).

In such scenarios, practitioners and researchers might identify matured projects that share some similarities to their local projects. Once found, then lessons learned could be *transferred* from the older to the new project. There are kinds of transfer:

- *Cross*: *cross-project defect prediction*. Lessons learned from *other* projects and applied to *this* project.
- *Within*: *within-project defect prediction*. Using data from *this* project, lessons learned from prior experience is used to makes predictions about later life cycle development.

To say the least, transfer learning is a very active research area in software engineering (SE). We can find more than 1,000 articles in the last five years alone (found using the query ”cross-project defect prediction,” [2]). By our count, within that corpus, there have been at least two dozen transfer learning methods [5]. Interesting methods evolved in that research include:

- Heterogeneous transfer that lets data expressed indifferent formats transferred from project to projects [73];
- Temporal transfer learning, which is a within-project defect prediction (*Within*) tool where earlier life cycle data is used to make predictions later in the life cycle [47].

Our reading of the literature is that, apart from our own research, the prior state-of-the-art in the SE literature is Nam et al.’s TCA+ (Transfer Component Analysis) method [75]. For a list of important abbreviations used in this paper see Table 1. Given data from some source and target project, TCA strives to “align” the source and target data via dimensionality rotation, expansion, and contraction. TCA+ is an extension to basic TCA that used automatic methods to find normalization options for TCA.

3.3 Technique : Tuning and Ensemble

Classification algorithms can be trained to classify a project commit as defective. Studies have shown their predictive performance can depend upon the set of hyper-parameter they are initially configured [1, 25].

For example, the k-nearest neighbors (KNN) that we explore in this study is available in the widely used scikit-learn [84] machine learning library. And KNN is set to the following default parameters based on standard machine learning literature as follows:

$n\_neighbors = 5, *, weights = \acute{u}niform\acute{}, algorithm = \acute{a}uto\acute{}, leaf\_size = 30, p = 2, metric = \acute{m}inkowski\acute{}, metric\_params = None, n\_jobs = None$

However, as seen from the numerous parameters available for KNN, there is a huge parameter space of options the KNN can be tried and tested (tuned). Therefore most machine learning algorithm’s hyper-parameters can be tuned using training data before tested on project releases in our case.

There are numerous ways to find the right set of parameters for a classifier given the training data. But the decision predominantly comes with the run-time cost. For example,

[2] Queried https://scholar.google.co.in/ in 2020

searching through all available options ‘Manual Search’ but it may not terminate. One may try ‘Random Search’ with a terminating condition but the probability of finding the near best parameter may be low. Fu et al. [25] explored ‘grid-search’ a baseline hyper-parameter approach in the field of machine learning [25] but found it to be very slow for defect prediction. Recently (2019) Agrawal et al. proposed a novel optimizer that terminates quickly after finding near optimal hyper-parameters for defect predictors [1]. More details on DODGE will be presented in §5.1.1. Therefore this paper will explore DODGE with early methods.

Another avenue of complex approach is the use of ensemble method, where the philosophy is why use just ‘one’ when ‘more’ is better. Numerous studies have shown defect predictors have shown significant improvement when a ensemble of classifiers were used rather than just one [31, 118, 129]. In 2017, Yang et al. proposed a two-layer ensemble approach for defect prediction called TLEL (Two-layer Ensemble Learner) that showed promising improvements when compared to predictors single classifiers. §5.1.1 elucidates its operation and this paper will explore TLEL as part of the complex methods to test the efficacy of early methods.

3.4 Issues with current approaches

Nevertheless, just because a technique is popular does not mean that it should be recommended. Our reading of the literature is most recent SE analytics papers have taken a complex approach (e.g. see the quotes in our prior work [103]) and use state of the art analytics without testing with simpler alternatives [140]. We argue there that it can be useful to try early-data-lite before adopting complex techniques that demand more data, delay analytic, does not offer explanations and utilize system resources. Because, for one thing, all that data might not be available. The availability of more data in an industrial setting is not assured. Also it may not be useful to learn from more data. Proprietary data may not be readily available for practitioners to build predictors for their local projects. Zimmermann et al. showed that transferring predictors from the same domain does not guarantee quality predictions [141]. Finally, due to privacy concerns, teams even within the same organization may not readily make their matured project available for others to use [86].

For another thing, several researchers have reported that a complex approach can be problematic:

- In our introduction, we reported on our own issues seen when learning from 700,000+ GitHub commits.
- At her ESEM’11 keynote address, Elaine Weyuker questioned that she will ever have the option to make the AT&T information public [122].
- In over 30 years of COCOMO effort, Boehm could share cost estimation data from only 200 projects.

Next, applying techniques discussed in §5.2.1 comes with a CPU run-time cost. When time to time researchers have strive and endorsed to look for simpler alternatives it is important that we explore them.

4 Related Work

In our prior we showed that historical data within the project were used in large quantities to build defect predictors were needless [103]. And as promised in our prior work in this

work we check the validity of our prior conclusion beyond within-project software defect prediction and test the efficacy of early methods in various contexts such as:

- Unpopular SE projects
- Transfer learning scenarios
- Hyper-parameter optimization and Ensemble approach

One reason to explore early life cycle data reasoning is that it has not been done before. Much of the SE transfer learning methods [14, 15, 20, 22, 26, 37, 38, 41–43, 52, 53, 58, 73, 74, 77, 81, 86–88, 95–98, 115, 119, 124, 130, 136, 138, 140] and Tuning or ensemble methods [1–3, 11, 12, 17, 25, 27, 39–41, 43, 43, 54–56, 59, 80, 81, 83, 88, 92, 93, 96–98, 100, 105, 107–109, 112–114, 118–121, 124, 126, 129, 133, 136, 139] could be characterized as "late-data" since they transferred all available data from one or more projects to build their predictors.

4.1 Literature Survey

To understand more about late-data reasoning in SE, we queried Google Scholar [3] to find defect prediction articles in three areas:

- We found 982 articles in Google Scholar using the query ("cross-project defect prediction") in the last ten years.
- We found more than 1,740 articles in the last five years alone (searching the query "defect prediction" AND "(tuning OR hyper-paramater optimization) and 3,050 articles using the query "defect prediction" AND "ensemble".

Following the advice of Agrawal et al. [3], and Mathews et al. [61] we focused only on "highly cited" papers, i.e., those with more than ten citations per year. Reading those papers, and after discarding papers pure of survey nature (or of not defect prediction space) we filtered the papers that performed some transfer learning experiments or a complex method (ensemble or tuning). We summarized the results of this literature survey in Table 2 (Transfer learning) and Table 3 (Tuning and Ensemble methods).

Within those three sets, we found three approaches to row selection:

- 'All' if the methods of that paper use all rows from one or more projects to build defect predictors.
- 'Part' if the methods of that paper explore a large search space of one or more projects to find a small set of rows that are worthy of transfer data.
- In one case, in 2016, we also found a 'None' approach that used no training but just clustered the test data to find outliers, which were then labeled as bugs [137]. Having recorded that method, this paper will not explore this minority approach.

Note that regardless of being "All" or "Part", the analysis looks at the data across the entire life cycle before return some or all of it.

As to other kinds of data selection, for all these papers, we counted:

- The number projects used in those study;
- The number of features used by their predictors.
- The applied any ensemble or tuning or both techniques.

[3] https://scholar.google.co.in/ in 2020

Table 2: 50 'highly cited' (more than 10 citations per year) papers that ran one or more *Cross* experiment(s) since 2010.

Year	#Data	#Features	Projects	Cites/Year	Paper
2011	■	20	2	16.44	[64]
2012	■	198	6	19.25	[57]
	■	17	10	39.13	[60]
	■	20	10	30.13	[34]
	■	20	7	23.38	[63]
	■	20	2	13.88	[9]
	■	8	9	25.38	[91]
2013		16	41	14	[115]
		17	8	48.43	[74]
		20	41	22.43	[88]
		20	10	19.57	[14]
		20	10	13.29	[86]
	■	20	14	13.29	[35]
	■	20	10	11.43	[33]
	■	54	12	32	[89]
2014		14	11	16.5	[26]
		20	10	20.5	[81]
	■	26	1398	18.5	[135]
2015		18	7	13.4	[97]
		20	11	19.8	[20]
		20	10	14.4	[138]
		20	17	12.4	[87]
		20	10	11.2	[15]
		28	11	24.4	[42]
	■	20	10	33	[32]
2016	None	**X**	26	35.5	[137]
		14	11	26.5	[43]
		14	6	19.75	[130]
		17	10	23.5	[96]
		20	10	35.5	[124]
		20	21	20.75	[41]
		20	30	11.75	[95]
		26	1390	16.25	[136]
		61	23	10.5	[53]
	■	**X**	10	70.25	[120]
2017		20	15	21.33	[98]
		61	34	80	[73]
		61	8	10.67	[77]
2018		6	58	21.5	[140]
		14	6	20.5	[22]
		20	11	23	[37]
		61	18	18.5	[52]
		61	28	14.5	[58]
		20	10	14	[119]
	■	61	16	19.5	[123]
	■	**X**	10	20	[23]
2019		14	6	17	[38]
	■	61	8	14	[19]
	■	20	7	20	[21]
	■	20	14	19	[59]

KEY: ■ Part, □ All, **None** - No training data, **X** - Features automated

When we tried to place this paper into Table 2 and Table 3, we found that our approach was, quite literally, off the charts. The methods advocated by this paper are neither "whole" nor "part" since we learn from early data, then stop collecting (so unlike all the research in Table 2 and Table 3, we never look at all the data). Further, our "number of projects"=1 (which does not even appear in Table 2) and we only use a handful of features (far less than the features used by other work in Table 2).

Table 3: 44 ‘highly cited’ (more than 10 citations per year) papers that built defect prediction models applying methods based on ensemble/tuning/both in the past decade (2011 to 2021).

Year	Cites/Year	Type	Data	Projects	Paper	Year	Cites/Year	Type	Data	Projects	Paper
2011	37.9		All	2	[11]	2013	25		All	6	[40]
2012	17.33		All	14	[108]	2014	33		All	6	[133]
2013	54.13		All	10	[118]	2014	36.86		All	20	[100]
2013	24		All	41	[88]	2016	15.8		All	1,390	[136]
2014	22		All	10	[81]	2016	33.4		Part	17	[25]
2015	53.33		All	29	[27]	2016	79.8		Part	10	[120]
2015	13.33		All	7	[97]	2017	29.25	TUNING	All	6	[54]
2015	18.17		All	6	[105]	2017	47.25		Part	7	[56]
2016	29.2		All	11	[43]	2018	37		All	9	[3]
2016	22.4		All	21	[41]	2018	44		All	8	[2]
2016	15.8		All	12	[121]	2019	14		All	10	[1]
2017	31	ENSEMBLE	All	6	[129]	2019	18.5		Part	14	[59]
2017	22.5		All	15	[98]	2020	28		All	32	[43]
2017	16.25		Part	11	[92]	2015	32.67		All	7	[109]
2018	32.67		All	27	[107]	2015	44		All	6	[55]
2018	21.33		All	15	[39]	2016	37.2		All	10	[124]
2018	27.67		All	12	[114]	2016	50.8		All	18	[112]
2019	26		All	44	[126]	2016	24.4	BOTH	All	10	[96]
2019	21.5		Part	10	[83]	2018	31		All	18	[12]
2020	28		All	2	[93]	2018	19.67		All	6	[119]
2020	31		All	3	[17]	2018	47.67		Part	18	[113]
2020	20		All	12	[80]	2019	13.5		All	25	[139]

KEY: ▀ Part, ▢ All

Hence we assert, with some confidence, that the methods of this paper have not been previously explored.

4.1.1 Representative Techniques

In order to design an appropriate experiment, we used the following guidelines.

Firstly, we are comparing the early sample to sampling over a larger space of project data. Hence, in the following, we will show *early* versus *all* experiments.

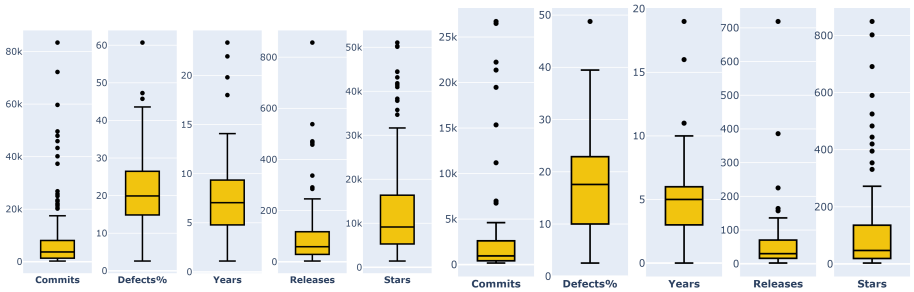
Secondly, there are two ways to find data: *within-project* and *cross-project*. Therefore, we will divide the *early* early-data and *all* late-data experiments into:

- early-data: early-within and early-cross
- late-data: all-within and all-cross
- Complex Methods : Optimizers (DODGE and HyperOpt) and ensemble method (TLEL).

Lastly, looking into the literature, we can see some clear state-of-the-art algorithms that should be represented in our study (specifically, the $TCA+$ and $Bellwether$ cross-project learning methods [53, 74]). Accordingly, when exploring *cross-project* learning, we will employ those methods.

To explain the exact methods used in this study requires some further details on those algorithms. Please see §5.1.1 for the algorithm details and for a discussion of what algorithms we selected. But, in summary, to the best of our knowledge, the early-data variants of these techniques have not been explored before in SE.

We understand the above areas does not cover all of the defect prediction literature but to the best of our knowledge we assert that we covered a range of active research areas within defect prediction to test the scope of our conclusion about early methods.



Distributions seen in all 1.2 millions commits of all 155 *popular* projects: median values of commits (3,728), percent of defective commits (20%), life span in years (7), releases (59) and stars (9,149).

(a) Popular projects (stars $>$ *1000*).

Distributions seen in all 258,000+ commits of all 89 *unpopular* projects: median values of commits (957), percent of defective commits (18%), life span in years (5), releases (30) and stars (47).

(b) Unpopular projects (stars $<$ *1000*).

Fig. 2: Data distributions.

5 Experimental Methods

The rest of this paper offers an experimental evaluation of *early life cycle* methods versus other learning policies for the data of Figure 1.

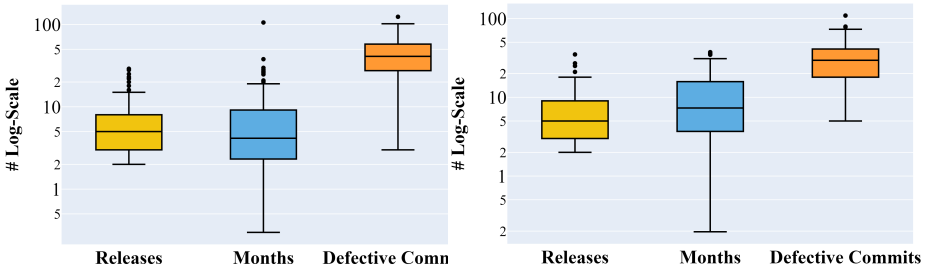
5.1 Data

All data used in this study comes from open source projects hosted on GitHub that other SE researchers can replicate. For details on that data, see Table 4, Figure 2 and Figure 3.

To mine this data, we looked into the dataset curated by Munaiah et al. [70] that has numerous criteria to differentiate an engineering project from a trivial one. Then we mined all the projects using Commit-Guru [94] similar to this work [103]. Projects were rejected according to the standard sanity checks by SE GitHub miners [103, 127]:

Table 4: 14 Commit level features that Commit Guru tool [44, 94] mines from GitHub repositories

Dimension	Feature	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	ENTROPY	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether the change is defect Changes that fixing the defect are more likely to introduce more defects fixing ?
History	NDEV	#dvelopers changing modified files
	AGE	Mean time from last to the current change
	NUC	#changes to modified files before
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem



(a) Popular projects. Median= 5 releases, 4 months, and 41 defective commits.

(b) Unpopular projects. Median= 5 releases, 7 months and 29 defective commits.

Fig. 3: Distributions seen in the first 150 commits.

Nature	Type	Method	Pre-processing	# Features (columns)	# of commits (rows)
late-data	*Cross*	Bellwether [53]	CFS, SMOTE and steps illustrated in §5.1	Selected by CFS.	All commits from the identified cross project.
		TCA+ [74]	SMOTE	5 components with linear kernel (data supplied with all features)	Pick first 150 and last 150 commits from the bellwether project.
early-data	*Cross*	$*E_{size}(Bellwether)$	Steps illustrated in §5.1 in [44, 49, 72]	LA, LT = lines added, lines of code in file before change	Sample equal number of defective and clean commits as available in the first 150 commits (not exceeding 25 each).
		$*E_{size}(TCA+)$	Steps illustrated in §5.1	2 components with linear kernel (data supplied only with LA and LT)	
early-data	*Within*	E [103]	CFS and steps illustrated in §5.1	Selected by CFS	(same as above)
		$*E_{size}$	Steps illustrated in §5.1	LA and LT	
late-data	*Within*	ALL	CFS, SMOTE and steps illustrated in §5.1	Selected by CFS.	All the commits before the release under test.

KEY: All early-data sampling methods are shaded ( ) and policies proposed in this study are indicated by *.

Table 5: This table lists three baselines approaches (where two of them are *Cross* and the remaining one is a *Within*) along with three early-data variants to support this study. Note many of these methods used the steps advised in [44, 49, 72]. Those steps are discussed in §5.1. Also, for a discussion on CFS, see §5.2.

– Less than 1% defective commits;
– Less than two releases;
– Less than one year of activity;
– No license information;
– Less than five defective and five clean commits.

Our sampling was done twice:

– Once for *popular* project; i.e., those with more than 1000 "star" rankings in GitHub;
– Once for *unpopular* projects; i.e., those with less than 1000 "star" rankings in GitHub.

This sampling yield 155 popular projects and 85 unpopular projects. 1000 stars were used as the cut-off since that what was used in Munaiah et al. [70]. Also, looking at Figure 2, we see that above and below 1000 stars, the distributions are very different. Specifically, above and below 1000 stars, the median number of stars is 9,149 and 47 (respectively).

The projects collected in this way were developed in widely-used programming languages (including Java, Python, C, C++, C#, Kotlin, JavaScript, Ruby, PHP, Fortran, Go, Shell, etc.) for various domains.

All 14 features used in this study are listed in Table 4. Those features are extracted from these projects using *Commit Guru* [94]. The use of these particular features has been prevalent and endorsed by prior studies [44, 89]. *Commit Guru* publicly available tool used in numerous works [49, 125] based on a 2015 ESEC/FSE paper. Those 14 features became the independent attributes used in this empirical study.

In light of results by Nagappan and Ball, we created relative churn and standardized LA (lines of code added) and LD (lines of code deleted) features by separating by LT (lines of code in a file before the change) and LT and NUC (number of unique changes to the modified files before) dividing by NF (number of modified files) [48, 72]. Likewise, we dropped ND (number of modified directories) and REXP (recent developer experience) since Kamei et al. revealed that NF and ND are correlated with REXP and EXP (developer experience). Lastly, we applied the logarithmic transformation to the remaining features (except for the boolean variable 'FIX') to handle skewness [102].

Similar to the back-trace approach by SZZ algorithm [106] each commit was labeled "defective" (based on certain defect-related keywords) or "clean" (otherwise) internally by *Commit Guru*. *Commit-guru* walks back in the code to find the changes associated with that commit.

This empirical study uses three sets of algorithms:

- The nine classification algorithms described in §5.1.1;
- Pre-processing algorithms (for some sampling policies) described in §5.2;
- The seven sampling methods are described in §5.2.1.

But, *CommitGuru* does not provide project release information. Therefore we cloned each GitHub project to our local machine and extracted GitHub releases/tags information by executing the following command shown below:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d"
```

5.1.1 Classifiers/Optimizers

We use all the ten classifiers and optimizers used in the early *Within* study [103]. The ten classifiers are prevalent in the SE literature chosen about Ghotra et al. [27] work that extensively compared 30+ defect prediction algorithms in four ranks.

Those classifiers/optimizers were:

- Logistic Regression (LR);
- Nearest neighbour (KNN) (minimum 5 neighbors);
- Decision Tree (DT);
- Random Forrest (RF)
- Naïve Bayes (NB);
- Support Vector Machines (SVM)
- TLEL
- DODGE
- Hyperopt

INPUT:

- A dataset
- $\mathcal{E} \in \{0.05, 0.1, 0.2\}$
- A goal predicate p; e.g., P_{opt} or $d2h$;
- Objective, either to maximize or minimize p.

OUTPUT:

- Optimal choices of preprocessor and learner with corresponding parameter settings.

PROCEDURE:

- Separate the data into train and test
- Choose set of preprocessors, data miners with different parameter settings from Table 6.
- Build a tree of options for preprocessing and learning. Initialize all nodes with a weight of 0.
- Sample at random from the tree to create random combinations of preprocessors and learners.
- Evaluate N_1 (in our case $N_1 = 12$) random samples on training set and reweigh the choices as follows:
  - Deprecate ($w = w - 1$) those options that result in the similar region of the performance score α ($\alpha \pm \mathcal{E}$)
  - Otherwise endorse those choices ($w = w + 1$)
- Now, for N_2 ($N_2 \in \{30, 100, 1000\}$) evaluations
  - Pick the learner and preprocessor choices with the highest weight and mutate its parameter settings. Mutation is done, using some basic rules, for numeric ranges of attribute (look for a random value between $(best, (best + worst)/2)$ seen so far in $N_1 + N_2$). For categorical values, we look for the highest weight.
- For $N_1 + N_2$ evaluations, track optimal settings (those that lead to best results on training data).
- Return the optimal setting and apply these to test data.

Fig. 4: Pseudo-code of DODGE replicated from [1]

– ManualUp/Down

DODGE: Agrawal et al.'s DODGE is a state-of-the-art hyper-parameter optimization method extensively assessed for defect prediction [1]. One critical problem in hyper-parameter optimization (such as grid-search or brute force) is the run-time overhead. DODGE overcomes this by terminating much faster by skipping redundant options.

DODGE is an ensemble tree of classifiers and pre-processors as shown in Table 6. DODGE is broadly a two-step process as shown in Figure 4. First, DODGE iteratively shrinks (prunes the tree) the tuning search space by ignoring redundant options sampled from the tree. Then in the next set of iterations, it finds near-optimal options by looking between the best and worst options seen so far.

DODGE is shown to perform much better than building models with classifiers or pre-processors directly with off-the-shelf default options [1]. Notably, Agrawal et al. have highlighted that DODGE fails on complex data sets (having intrinsic dimensional $\mu > 8$). We found the intrinsic dimensionality of 240 projects explored in this study to be $\mu < 2$. Moreover, this being a defect prediction study, we explore DODGE in this study.

Hyperopt: A prevalent hyper-parameter optimizer proposed by Bergstra et al. in [7]. Agrawal et al. in a recent defect prediction work [1] showed DODGE to outperform hyperopt for software defect prediction. Yet, we include Hyperopt because its a state of the art optimizer in the machine learning community that may augument early methods and works differently when compared to DODGE. For our experiments we used Tree-structured Parzen Estimator (TPE) wrapped in the Hyperopt toolkit publicly available here [8].

Table 6: Hyperparameter tuning options explored in this paper. Note that we make no claim that this is a complete list of options. Rather, we merely claim that a reader of the recent SE literature on hyperparameter optimization might be tempted to try some subset of the following.

DATA PRE-PROCESSING
Software defect prediction:

- **Transformations**
  - StandardScaler
  - MinMaxScaler
  - MaxAbsScaler
  - RobustScaler(quantile_range=(a, b))
    - a,b= randint(0,50), randint(51,100)
  - KernelCenterer
  - QuantileTransformer(n_quantiles=a, output_distribution=c, subsample=b)
    - a, b = randint(100, 1000), randint(1000, 1e5)
    - c=randchoice([‘normal’,‘uniform’])
  - Normalizer(norm=a)
    - a = randchoice([‘l1’, ‘l2’,‘max’])
  - Binarizer(threshold=a)
    - a= randuniform(0,100)

LEARNERS
Defect prediction and text mining:

- DecisionTreeClassifier(criterion=b, splitter=c, min_samples_split=a)
  - a= randuniform(0.0,1.0)
  - b, c= randchoice([‘gini’,‘entropy’]), randchoice([‘best’,‘random’])
- RandomForestClassifier(n_estimators=a,criterion=b, min_samples_split=c)
  - a,b = randint(50, 150), randchoice([’gini’, ’entropy’])
  - c = randuniform(0.0, 1.0)
- LogisticRegression(penalty=a, tol=b, C=float(c))
  - a=randchoice([‘l1’,‘l2’])
  - b,c = randuniform(0.0,0.1), randint(1,500)
- MultinomialNB(alpha=a)
  - a= randuniform(0.0,0.1)
- KNeighborsClassifier(n_neighbors=a, weights=b, p=d, metric=c)
  - a, b = randint(2, 25), randchoice([‘uniform’, ‘distance’])
  - c = randchoice([‘minkowski’,‘chebyshev’])
  - if c==’minkowski’: d= randint(1,15) else: d=2

Hyperopt is stochastic method that produces random hyper-parameter settings to TPE. The TPE groups the evaluations of various hyper-parameter settings to best and rest. Essentially, TPE reflects over the history of evaluations seen up to date to jump to the next best setting to explore.

The TPE explores the best hyper-parameter setting to from the history of evaluations seen to date by order. TPE selects the best hyper-parameter options is then modelled as a Gaussian with its own mean and standard deviation. The groups are modelled as a Gaussian computed with its own mean and standard deviation.

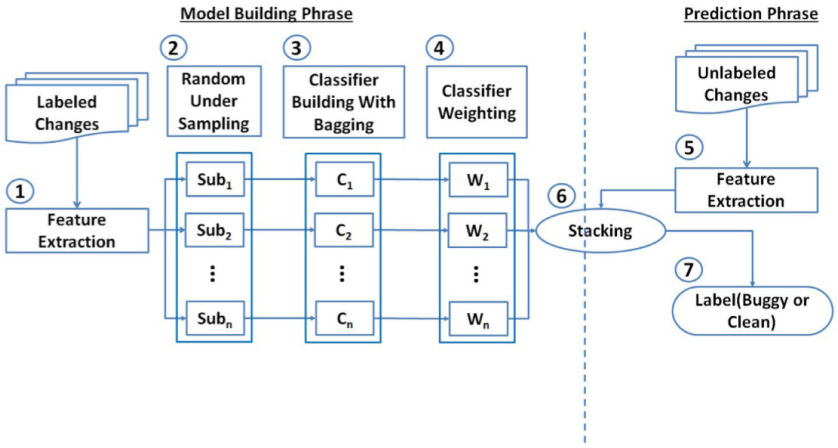


Fig. 5: Two-Layer Ensemble Learning framework replicated from the work by Yang et al. [129]

Two-layer ensemble learning (TLEL): Yang et al. in 2017 proposed an ensemble approach that leverages decision tree with bagging similar to a Random Forest model [129] depicted in Figure 5. A difference here is that Random Forest uses many decision trees, but in TLEL, the training data is under-sampled randomly to train many yet different Random Forest models. A commit is classified by TLEL as defective when most of the Random Forest models trained with different samples of data being stacked agree (second layer ensemble).

Manual Up/Down: In the past, Menzies et al. showed trivial approaches that use no training information like Manual Up/Down outperformed in identifying defects compared to complex methods [66]. Recently (2018) by Zhou et al. reported that simple size based models show a promising predictive performance, therefore advised researchers to include Manual Up/Down methods as a baseline while proposing any new technique [140].

Koru et al. related a module’s defect proneness with their size. In one case with two commercial projects they found smaller modules were defective [50] whereas in another they found larger classes were more defect prone [51]. There we include both their methods to classify our commits:

- **ManualDown** We classify all test commits as defective if its size (’la’ in Table 4) is less than the median size among the test commits [50]. The philosophy here is more minor changes should be inspected first and therefore penalized.
- **ManualUp** We classify all test commits as defective if its size (’la’ in Table 4) is greater than or equal to the median size among the test commits [51]. The philosophy here is more enormous changes should be inspected first and therefore penalized.

To reiterate, these two methods do not require any training commits.

5.1.2 Pre-processers

There are many ways to sample data in order to build a model, and no paper can cover them all. Based on our reading of the papers in Table 2, some frequency counts of different

methods published in [103] and using own engineering judgement, the following methods are representative of much of the prior research in this field:

- The steps advised by [48, 72, 102] (see §5.1);
- Correlation-based Feature Selection (CFS, see §5.2);
- Synthetic Minority Over-Sampling (SMOTE, see §5.2).

Not all combinations of pre-processing steps are valid. Table 5 show the types of pre-processing applied for each sampling policy used in this paper.

(Technical aside: just to document that we avoid a common threat to validity, we state there that we applied SMOTE only to the training data but *not* to the test data [3]).

5.2 Algorithms

This next section offers more details on our pre-processors and other algorithms.

Correlation-based Feature Selection (CFS): CFS is a prevalent feature selection technique proposed by Hall [29]. That is often used in building supervised defect prediction models, especially while building defect predictors [48]. A heuristic-based technique to incrementally assess a subset of features. Internally, a best-first search is applied to identify influential sets of features but are not correlated with each other. Nevertheless, it should be correlated with the target (classification). Below we show how each subset is computed:

$merits = krcf/\sqrt{k + k(k-1)r_{ff}}$ where:

- $merits$ is the worth of some subset s containing k features;
- rcf is a score that clarifies the association of that set of features to the class;
- rff is the component to include mean score and association between the things in s, where rcf ought to be huge and r_{ff}.

Synthetic Minority Over-Sampling (SMOTE): A sampling technique proposed by Chawla et al. [16] to handle the class imbalance problem. Because when there are an unequal number of clean and defective commits (or modules, records, etc.) in the training set, classifiers can scuffle to discover the target class. SMOTE is recommended by many researchers in the space of defect prediction [3, 113].

SMOTE uses K-nearest neighbors (minimum five commits required) to extrapolate and synthesize artificial data points (commits) to balance the training set. As mentioned earlier, SMOTE is not needed for every sampling policy listed in Table 5. Early 'E' based policies are balanced with an equal number of defective and clean commits by definition.

5.2.1 Sampling Methods

Figure 6 shows a set of options from which we can generate a large number of sampling options. For example, training data come from *within* or be drawn *cross* from another project. Also, in the bellwether method, we explore and prune most of the training data (only using the project that yields a model that outperforms all the other project models).

To find representative sampling techniques, we checked the papers listed in Table 2 for standard sampling policies, *Cross* techniques, projects, and features. We found numerous *Cross* techniques reported in the past decades; for example, Sousuke's work recently (2020) explored 24 *Cross* approaches [5]. In the context of *Within*, Shrikanth et al. identified many sampling strategies (based on recent releases, recent months of data, and all past data) to build

predictors [103]. However, in the context of *Cross*, we found only two ('whole' or 'part'). Nevertheless, both 'whole' or 'part' are late-data. The focus of this study is to check the efficacy of using early-data approaches in prevalent *Cross* approaches and not rank numerous *Cross* techniques. We chose two representative *Cross* techniques, specifically $TCA+$ and $Bellwether$, because:

- $TCA+$ is an active *Cross* technique in this space based on the seminal work by Nam et al. [74].
- $Bellwether$, a recent (2018) baseline approach by Krishna et al. that performed better than $TCA+$ [52]

We test these two and the following techniques on over 12,000+ releases across multiple classification algorithms. Let us look into each of these late-data techniques below.

Bellwether Project ($Bellwether$): All the commits from a *bellwether* project are used as the training data [52]. The bellwether project is identified by comparing the predictive performance scores of every project with every other project within the population. The bellwether transfer learning method that assumes "*When a community works on software, then there exists one exemplary project, called the bellwether, which can define predictors for the others.* [52]". According to Krishna et al. [52]" that finding the bellwether requires an $O(N^2)$ study– an approach that can be difficult to scale to 100s or 1000s of projects.

Transfer Component Analysis (TCA+): Pan et al. proposed a domain adaptation technique that enables the transition of information between source and target domains [79]. Extending that, Nam et al. stacked many normalization rules on top of TCA to form TCA+ [76]. A specific normalization rule is applied based on the similarity of the data set characteristics between the source and the target project. Rahul and Menzies [52] report TCA+ to perform better than other transfer methods, namely Transfer Naive Bayes [60] and Value Cognitive Boosting Learner [96].

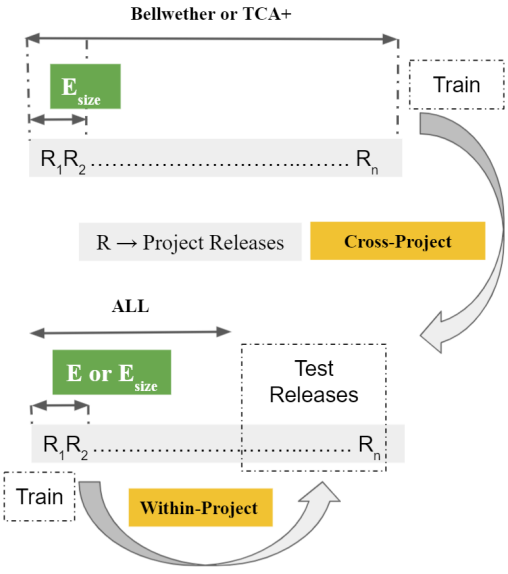


Fig. 6: A visual map to gauge the sampling policies listed in Table 5.

An approach to transfer learn using TCA+ is as follows:

- Choose a bellwether project randomly from all the bellwether projects.
- Figure 10 shows for TCA+ that as the number of training commits increases, the runtime cost of TCA+ increases drastically. It is practically not feasible to test in all 12,000+ releases repeated for all ten classifiers. In the case of $Bellwether$ based policies, we can at least cache the predictor as it uses a fixed set of all the commits. Caching is impossible with TCA+ as the approach decides transformation rules based on both train and test commits.
- To manage the runtime complexity without having to hide the data-hungriness of TCA+, we do the following:
  - We built defect predictions with varying sizes in our prior work [103] and found that predictors needed a minimum of 150 training commits. As mentioned above unlike other methods $TCA+$ is CPU intensive, therefore we could only increase training commits up-to 300 (2x 150). This is not a sampling threat because later in §6 we find the early-data-lite $TCA+$ variant performs better than $TCA+$ trained with 300 commits supporting our early conjecture.
  - But picking the first 300 commits would ignore the recent project data and may not represent the project life cycle. Thus of those 300 commits, we sample the earliest 150 commits and the latest (recent) 150 commits of the project data. Nevertheless, note in the context of *Within* Shrikanth et al. showed that using all the project commits is needless for predicting defects [103].

For TCA+ related policies listed in Table 5 we reused the implementation from their replication package [4] by Kondo et al., which is an EMSE'19 article about feature reduction techniques on defect prediction models [48].

Next, we will discuss the two early-data variants of the above two techniques.

Early Bellwether ($E_{size}(Bellwether)$): Instead of using all the commits in the bellwether project, a small sample of training data is curated. Specifically by randomly sampling 25 defective and 25 clean changes from the first (earliest) 150 commit (a method proposed by Shrikanth et al. [103]). All the data features are removed except two size-based features, 'LA' and 'LT.' The rationale for choosing two features are discussed later in §5.5.

Early TCA+ ($E_{size}(TCA+)$): This proposed technique is similar to TCA+ except for the 'sampling method'. Instead of using all the commits in the selected project, a small sample of training data is curated. Specifically by randomly sampling 25 defective and 25 clean changes from the first (earliest) 150 commit (a method proposed by Shrikanth et al. [103]). All the data features are removed except two size-based features, 'LA' and 'LT.' The rationale for choosing two features are discussed later in §5.5.

Then, to compare *Cross* with *Within* techniques, we chose E that is endorsed by Shrikanth et al. in [103] and shown to outperform predictors that use all past commits or recent commits. To check of E can be sufficed with just two features as we did for $E_{size}(Bellwether)$, we created E_{size}.

Early Sampling (E): The training data is curated randomly sampling 25 defective and 25 clean changes from the first 150 changes made within the project under study [103].

Early Sampling with few features (E_{size}): Training data is curated same as E but it is restricted to two size based features 'LA' and 'LT' listed in Table 4. The rationale for this is discussed later in §6.

[4] https://sailhome.cs.queensu.ca/replication/featred-vs-featsel-defectpred/

Sampling methods that are not super-scripted by $+$ imply that features are selected while building predictors as required using the correlation-based Feature Selection (CFS) feature selection discussed in §5.2.

Lastly, we note here that this work is 'not' to rank different *Cross* techniques like $TCA+$ or $Bellwether$ but to 'check' if such prevalent techniques can suffice (or do better) with early-data approaches.

5.3 Evaluation Criteria

We use all the seven criteria used in the baseline study to gauge the predictors built using various sampling policies. That study consults from widely-used measures [6, 24, 44, 49, 62, 67, 111, 113, 118, 128, 132, 134, 135] in the defect prediction literature.

Note for the following seven predictive performance measures:

- Except for Initial number of False Alarms, all six other measures range from 0 to 1;
- *D2H, IFA, Brier, PF* of these criteria need to be minimized, i.e., for these criteria *less* is *better*.
- For Three of these *AUC, Recall, G-Measure* criteria need to be maximized, i.e., for these criteria *more* is *better*.

Prior work has shown that precision has significant issues for unbalanced data. We do not include that in our evaluation [67]. Prior reviewers of this paper have noted that this might mean we miss certain effects relating to that section of the evaluation space. To address that point, we have added in an evaluation metric that draws from multile "corners" of the evaluation space (see the MCC measure of §5.3.8)

Before listing these criteria, we note that in practice, at least for the data explored here, many of them turned out to be uninformative. For example, IFA turned out to has statistically indistinguishable results across all our treatments. Similarly, all our better methods will achieve similar G-Measures (so that measure will not be so critical to determining what treatment is "best").

5.3.1 Brier

Brier is the absolute predictive accuracy measure. Numerous defect prediction papers [49, 62, 111, 113] endorse this measure. Let C be the total number of the test commits. Let y_i be 1 (for defective commits) or 0 otherwise. Let $\hat{y_i}$ be the probability of commit being defective (computed from the loss function in scikit-learn library [85]).

Then:

$$Brier = \frac{1}{C}\sum_{t=1}^{C}(y_i - \hat{y_i})^2 \tag{1}$$

5.3.2 Initial number of False Alarms (IFA)

Based on the observation by Parnin and Orso [82] that developers lose their trust in such analytics if they encounter many initial false alarms. Thus by simply counting the number of false alarms encountered after sorting the commits in the order of probability of being defect-prone. IFA is simply the number of false alarms before finding the first actual alarm.

5.3.3 Recall

Recall is the number of inspected defective commits divided by all the defective commits.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{2}$$

5.3.4 False Positive Rate (PF)

PF is the ratio between the number of clean commits predicted as defective to all the defective commits (irrespective of classification).

$$PF = \frac{False\ Positives}{False\ Positives + True\ Negatives} \tag{3}$$

5.3.5 Area Under the Receiver Operating Characteristic curve (AUC)

It is simply the area under the curve between the false-positive rate and true positive rate.

5.3.6 Distance to Heaven (D2H)

D2H or "distance to heaven" is computed as an aggregation on two metrics Recall and False Positive Rate (PF). Where "heaven" is a place with $Recall = 1$ & $PF = 0$ [18].

$$D2H = \frac{\sqrt{(1 - Recall)^2 + (0 - PF)^2}}{\sqrt{2}} \tag{4}$$

5.3.7 G-measure (GM)

GM is computed as a harmonic mean between the compliment of PF and Recall. It is measured as shown below:

$$G - Measure = \frac{2 * Recall * (1 - PF)}{Recall + (1 - PF)} \tag{5}$$

GM and D2H essentially combine the same two measures, $Recall$ and PF. Nevertheless, we still employ those as they have been used endorsed separately in the literature. Notably, as seen from results in [103] and in this work shown later in §6, it is not necessary that achieving good results on GM would also associate with good D2H (or vice-versa).

Due to the nature of the classification process, some criteria will always offer contradictory results:

- A classifier may simply achieve 100% $Recall$ just by labeling all the test commits as defective. But as a side-effect, that method will incur a high PF.
- Secondly, a classifier may classify all test commits as clean to show 0% PF, but that method will incur a very low $Recall$.
- Lastly, Brier and Recall are also antithetical since reducing the loss function implies missing some conclusions lowering $Recall$.

5.3.8 Mathew's Correlation Coefficient (MCC)

MCC utilizes all the four computations namely True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) of the confusion matrix, such that:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6}$$

Thus, many researchers endorse the use of MCC, especially in the space of software defect prediction [49, 131]. It returns a score between -1 and $+1$, where $+1$ indicates higher predictive performance, -1 contrary predictions and 0 indicates most of the predictions poor predictive performance (random).

5.4 Statistical Tests

Predictors built using each of the sampling policies listed in Table 5 are tested on all appropriate (future) project releases. Then we compute each of the evaluation criteria discussed above in §5.3. Then each of the scores is grouped and exported by a sampling policy and classifier pair.

To reiterate, each population is a collection of a specific evaluation score. Moreover, populations can have the same median but have an entirely different distribution. Thus to rank each of those populations of evaluation scores, we use the Scott-Knott test recommended by Mittas et al. in TSE'13 paper [69]. This procedure is a top-down bi-clustering method used to rank different predictors created using different sampling policies and classifiers under a specific evaluation measure. This method sorts a list of l those evaluation scores ls by their median score. Next, it then splits l into sub-lists *m, n*. This is done to maximize the expected value of differences in the observed performances before and after divisions.

For lists l, m, n of size ls, ms, ns where $l = m \cup n$, the "best" division maximizes $E(\Delta)$; i.e. the difference in the expected mean value before and after the spit:

$$E(\Delta) = \frac{ms}{ls} abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} abs(n.\mu - l.\mu)^2$$

Additionally, a conjunction of bootstrapping and A12 effect size test by Vargha and Delaney [116] is applied to avoid "small effects" with statistically significant results.

Important note: we apply the Scott-Knott test independently to all the seven evaluation criteria; i.e., when we compute ranks, we do so for (say) $Recall$ separately to PF.

5.5 Other Details

We wish to report not just performance scores but also offer some details about the model that generates those scores. To that end, we will exploit the bellwether effect. Specifically, we will (a) find the most important features then (b) using them in bellwethers that represent most of our data.

6 Results

An overview of our experiment is shown in Figure 6. We build predictors using the sampling strategies listed in Table 5 and classifiers elucidated in §5.1.1. Then we test all project releases

using those predictors. We do this similar to our prior work [103] as this is a realistic way to compare all sampling policies and methods on a similar scale.

6.1 **RQ1: Can we build early defect prediction models from unpopular projects?**

6.1.1 Motivation

Our prior results that endorsed early methods were scoped to only popular GitHub projects [103]. Choosing only popular projects is a sampling decision often made by many SE researchers in empirical studies to mitigate generalizability threat [71]. As mentioned earlier in §5.1, unpopular projects could be non-trivial projects (like homework assignments), and that could affect the conclusion.

On the contrary, a strong argument could be that perhaps popular projects may not realistically represent software engineering in the real world. In other words, not all projects are fully staffed with a sufficient budget. Therefore it is necessary and worthy to check the value of early methods on the unpopular sample of projects. §5.1 and §7 discuss the selection of unpopular projects we sampled from GitHub is non-trivial.

6.1.2 Approach

To check whether the proposed early method work on unpopular projects we compare defect predictors built by sampling training commits using E with those that sampled all past data ie., ALL. We do not consider other stratification like release, or 3 months since ALL subsumes more data. Notably, ALL is a prevalent (50%) sampling methods in software defect prediction [103]. Therefore to assess defect predictors on each project release this we construct the experiment as follows:

- First we sample training data within the project. The sampling will depend upon either E or ALL.
- The sampled training data is pre-processed and appropriate features are selected as listed in Table 4.
- Classifiers listed in §5.1.1 are instantiated with copies of pre-processed training data.
- All the instantiated predictors are tested on all 85 unpopular project releases and their predictive performance is gauged using 7 measures elucidated in §5.3.
- Lastly, their scores (eg., population of recall scores) per classifier+policy (pair) is ranked using the Scott-Knott test elucidated in §5.4.
-

Note:

- To avoid a methodological error, we do not test/consider project releases before the first 150 commits.
- Therefore, the population of evaluation measures are tested on equal number of unpopular project releases.

6.1.3 Results

Table 7 shows the result of the predictors tested in all applicable project releases of 85 unpopular projects. Policy E uses fewer early data (row #1) performs just as same as predictors that were built sampled with ALL (all past commits) in row #3.

Table 7: '11' *within-project* defect prediction models tested 'only' in 85 *unpopular* project's releases. In the first row, "+" and "-" denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote "early-data" sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §5.4). The 'wins' column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing '( )'.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
E	SVM		71 (50)	33 (28)	65 (20)	39 (24)	34 (19)	64 (32)	1 (4)	25 (32)
E	LR	6	77 (44)	43 (30)	65 (18)	40 (22)	38 (20)	67 (27)	1 (4)	25 (30)
ALL	SVM		50 (55)	15 (18)	66 (24)	38 (32)	23 (15)	55 (53)	1 (5)	30 (45)
E	RF		67 (55)	37 (26)	63 (20)	41 (22)	37 (18)	61 (33)	2 (5)	22 (33)
ALL	RF	5	50 (53)	16 (22)	64 (24)	42 (30)	25 (16)	53 (48)	1 (4)	28 (44)
ALL	KNN		50 (46)	20 (18)	64 (22)	40 (26)	26 (14)	54 (42)	1 (4)	26 (40)
E	KNN	4	71 (50)	44 (30)	62 (19)	44 (19)	41 (19)	62 (28)	2 (5)	20 (31)
ALL	NB	2	14 (56)	9 (26)	50 (13)	64 (24)	25 (21)	13 (52)	4 (13)	0 (31)
E	DT		57 (56)	42 (35)	56 (20)	48 (24)	42 (22)	53 (37)	2 (5)	11 (34)
E	NB	1	50 (63)	28 (36)	57 (18)	47 (30)	35 (23)	51 (51)	2 (6)	16 (33)
ALL	DT		50 (43)	30 (23)	57 (18)	45 (24)	34 (18)	52 (36)	2 (5)	15 (33)

Early methods are 'not' scoped to only popular projects.

6.2 **RQ2: Can we build early defect prediction models with fewer features?**

6.2.1 Motivation

The prior work showed that it is possible to build operable defect predictors using fewer early project data. One reason in the preceding work was that much of the knowledge required (defects) to build defect predictors were reported in a few months of the project. In this work, we explore that region again to understand which of those 14 features listed in Table 4 are essential to the early predictor E. And if the answer to that is a few, can we restrict the predictors to only learning from a fixed set of features while producing operable performance? It would be easier to explain with fewer features as to why a specific commit was classified as defective by the predictor.

6.2.2 Approach

To answer this RQ we perform three experiments as follows:

- In the first experiment we report the frequency of features chosen by CFS while building defect predictors which were sampled using E.
- In the second experiment we create a new sampling policy (say $E_?$) that only uses most frequently chosen feature identified in the above step.
- We create two predictors that sampled training commits (pre-processed as listed in Table 5) using E and $E_?$ and test on all project releases.

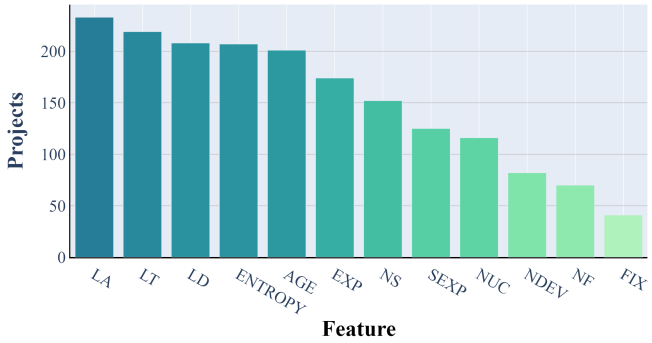


Fig. 7: Frequency of features chosen by the CFS feature selector (using just the first 150 commits) sampled using E in all the 240 projects

– In the third experiment, we also test all project releases using the ManualUp andManual-Down approach (elucidated in §5.1.1) that uses no training information. This is because researchers have advised to include such trivial approaches that does not use any training information [140].
– Lastly, we compare the predictive performance of defect predictors that were built using E, $E_?$ and $ManualDown$ and $ManualUp$ on all seven evaluation measures listed in §5.3 and rank them using the Scott-Knott test elucidated in §5.4.

6.2.3 Results

We make the following observations from Figure 7 and Table 8:

Finding Important Features: Shrikanth et al.'s early sampling rule 'E' shown to be adequate for *Within* [103]. Given that we are building models from such a sample, it is tempting to ask, "does that small sample produce a succinct model?". Note that this was indeed the case, then we could offer a clear report on what factors most influence defects.

Figure 7 shows the frequency of features selected by the CFS algorithm (see §5.2) across all our 240 projects (using just the first 150 commits). To select the "best" features from that space, we build models using the top $1 \leq x \leq 12$ ranked features, stopping with $x + 1$ features performed no better than x features (and here, we are testing all the releases that occurred after that first 150 commits). That procedure reported that models learned from two size-ranked features performed as well as anything else:

– LA: Lines of code added
– LT: Lines of code in a file before the change

Using the results from Figure 7 we built early defect predictor only using the top two features 'LA' and 'LT'. Therefore $E_?$ becomes E_{size}. The results from Table 8 clearly shows that E_{size} have more wins than predictors that sampled training commit using E. And by extension to prior result ALL or other stratification like recent release or based on months. While $ManualDown$ is seen at the top of the table due to higher recall, it also produced many false-alarms (see row #3 where PF is 48%).

Table 8: '14' *within-project* defect prediction models tested in all 240 project's releases. In the first row, "+" and "-" denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote "early-data" sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §5.4). The 'wins' column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing '( )'.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
E_{size}	LR		75 (40)	28 (25)	70 (15)	34 (17)	30 (17)	70 (25)	1 (3)	34 (26)
E_{size}	SVM	6	73 (44)	30 (27)	69 (15)	35 (19)	30 (18)	69 (26)	1 (3)	33 (27)
-	$ManualDown$		86 (25)	**48 (15)**	70 (12)	36 (11)	41 (17)	75 (14)	1 (4)	30 (22)
E_{size}	KNN	4	70 (41)	33 (26)	67 (17)	37 (19)	33 (18)	66 (26)	1 (3)	28 (28)
E_{size}	NB		75 (53)	38 (41)	63 (17)	44 (23)	36 (25)	61 (35)	1 (4)	24 (31)
E	LR		70 (42)	37 (33)	63 (16)	41 (22)	36 (20)	62 (30)	1 (4)	24 (28)
E	SVM		67 (42)	33 (31)	64 (17)	40 (20)	34 (19)	63 (30)	1 (4)	25 (28)
E	KNN		66 (43)	38 (30)	62 (18)	42 (20)	37 (20)	59 (28)	1 (4)	21 (31)
E_{size}	RF	2	64 (40)	31 (30)	64 (17)	40 (20)	33 (18)	61 (28)	1 (4)	25 (30)
E_{size}	DT		62 (39)	38 (30)	61 (18)	43 (19)	38 (20)	57 (28)	1 (4)	19 (32)
E	RF		60 (46)	32 (33)	61 (18)	43 (21)	35 (21)	56 (33)	1 (4)	20 (32)
E	DT		56 (46)	41 (37)	56 (17)	49 (22)	41 (24)	52 (33)	2 (5)	11 (30)
E	NB		50 (75)	27 (48)	54 (14)	55 (30)	36 (26)	38 (62)	2 (7)	8 (26)
-	$ManualUp$	0	14 (25)	52 (15)	30 (12)	73 (11)	59 (17)	16 (27)	7 (13)	-30 (22)

At-least for defect prediction, early data with two features 'la' and 'lt' is better than using recent (or more) data with many features.

6.3 **RQ3: Can we build early defect prediction models from transferring fewer data from other projects?**

6.3.1 Motivation

§5.2.1 lists both the situations and challenges of when one might need to transfer project data to build defect predictors. Although we cannot solve all *Cross-project* challenges in one article, we could pacify some of them by checking the efficacy of early methods in this context. If early methods work, then we can transfer relevant data from vast search space (many projects) in less time. Further, we can also reduce the need for data availability as we will only need a small portion of early data.

6.3.2 Approach

To check the efficacy of *Cross-project* methods against early methods we will build predictors using the following sampling strategies as below:

- *Cross-project* methods $Bellwether$ and $TCA+$
- We created early variants of *Cross-project* methods $Bellwether$ and $TCA+$ namely $E_{size}(Bellwether)$ and $E_{size}(TCA+)$. The details of these sampling strategies are elucidated in Table 5 and portrayed visually in Figure 6.
- We also include the within-project E_{size} to compare *Cross* and *Within*.

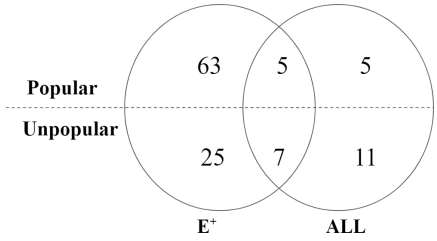


Fig. 8: Number of "satisfactory" bellwethers identified independently by two different sampling policies in all 240 projects.

– Note: We do not include ALL or E because E_{size} outperformed those sampling strategies in Table 8 and in our prior result [103].

The *Cross-projects* methods require a representative project among the 240 projects to transfer data from.

Finding Representative Cross-Project: Using just LA and LT for each of our projects, we found a model and measured its performance on the other projects. If that median performance was more the 70% 'Recall' and less than 30% 'PF', then the model was deemed "satisfactory". In a result that endorses the use of early life cycle data, Figure 8 shows that *more* of the models found via the E_{size} method was "satisfactory" than those found using all the data. That effect is particularly marked in the popular projects[5].

Lastly like in previous RQ's we compare the predictive performance of the various defect predictors that were built using different sampling policies on all seven evaluation measures listed in §5.3 and rank them using the Scott-Knott test elucidated in §5.4.

Note: We avoid the methodological error of testing the representative *Cross-project* in this experiment.

6.3.3 Results

We make the following observations using Table 9, Figure 9 and Figure 10.

– Rows #1 and #2 top of the Table 9 confirm that it is better to build *Cross* based predictors using early-data-lite method E_{size} than using sampling strategies that use more data (like $TCA+$ or $Bellwether$).
– Figure 9 and Figure 10 shows that it is much faster to identify and build bellwethers from a large pool of projects using E_{size} than $Bellwether$. This is because E_{size} based predictors need not update (re-train) by accumulating newer project commits for every new project release.
– Row #6 of Table 9 is the position of our prior result [103] that builds predictors using *Within* E project commits. That proves to show that we can do better than local data if we find good bellwethers.

[5] Clearly, this analysis might be overly dependent on the "magic numbers" used to select "satisfactory" bellwethers, i.e., 70% 'Recall' and less than 30% 'PF.' But we have run our experiments using 12 ($i <= 12$) different bellwether projects with nearly equal frequency (see the seven unpopular + 5 popular bellwethers in the middle of Figure 8). The statistical analysis of the 12 $E_{size\,Bellwether^i}$ using §5.4 shows that (a) they tie with each other and (b) perform as well or better than the other transfer learning methods explored in this paper.

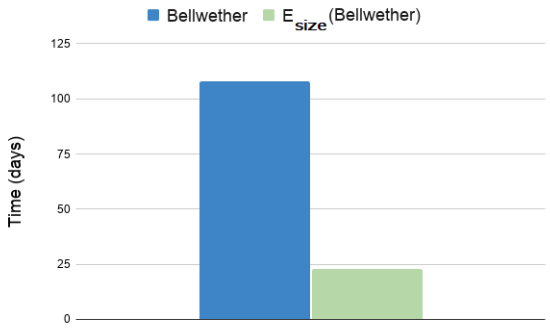


Fig. 9: Time taken to identify qualifying bellwether projects in all 240 projects using $Bellwether$ and $E_{size}(Bellwether)$ policies. Note that in our experiments, we ran on a multi-core cloud-based CPU farm. To collect the data here (which assumes we are running on a single-core machine), we summed the CPU time across all the cores in our experimental rig. We assert that this sum is valid since, in our experiments, there is nearly no communication between the cores (except that very end to accumulate the results).

Transfer Learning methods perform better (faster and accurate) when they are sampled using the early method.

6.4 **RQ4: Do complex methods supersede early defect prediction models ?**

6.4.1 Motivation

Our prior work and RQ’s 1, 2, and 3 have used classifiers with default parameter settings (off the shelf). However, numerous studies, especially in the space of software defect prediction, have shown considerable improvement in predictive performance when the classifiers are

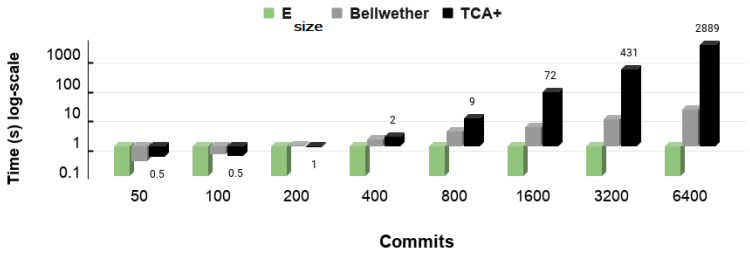


Fig. 10: Time taken by different sampling policies to train a model based on the number of commits.

Table 9: '30' *cross-project* and '1' *within-project* (denoted as $\Longrightarrow$) defect prediction models tested in all 240 project's releases. In the first row, "+" and "-" denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote "early-data" sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §5.4). The 'wins' column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing '( )'.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
E_{size} (Bellwether)	SVM		91 (26)	40 (34)	70 (16)	35 (21)	35 (23)	74 (23)	1 (4)	32 (26)
E_{size} (TCA+)	SVM	6	88 (33)	38 (28)	70 (16)	35 (19)	34 (21)	74 (24)	1 (4)	32 (26)
E_{size} (Bellwether)	LR		85 (30)	34 (27)	73 (15)	31 (17)	31 (18)	76 (20)	1 (4)	35 (25)
E_{size} (TCA+)	LR		82 (36)	32 (25)	72 (16)	33 (19)	31 (19)	74 (22)	1 (4)	34 (26)
E_{size} (TCA+)	KNN		79 (50)	32 (25)	70 (18)	35 (20)	32 (18)	72 (28)	1 (5)	30 (28)
$\Longrightarrow$ E_{size} (Within)	LR	5	78 (43)	25 (25)	73 (16)	31 (18)	27 (18)	73 (28)	1 (3)	37 (27)
E_{size} (Bellwether)	KNN		78 (43)	30 (26)	70 (17)	34 (19)	30 (19)	72 (27)	1 (3)	32 (27)
E_{size} (TCA+)	RF		75 (46)	31 (24)	69 (17)	35 (19)	32 (18)	71 (28)	1 (4)	30 (28)
E (TCA+)	LR	4	80 (40)	41 (19)	69 (16)	36 (15)	38 (17)	72 (24)	2 (5)	28 (27)
E_{size} (Bellwether)	RF		71 (41)	28 (26)	68 (18)	36 (21)	30 (19)	67 (28)	1 (4)	28 (28)
Bellwether	SVM	3	21 (44)	4 (10)	57 (17)	56 (30)	19 (15)	25 (49)	1 (7)	23 (40)
Bellwether	RF		19 (36)	3 (8)	56 (15)	56 (25)	19 (15)	23 (41)	1 (7)	22 (39)
E_{size} (Bellwether)	NB		88 (33)	49 (34)	64 (19)	42 (22)	42 (23)	67 (28)	2 (5)	23 (27)
E_{size} (TCA+)	NB		82 (38)	43 (30)	67 (18)	39 (21)	38 (22)	70 (26)	2 (5)	25 (28)
Bellwether	LR	2	76 (51)	33 (43)	67 (16)	39 (20)	32 (24)	66 (34)	1 (4)	31 (28)
E_{size} (TCA+)	DT		71 (39)	32 (25)	67 (19)	37 (19)	33 (18)	67 (27)	2 (4)	27 (28)
Bellwether	KNN		25 (41)	6 (10)	59 (17)	53 (27)	20 (16)	28 (44)	1 (5)	24 (39)
E (TCA+)	KNN		71 (53)	37 (28)	64 (20)	40 (21)	36 (20)	65 (33)	2 (5)	23 (28)
E (TCA+)	SVM		70 (55)	34 (31)	64 (20)	41 (24)	35 (21)	64 (40)	2 (5)	23 (32)
TCA+	LR		70 (39)	50 (22)	61 (22)	42 (21)	45 (22)	64 (28)	2 (6)	17 (35)
E (TCA+)	NB		67 (56)	39 (38)	60 (20)	46 (24)	38 (24)	60 (44)	2 (6)	17 (32)
E_{size} (Bellwether)	DT		67 (44)	30 (26)	65 (20)	39 (23)	33 (19)	63 (32)	1 (4)	25 (32)
E (TCA+)	DT		67 (42)	37 (25)	63 (19)	41 (20)	38 (18)	62 (30)	2 (5)	20 (30)
E (TCA+)	RF	1	67 (40)	34 (23)	65 (20)	38 (19)	35 (18)	65 (28)	2 (5)	24 (28)
Bellwether	NB		50 (60)	15 (21)	64 (22)	41 (32)	25 (16)	53 (55)	1 (5)	26 (38)
TCA+	SVM		50 (50)	23 (27)	61 (21)	45 (25)	31 (21)	52 (41)	2 (5)	20 (34)
TCA+	RF		50 (50)	31 (32)	57 (17)	48 (23)	36 (23)	49 (36)	2 (6)	13 (28)
TCA+	DT		50 (40)	34 (28)	56 (17)	49 (21)	39 (21)	48 (31)	2 (6)	11 (28)
TCA+	KNN		40 (42)	19 (22)	57 (18)	49 (24)	28 (21)	42 (36)	2 (5)	16 (32)
Bellwether	DT		33 (33)	13 (12)	59 (16)	48 (21)	24 (16)	38 (34)	1 (5)	19 (32)
TCA+	NB	0	23 (47)	19 (35)	50 (14)	61 (22)	34 (28)	23 (48)	3 (9)	0 (28)

tuned [1, 25] or using an ensemble approach [129]. But note the downside of tuning is the run-time overhead.

Therefore, it is essential to check if complex methods like tuning or ensemble favor sampling policies that use recent (or more) data than those confined to the early regions. It is also motivating to check if early methods can benefit from these complex methods. Perhaps the run-time overheads can be alleviated by working with fewer early data.

6.4.2 Approach

We build predictors using the state of the art methods, specifically $DODGE$, $Hyperopt$ and $TLEL$ elucidated in §5.1.1.

From RQ's 1 to 3 in either *Within* or *Cross*context E_{size} has performed similar or better than prevalent sampling strategies, indicating the importance of early regions of the software

project. Therefore like RQ#3 we create variants of $DODGE$, $Hyperopt$ and $TLEL$ using E_{size}. In other-words we will input fewer early data E_{size} and use those complex methods to check if they perform better or worse.

Lastly, we measure the predictive performance of the above approaches listed in §5.3 and rank the predictive performance of defect predictors that were built using the Scott-Knott test elucidated in §5.4.

6.4.3 Results

We make the following observations from Table 10:

- E_{size} (row #1) performs better than all complex methods that use either more data ALL or fewer early date E_{size}.
- Closest to E_{size} is the two-layer ensemble method that achieves almost similar results but with more data and more processing. But note $TLEL$ (row #2) was trained on every new project release accumulating more data, whereas E_{size} or any E based method is always trained just 'once'.

Table 10: 7 *within-project* defect prediction models tested in all 240 project's releases. In the first row, "+" and "-" denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote "early-data" methods (and all other rows employ optimizer or ensemble approach). Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §5.4). The 'wins' column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing '( )'.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
E_{size}	LR	7	73 (43)	28 (26)	70 (15)	34 (18)	30 (17)	69 (26)	1 (3)	35 (27)
ALL	TLEL	2	74 (50)	36 (56)	61 (20)	46 (31)	36 (31)	54 (46)	1 (4)	21 (38)
E_{size}	TLEL		67 (38)	33 (30)	64 (17)	39 (19)	34 (19)	62 (27)	1 (4)	26 (30)
E_{size}	HyperOpt		27 (60)	11 (31)	54 (14)	56 (28)	28 (22)	28 (55)	2 (11)	10 (28)
E_{size}	DODGE	1	100 (80)	68 (91)	50 (7)	71 (17)	49 (46)	0 (42)	3 (9)	0 (18)
ALL	DODGE		100 (80)	98 (69)	50 (0)	71 (3)	66 (36)	0 (17)	4 (9)	0 (0)
ALL	HyperOpt	0	50 (89)	40 (66)	50 (9)	63 (21)	44 (30)	27 (53)	3 (9)	0 (21)

Notably, Figure 11 confirms it is far more faster to build predictors sampled using E_{size} that neither needs SMOTE or CFS and only trained once throughout the project life-cycle.

Neither Optimizers nor ensemble methods surpass simpler early methods.

7 Threats to Validity

7.1 Sampling Bias

Generalizability of the conclusions will rely on the examples considered; i.e., what is essential here may not be genuine all over. Although the prevalent practice of such empirical studies is

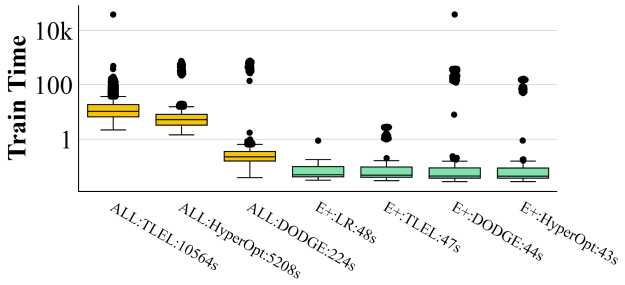


Fig. 11: Time taken by different sampling policies to train a model based on the number of commits.

to use popular OS GitHub projects, we broaden the scope by including unpopular projects. Nevertheless, all these projects are non-trivial engineering projects developed in numerous programming languages for various domains. Notably, lessons did not vary in either of these two populations of projects.

7.2 Construct Validity

We mined the project samples using *Commit-Guru*. *Commit-Guru* determines risky Commits using a method similar to SZZ [106]. Such an approach despite prevalent is widely debated for its false-positives. To minimize that threat *Commit-guru* only considers files that are source-code files and not documents like readme or pdf files. *Commit-Guru* classifies commit into different categories one of them is *merge-commit*. Studies have shown merge commit have little to no influence over making real changes to the software, therefore in this study we filter all merge commits.

Lastly, to further bold the validity of our early effect we cross-checked our results with the two projects 'QT' and 'OPENSTACK' used in the 2017 TSE article by McIntosh and Kamei [62] and found our early trend and effect to hold. Please see the early trend of those two projects in Figure 12 and the results in Table 11. Table 11 shows results of defect predictors that sampled training commits using recent six months (M6) endorsed in [62] .Vs. E_{size} endorsed in this study. Note our results are compared with ten classifiers and seven evaluation measures.

7.3 Learner bias

For identifying bellwethers by all-pairs experiment (each project is tested on all 12,000+ releases), we used only one classifier, 'Logistic Regression.' Perhaps other classifiers may qualify other projects are bellwethers. Using just one classifier may not be a threat for the following two reasons. Firstly a Logistic-Regression is recommended in the baseline study [103] and widely endorsed in classifier in defect prediction literature. Secondly, in all our results in §6 Logistic-Regression based predictors were in-par and better than five other classifiers explored in this study.

Fig. 12: A repeated pattern where most defects (shaded in red) occur early in the life cycle (before the green dotted line) is also observed among the two projects (QT and OPENSTACK systems) with 37,524 commits assessed recently by McIntosh and Kamei in [62].

Table 11: ‘12’ *within-project* defect prediction models tested ‘only’ in two projects ‘QT’ and ‘OPENSTACK’. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote “early-data” sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §5.4). The ‘wins’ column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing ’( )’.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
E_{size}	RF	6	71 (13)	35 (8)	67 (7)	33 (6)	35 (8)	69 (10)	5 (7)	19 (9)
E_{size}	NB	5	75 (14)	43 (7)	67 (9)	35 (7)	42 (8)	71 (9)	6 (8)	17 (10)
E_{size}	SVM	4	67 (18)	33 (9)	67 (7)	34 (8)	33 (8)	66 (14)	4 (7)	18 (11)
E_{size}	LR		64 (16)	27 (7)	67 (8)	33 (8)	28 (6)	65 (12)	4 (8)	21 (11)
$M6$	SVM	3	14 (25)	6 (5)	54 (10)	61 (17)	13 (7)	17 (28)	3 (8)	9 (16)
$M6$	RF		13 (19)	4 (5)	55 (8)	61 (13)	9 (10)	16 (22)	3 (8)	11 (17)
$M6$	KNN	2	20 (13)	8 (6)	56 (7)	56 (9)	13 (8)	24 (14)	3 (7)	11 (13)
E_{size}	DT	1	67 (18)	48 (6)	59 (7)	42 (6)	47 (5)	63 (11)	7 (9)	8 (10)
E_{size}	KNN		65 (17)	35 (18)	65 (7)	36 (7)	35 (16)	65 (13)	5 (9)	16 (10)
$M6$	DT		25 (21)	12 (12)	55 (7)	54 (12)	15 (14)	28 (22)	4 (11)	8 (11)
$M6$	NB		14 (22)	9 (12)	52 (8)	62 (12)	17 (11)	17 (21)	6 (13)	5 (13)
$M6$	LR	0	10 (28)	10 (13)	51 (6)	64 (17)	17 (12)	12 (28)	9 (52)	1 (10)

Further, an empirical study can only focus on a handful of representative classifiers. Thus we chose ten classifiers (Logistic Regression, Nearest neighbor, Decision Tree, Random Forrest, and Naïve Bayes). These ten classifiers cover a broad range of classification algorithms [27].

7.4 Evaluation bias

This paper uses both ‘Recall’ and ‘PF’ to identify bellwethers and seven evaluation measures (Recall, PF, IFA, Brier, GM, D2H, MCC and AUC) to compare the policies extensively. Other widely used measures in defect prediction are precision and f-measure. However, as mentioned earlier, those threshold dependant metrics have issues with unbalanced data [67].

7.5 Input Bias

All our proposed sampling policies randomly samples 50 commits from the first 150 commits of the project. Along these lines, it could be true that different executions could yield different results. However, this is not a threat, as our conclusions hold on a large sample size of 12,000+ releases.

8 Discussion

8.1 Management Implications

Much prior research has struggled to find stable conclusions that hold across multiple projects. Menzies et al. warn that many global lessons (generalizations) are not supported locally [64]. That is to say, the more data we see, the more exceptions and special cases might appear in the models learned from that data. Such conclusion instability in SE has been often recorded in [64, 104, 141].

Is that the best we can do? Ideally, SE research can offer stable general defect prediction principles (such as those seen above) to guide project management, software standards, education, tool development, and legislation about software. Such conclusion stability would have benefits for *trust, insight, training*, and *tool development*.

Trust: Conclusion instability is unsettling for project managers. Hassan [30] warns that managers lose trust in software analytics if its results keep changing. But if we can find stable conclusions (e.g., using early life cycle bellwethers), that would give project managers clear guidelines on many issues, including (a) when a certain module should be inspected; (b) when modules should be refactored; and (c) deciding where to focus on expensive testing procedures.

Insight: Sawyer et al. assert that insights are essential to catalyzing business initiative [99]. From Kim et al. [46] perspective, software analytics is a way to obtain fruitful insights that guide practitioners to accomplish software development goals, whereas for Tan et al. [110] such insights are a central goal. From a practitioner's perspective, Bird et al. [10] report, insights occur when users respond to software analytics models. Frequent model regeneration can exhaust users' ability for confident conclusions from new data. In this regard, we note that the early life cycle models found would be a stable source of insight for much of the project life cycle.

Tool development and Training: Previously [104] we warned that unstable models make it hard to onboard novice software engineers. Without knowing what factors most influence the local project, it is hard to design and build appropriate tools for quality assurance activities. Hence, here again, the stability of our early life cycle detectors is very useful.

All these problems with trust, insight, training, and tool development can be solved if, early on in the project, a defect prediction model can be learned that is effective for the rest of the life cycle (which is the main result of this paper). Within that data, we have found that models learned after just 150 commits, perform just as well as anything else. In terms of resolving conclusion instability, this is a very significant result since it means that for much of the life cycle, we can offer stable defect predictors.

One way to consider the impact of such early life cycle predictors is to use the data of Figure 13. That plot shows that software employees usually change projects every 52 months (either moving between companies or changing projects within an organization). According to Figure 13, in seven years (84 months), the majority of workers and managers would first

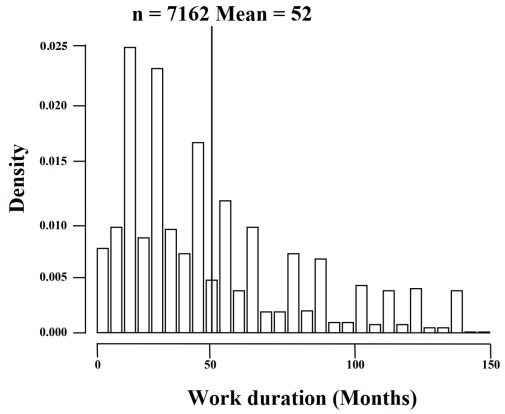


Fig. 13: Work duration histograms on particular projects; from [101]. Data from: Facebook, eBay, Apple, 3M, Intel and Motorola.

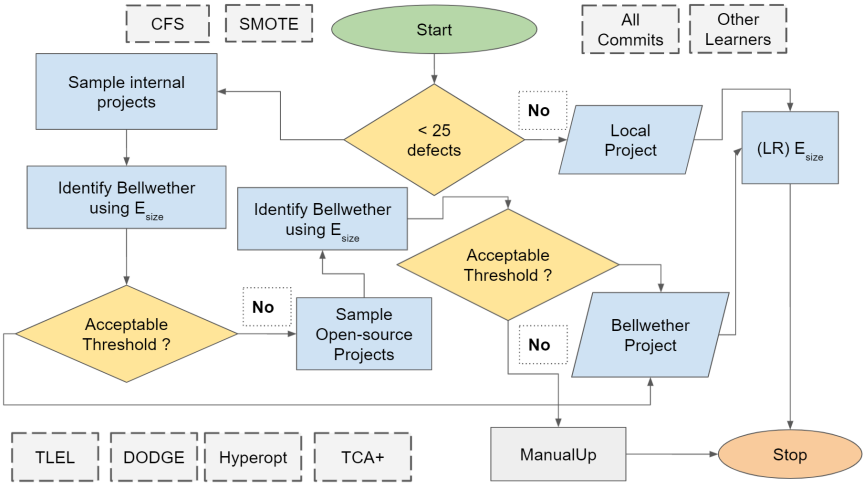


Fig. 14: A decision-tree to build defect predictors drawn using the results from §6 that find many complex methods (shown as disconnected blocks with dashed borders) to be needless.

appear on a job *after* the initial four months required to learn a defect predictor. Hence, for most workers and managers, the detectors learned via the methods of this paper would be the “established wisdom” and “the way we do things here” for their projects. This means that a detector learned in the first four months would be a suitable oracle to guide training and hiring; the development of code review practices; the automation of local “bad smell detectors”; as well as tool selection and development.

In summary, we created a simple decision tree in Figure 14 that would help practitioners to choose the appropriate method to build a defect predictor (endorsed as per our results) given a specific scenario. That figure endorses practitioners to build predictors using our

"early-data-lite" method E_{size} in both $Within$ and $Cross$ scenarios. However, in the worst scenario, when there is no access to data or the inability to find good bellwether projects, we suggest practitioners use $ManualUp$ until their local or neighboring projects report more defects (> 24). Note: Many complex methods and late-data explored in this study is unused in creating this decision-tree.

8.2 Systems Implications

Figure 9 shows the run-time required to find bellwether projects in all 240 projects using $Bellwether$ or $E_{size}(Bellwether)$ policies. Note that the early life cycle methods use 2.5 months less CPU than the alternative.

Since we only need the first 150 commits to identify a suitable project, many projects in their early stages may be included in the pool of transferable projects. Importantly Figure 8 show when sampled using E_{size} 30% "more projects" were qualified in much "less time". The choice of bellwether does not matter; perhaps, in practice, practitioners need not continue to look for bellwethers and terminate early as soon as they find the first bellwether. This could also mean lesser privacy concerns and enable practitioners to share fewer data freely. The early-data nature of our methods can also help to gauge sophisticated techniques like TCA+ faster.

Note: We do not claim $E_{size}(Bellwether)$ is data-lite, because it would still process all the projects in the search-space. But we do endorse E_{size} over current methods $Bellwether$ or $TCA+$. Because E_{size} is faster and accurate (see §6.3), as it uses fixed data (150 commits) and does not re-train for every new project release.

8.3 Research Implications

We have shown that when defect data contains information, that information may be densest is a small part of the historical record of a project. While we have *not* shown that other kinds of SE data have the same density effect, we would argue that now it is at least an open question that "have we been learning from the wrong parts of the data?".

Finally, we should now view it as a potential methodological error to reason across all data in a project. In the specific case of defect prediction we must now revisit any conclusion based just on later life cycle data There are many examples of such conclusions. For example:

- Hoang et al. says "We assume that older commits changes may have characteristics that no longer effects to the latest commits" [36].
- Also, it is common practice in defect prediction to perform "recent validation" where predictors are tested on the latest release after training from the prior one or two releases [25, 49, 62, 109].

More generally, before researchers focus on later life cycle data, they must first check that their (e.g.) buggy commit data occur at equal frequency across the life cycle. If the buggy commits data is isolated to a specific region, then predictors should be built from that region. This work finds that most buggy commits are found early in the project life-cycle.

9 Conclusion

This work expanded the validity of our prior work on "early sampling", which was scoped only in the context of within-project defect predictors and popular GitHub projects. In this

work, we checked the scope to show the efficacy of “early sampling” and highlight that we can simplify software analytics to a great extent. Because, when

“When data keep changing, the models we can learn from that data may also change. If conclusions become too fluid (i.e., change too often), then no one has a stable basis for making decisions or communicating insights.” [103].

Issues with conclusion instability, run-time complexity, and explanations of models improve if we can learn a predictive model early in the life cycle that is effective for the rest of the project. Specifically, in this work, we showed:

- Early predictive models can be built even among unpopular GitHub projects.
- When early project data is transferred, they produce accurate and faster predictive models
- CPU intensive methods such as Hyper-parameter-optimization or non-explainable ensemble method like TLEL may not be required since early models with two features sufficed.

Perhaps we can simplify many SE tasks by learning early in the project life-cycle. Therefore, we would like to check for early trends in other domains within SE for future work.

Acknowledgements

This work was partially supported by NSF grant #XXX.

References

1. Amritanshu Agrawal, Wei Fu, Di Chen, Xipeng Shen, and Tim Menzies. How to"" dodge"" complex software analytics. *IEEE Transactions on Software Engineering*, 2019.
2. Amritanshu Agrawal, Wei Fu, and Tim Menzies. What is wrong with topic modeling? and how to fix it using search-based software engineering. *Information and Software Technology*, 98:74–88, 2018.
3. 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.
4. Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.
5. Sousuke Amasaki. Cross-version defect prediction: use historical data, cross-project data, or both? *Empirical Software Engineering*, pages 1–23, 2020.
6. Kwabena Ebo Bennin, Jacky W Keung, and Akito Monden. On the relative value of data resampling approaches for software defect prediction. *Empirical Software Engineering*, 24(2):602–636, 2019.
7. James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 2546–2554, USA, 2011. Curran Associates Inc.
8. James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.
9. Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E Hassan. Think locally, act globally: Improving defect and effort prediction models. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 60–69. IEEE, 2012.
10. Christian Bird, Tim Menzies, and Thomas Zimmermann. *The Art and Science of Analyzing Software Data*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.
11. Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.
12. David Bowes, Tracy Hall, and Jean Petrić. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26(2):525–552, 2018.
13. Lionel C Briand, Walcelio L. Melo, and Jurgen Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE transactions on Software Engineering*, 28(7):706–720, 2002.
14. Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261. IEEE, 2013.
15. Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, 2015.
16. 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.
17. Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.
18. Di Chen, Wei Fu, Rahul Krishna, and Tim Menzies. Applications of psychological science for actionable analytics. *FSE'19*, 2018.
19. Jinyin Chen, Yitao Yang, Keke Hu, Qi Xuan, Yi Liu, and Chao Yang. Multiview transfer learning for software defect prediction. *IEEE Access*, 7:8901–8916, 2019.
20. Lin Chen, Bin Fang, Zhaowei Shang, and Yuanyan Tang. Negative samples reduction in cross-company software defects prediction. *Information and Software Technology*, 62:67–77, 2015.
21. Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, and Chao Ni. Software defect number prediction: Unsupervised vs supervised methods. *Information and Software Technology*, 106:161–181, 2019.
22. Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. Multi: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology*, 93:1–13, 2018.
23. Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*, 2018.
24. Marco Dâ€™Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

25. Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016.
26. Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181, 2014.
27. Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800. IEEE, 2015.
28. Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.
29. Mark A Hall and Geoffrey Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data engineering*, 15(6):1437–1447, 2003.
30. A. Hassan. Remarks made during a presentation to the ucl crest open workshop, March 2017.
31. Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.
32. Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170–190, 2015.
33. Zhimin He, Fayola Peters, Tim Menzies, and Ye Yang. Learning from open-source projects: An empirical study on defect prediction. In *2013 ACM/IEEE international symposium on empirical software engineering and measurement*, pages 45–54. IEEE, 2013.
34. Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.
35. Steffen Herbold. Training data selection for cross-project defect prediction. In *Proceedings of the 9th international conference on predictive models in software engineering*, pages 1–10, 2013.
36. T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45, 2019.
37. Seyedrebvar Hosseini, Burak Turhan, and Mika M¨antyl¨a. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *Information and Software Technology*, 95:296–312, 2018.
38. Qiao Huang, Xin Xia, and David Lo. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering*, 24(5):2823–2862, 2019.
39. Shamsul Huda, Kevin Liu, Mohamed Abdelrazek, Amani Ibrahim, Sultan Alyahya, Hmood Al-Dossari, and Shafiq Ahmad. An ensemble oversampling model for class imbalance problem in software defect prediction. *IEEE access*, 6:24184–24195, 2018.
40. Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013.
41. Xiao-Yuan Jing, Fei Wu, Xiwei Dong, and Baowen Xu. An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Transactions on Software Engineering*, 43(4):321–339, 2016.
42. Xiaoyuan Jing, Fei Wu, Xiwei Dong, Fumin Qi, and Baowen Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 496–507, 2015.
43. Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.
44. Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.
45. Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.
46. Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 96–107, New York, NY, USA, 2016. ACM.
47. Ekrem Kocaguneli, Tim Menzies, and Emilia Mendes. Transfer learning in effort estimation. *Empirical Software Engineering*, 20(3):813–843, June 2015.
48. Masanari Kondo, Cor-Paul Bezemer, Yasutaka Kamei, Ahmed E Hassan, and Osamu Mizuno. The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering*, 24(4):1925–1963, 2019.

49. Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering*, 25(1):890–939, 2020.
50. A Güneş Koru, Khaled El Emam, Dongsong Zhang, Hongfang Liu, and Divya Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473–498, 2008.
51. A Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2008.
52. Rahul Krishna and Tim Menzies. Bellwethers: A baseline method for transfer learning. *IEEE Transactions on Software Engineering*, 45(11):1081–1105, 2018.
53. Rahul Krishna, Tim Menzies, and Wei Fu. Too much automation? the bellwether effect and its implications for transfer learning. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 122–131, 2016.
54. An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229. IEEE, 2017.
55. Issam H Laradji, Mohammad Alshayeb, and Lahouari Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58:388–402, 2015.
56. Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.
57. Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.
58. Zhiqiang Li, Xiao-Yuan Jing, Fei Wu, Xiaoke Zhu, Baowen Xu, and Shi Ying. Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction. *Automated Software Engineering*, 25(2):201–245, 2018.
59. Chao Liu, Dan Yang, Xin Xia, Meng Yan, and Xiaohong Zhang. A two-phase transfer learning model for cross-project defect prediction. *Information and Software Technology*, 107:125–136, 2019.
60. Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.
61. George Mathew, Amritanshu Agrawal, and Tim Menzies. Finding trends in software research. *IEEE Transactions on Software Engineering*, 2018.
62. Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2017.
63. Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822–834, 2012.
64. Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 343–351. IEEE, 2011.
65. Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2006.
66. Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.
67. Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54, 2008.
68. Ayse Tosun Misirli, Ayse Bener, and Resat Kale. Ai-based software defect predictors: Applications and benefits in a case study. *AI Magazine*, 32(2):57–68, 2011.
69. N. Mittas and L. Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Trans SE*, 39(4):537–551, April 2013.
70. Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.
71. Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, December 2017.
72. Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

73. Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2017.
74. Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *2013 35th international conference on software engineering (ICSE)*, pages 382–391. IEEE, 2013.
75. Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings - International Conference on Software Engineering*, pages 382–391, 2013.
76. Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings - International Conference on Software Engineering*, pages 382–391, 2013.
77. Chao Ni, Wang-Shu Liu, Xiang Chen, Qing Gu, Dao-Xu Chen, and Qi-Guo Huang. A cluster based feature selection method for cross-project software defect prediction. *Journal of Computer Science and Technology*, 32(6):1090–1107, 2017.
78. Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.
79. Sinno Jialin Pan, Ivor W Tsang, James T Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2010.
80. Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications*, 144:113085, 2020.
81. Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. Cross-project defect prediction models: L'union fait la force. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 164–173. IEEE, 2014.
82. Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.
83. Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 2019.
84. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
85. 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. *the Journal of machine Learning research*, 12:2825–2830, 2011.
86. Fayola Peters, Tim Menzies, Liang Gong, and Hongyu Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering*, 39(8):1054–1068, 2013.
87. Fayola Peters, Tim Menzies, and Lucas Layman. Lace2: Better privacy-preserving data sharing for cross project defect prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 801–811. IEEE, 2015.
88. Fayola Peters, Tim Menzies, and Andrian Marcus. Better cross company defect prediction. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 409–418. IEEE, 2013.
89. Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441. IEEE, 2013.
90. Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 424â€“434, New York, NY, USA, 2014. Association for Computing Machinery.
91. Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the"" imprecision"" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.
92. Santosh Singh Rathore and Sandeep Kumar. Towards an ensemble based system for predicting the number of software faults. *Expert Systems with Applications*, 82:357–382, 2017.
93. Wasiur Rhmann, Babita Pandey, Gufran Ansari, and Devendra Kumar Pandey. Software fault prediction based on change metrics using hybrid algorithms: An empirical study. *Journal of King Saud University-Computer and Information Sciences*, 32(4):419–424, 2020.
94. Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 966–969. ACM, 2015.
95. Duksan Ryu and Jongmoon Baik. Effective multi-objective na¨ıve bayes learning for cross-project defect prediction. *Applied Soft Computing*, 49:1062–1077, 2016.
96. Duksan Ryu, Okjoo Choi, and Jongmoon Baik. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering*, 21(1):43–71, 2016.

97. Duksan Ryu, Jong-In Jang, and Jongmoon Baik. A hybrid instance selection using nearest-neighbor for cross-project defect prediction. *Journal of Computer Science and Technology*, 30(5):969–980, 2015.
98. Duksan Ryu, Jong-In Jang, and Jongmoon Baik. A transfer cost-sensitive boosting approach for cross-project defect prediction. *Software Quality Journal*, 25(1):235–272, 2017.
99. Robert Sawyer. Bi‚Äô™s impact on analyses and decision making depends on the development of less complex applications. In *Principles and Applications of Business Intelligence Research*, pages 83–95. IGI Global, 2013.
100. Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.
101. Alon Sela and Hila Ben-Gal. Big data analysis of employee turnover in global media companies, google, facebook and others. pages 1–5, 12 2018.
102. Emad Shihab, Zhen Ming Jiang, Walid M Ibrahim, Bram Adams, and Ahmed E Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2010.
103. N. Shrikanth, S. Majumder, and T. Menzies. Early life cycle software defect prediction. why? how? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 448–459, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.
104. NC Shrikanth and Tim Menzies. Assessing practitioner beliefs about software defect prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 182–190. IEEE, 2020.
105. Michael J Siers and Md Zahidul Islam. Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. *Information Systems*, 51:62–71, 2015.
106. Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.
107. Qinbao Song, Yuchen Guo, and Martin Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 45(12):1253–1269, 2018.
108. Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1806–1817, 2012.
109. Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.
110. Shiang-Yen Tan and Taizan Chan. Defining and conceptualizing actionable insight: a conceptual framework for decision-centric analytics. *arXiv preprint arXiv:1606.03510*, 2016.
111. C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.
112. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, pages 321–332, 2016.
113. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2018.
114. Haonan Tong, Bin Liu, and Shihai Wang. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology*, 96:94–111, 2018.
115. Burak Turhan, Ayşe Tosun Mısırlı, and Ayşe Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology*, 55(6):1101–1118, 2013.
116. András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
117. Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 2018.
118. Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

119. Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.
120. Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.
121. Tiejian Wang, Zhiwu Zhang, Xiaoyuan Jing, and Liqiang Zhang. Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering*, 23(4):569–590, 2016.
122. Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.
123. Fei Wu, Xiao-Yuan Jing, Ying Sun, Jing Sun, Lin Huang, Fangyi Cui, and Yanfei Sun. Cross-project and within-project semisupervised software defect prediction: A unified approach. *IEEE Transactions on Reliability*, 67(2):581–597, 2018.
124. Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.
125. Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.
126. Zhou Xu, Jin Liu, Xiapu Luo, Zijiang Yang, Yifeng Zhang, Peipei Yuan, Yutian Tang, and Tao Zhang. Software defect prediction based on kernel pca and weighted extreme learning machine. *Information and Software Technology*, 106:182–200, 2019.
127. Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering*, 2020.
128. Meng Yan, Xin Xia, David Lo, Ahmed E Hassan, and Shanping Li. Characterizing and identifying reverted commits. *Empirical Software Engineering*, 24(4):2171–2208, 2019.
129. Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, 2017.
130. 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, 2016.
131. Jingxiu Yao and Martin Shepperd. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 120–129. 2020.
132. Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. Mining software defects: should we consider affected releases? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 654–665. IEEE, 2019.
133. Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.
134. Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, 43(5):476–491, 2016.
135. Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191, 2014.
136. Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering*, 21(5):2107–2145, 2016.
137. Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320. IEEE, 2016.
138. Yun Zhang, David Lo, Xin Xia, and Jianling Sun. An empirical study of classifier combination for cross-project defect prediction. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 264–269. IEEE, 2015.
139. Tianchi Zhou, Xiaobing Sun, Xin Xia, Bin Li, and Xiang Chen. Improving defect prediction with deep forest. *Information and Software Technology*, 114:204–216, 2019.

140. Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(1):1–51, 2018.
141. Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.