

The Early Bird Catches the Worm: Better Early Life Cycle Defect Predictors

N.C. Shrikanth and Tim Menzies

Abstract—Before researchers rush to reason across all available data, they should first check if the information is densest within some small region. We say this since, in 240 GitHub projects, we find that the information in that data “clumps” towards the earliest parts of the project. In fact, 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 (using weeks, not months, of CPU time). Also, we can find simple models (with just two features) that generalize to hundreds of software projects.

Based on this experience, we warn 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/early-defect-prediction-tse>.

Index Terms—defect prediction, software analytics, transfer learning

1 INTRODUCTION

In defect prediction, researchers usually are “data-hungry” and assume that if data is useful, then more data is much more useful. For example:

- “Long-term JIT models should be trained using a cache of plenty of changes” [1];
- “...as long as it is large; (prediction performance) is likely to be boosted more by the sample size” [2];
- “It is natural to think that accumulating multiple releases can be beneficial because it represents the variability of a project” [3].

In our work, we have sometimes seen unsatisfactory results from such a data-hungry 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 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 “data-lite” approach that just uses early life cycle data to predict for defects. The literature review of §3.4 shows that,

surprisingly, this approach to defect prediction has been overlooked by prior work. 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 “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 [4]–[53] needlessly complicated an inherently simple process.

The rest of this paper explains our method; and offers experimental evidence that this early life cycle transfer learning algorithm is simpler, faster, and more stable than other transfer learning methods. 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 [54]¹. We calculate that only Figure 1.A and 2.5 pages of this text (e.g. §6.1) come from that prior work.

• The authors are at the Department of Computer Science, North Carolina State University, Raleigh, USA. snaraya7@ncsu.edu, tim@ieee.org

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

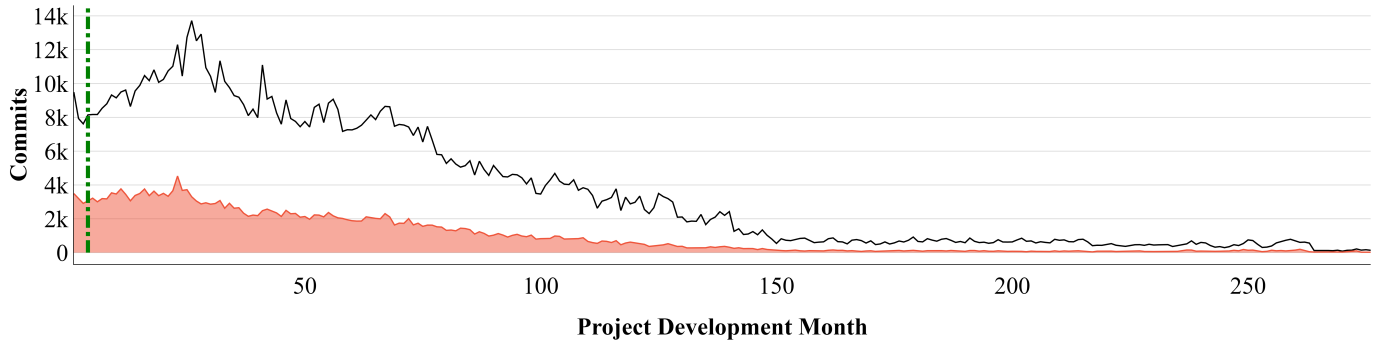


Figure 1.A: 155 popular GitHub projects ($\#stars > 1000$). Data from 1.2 million commits.

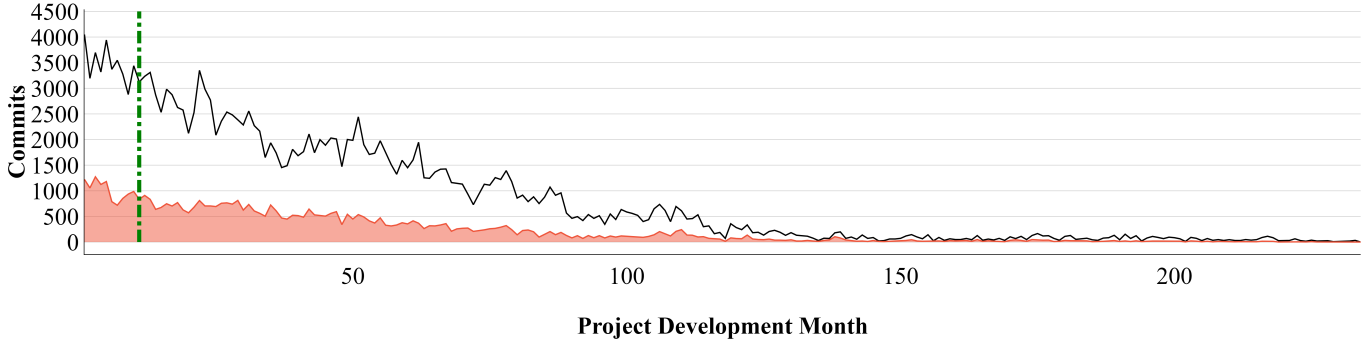


Figure 1.B: 85 unpopular GitHub projects ($\#stars < 1000$). Data from 253,289 commits.

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.

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 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 4, 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 [55]. 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 [56]. 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 [57].

Defect prediction uses data miners to input static code attributes and output models that predict where the code probably contains most bugs [58], [59]. The models learned in this way are very effective and relatively cheap to build:

- *Effective*: Misirili et al. [60], and Kim et al. [61] report considerable cost savings when such predictors are used in guiding industrial quality assurance processes.
- *Relatively simpler to implement* Also, Rahman et al. [62] 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.

3.2 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 [63]).

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:

- CPDP: *cross-project defect prediction*. Lessons learned from *other* projects and applied to *this* project.
- WPDP: *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 [3]. Interesting methods evolved in that research include:

- Heterogeneous transfer that lets data expressed in indifferent formats transferred from project to projects [40];
- Temporal transfer learning, which is a within-project defect prediction (WPDP) tool where earlier life cycle data is used to make predictions later in the life cycle [64].

This paper uses the same tools to collect data from all 240 projects. Also, recalling the vertical green lines in Figure 1, we are using prior project data to make predictions about the future. Hence, this paper is about a *non-heterogeneous temporal* transfer learning method.

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+ method [65]. 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 Data-Hungry versus Data-Lite

Our reading of the literature is most recent SE analytics papers have taken a data-hungry approach (e.g. see the quotes in our introduction from [1]–[3]). Nevertheless, just because a technique is popular does not mean that it should be recommended. We argue there that it can be useful to try data-lite before adopting data-hungry. 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 [66]. Finally, due to privacy concerns, teams even within the same organization may not readily make their matured project available for others to use [15].

For another thing, several researchers have reported that a data-hungry 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 [67].
- In over 30 years of COCOMO effort, Boehm could share cost estimation data from only 200 projects.

3.4 Related Work

One reason to explore early life cycle data reasoning is that it has not been done before. Much of the SE transfer learning

TABLE 1: 50 ‘highly cited’ (more than 10 citations per year) papers that ran one or more CPDP experiment(s) since 2010.

Year	#Data	#Features	Projects	Cites/Year	Paper
2011		20	2	16.44	[4]
		198	6	19.25	[5]
		17	10	39.13	[6]
		20	10	30.13	[7]
		20	7	23.38	[8]
		20	2	13.88	[9]
		8	9	25.38	[10]
		16	41	14	[11]
		17	8	48.43	[12]
		20	41	22.43	[13]
		20	10	19.57	[14]
		20	10	13.29	[15]
		20	14	13.29	[16]
		20	10	11.43	[17]
		54	12	32	[18]
		14	11	16.5	[19]
		20	10	20.5	[20]
		26	1398	18.5	[21]
		18	7	13.4	[22]
		20	11	19.8	[23]
		20	10	14.4	[24]
		20	17	12.4	[25]
		20	10	11.2	[26]
		28	11	24.4	[27]
		20	10	33	[28]
	None	X	26	35.5	[29]
		14	11	26.5	[30]
		14	6	19.75	[31]
		17	10	23.5	[32]
		20	10	35.5	[33]
		20	21	20.75	[34]
		20	30	11.75	[35]
		26	1390	16.25	[36]
		61	23	10.5	[37]
		X	10	70.25	[38]
		20	15	21.33	[39]
		61	34	80	[40]
		61	8	10.67	[41]
		6	58	21.5	[42]
		14	6	20.5	[43]
		20	11	23	[44]
		61	18	18.5	[45]
		61	28	14.5	[46]
		20	10	14	[47]
		61	16	19.5	[48]
		X	10	20	[49]
		14	6	17	[50]
		61	8	14	[51]
		20	7	20	[52]
		20	14	19	[53]

KEY:  Part,  Whole, **None** - No training data, **X** - Features automated

methods [11]–[15], [19], [20], [22]–[27], [30]–[37], [39]–[47], [50] could be characterized as “data-hungry” since they transferred all available data from one or more projects to build their predictors.

To understand more about data-hungry reasoning in SE, we performed the literature search summarized in Table 1. In March 2021, we found 982 articles in Google Scholar using the query (“cross-project defect prediction”) published since 2011. Following the advice of Agrawal et al. [68], and Mathews et al. [69] we focused 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 were not CPDP related) we found 50 papers that performed some transfer learning experiments. Within that set, we found three approaches to row selection:

- ‘Whole’ if the methods of that paper transfer all rows from one or more projects.
- ‘Part’ if the methods of that paper explore a large search space of one or more projects to find a small

2. Queried <https://scholar.google.co.in/> in 2020

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 [29]. Having recorded that method, this paper will not explore this minority approach.

Note that regardless of being “whole” 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.

When we tried to place this paper into Table 1, 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 1, we never look at all the data). Further, our “number of projects”=1 (which does not even appear in Table 1) and we only use a handful of features (far less than the features used by other work in Table 1).

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

3.4.1 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* data-lite and *all* data-hungry experiments into:

- data-lite: early-within and early-cross
- data-hungry: all-within and all-cross

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 [12], [37]). 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 §4.1.1 for the algorithm details and for a discussion of what algorithms we selected. But, in summary, to the best of our knowledge, the data-lite variants of these techniques have not been explored before in SE.

4 EXPERIMENTAL METHODS

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

4.1 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 2, Figure 2 and Figure 3.

To mine this data, we looked into the dataset curated by Munaiah et al. [72] that has numerous criteria to differentiate an engineering project from a trivial one. Then we mined all the projects using Commit-Guru [71] similar to this work [54]. Projects were rejected according to the standard sanity checks by SE GitHub miners [54], [73]:

- Less than 1% defects;
- 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. [72]. 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).

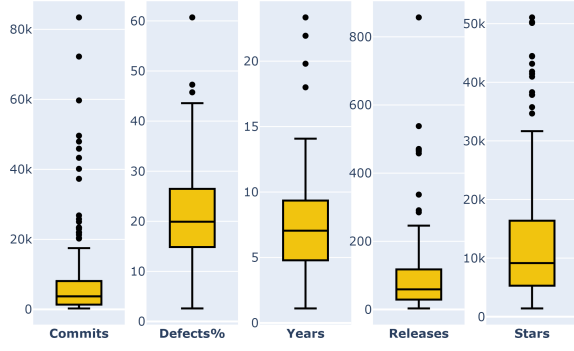
The projects collected in this way were developed in widely-used programming languages (including Java, Python, C, C++, C#, Kotlin, JavaScript, Ruby, PHP, Fortran, Go, Shell, etc.) for various domains.

All 14 features used in this study are listed in Table 2. Those features are extracted from these projects using *Commit Guru* [71]. The use of these particular features has been prevalent and endorsed by prior studies [18], [70]. *Commit Guru* publicly available tool used in numerous works [75], [76] 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

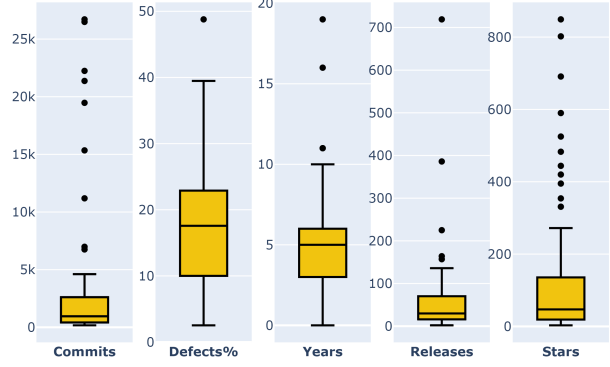
TABLE 2: 14 Commit level features that Commit Guru tool [70], [71] mines from GitHub repositories

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	#developers 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



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).

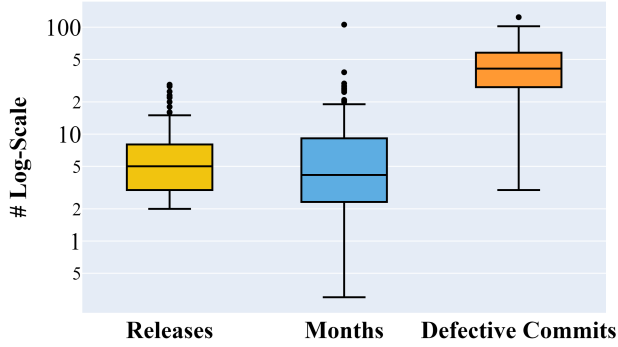
(a) Popular projects (stars > 1000).



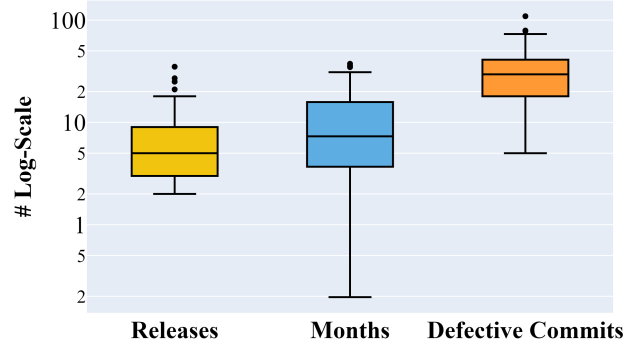
Distributions seen in all 258,000+ commits of all 89 *unpopular* projects: median values of commits (957), percent of defective commits (18%), life span in years (5), releases (30) and stars (47).

(b) Unpopular projects (stars < 1000).

Fig. 2: Data distributions.



(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)
Data-hungry	CPDP	Bellwether [37]	CFS, SMOTE and steps illustrated in §4.1	Selected by CFS.	All commits from the identified cross project.
		TCA+ [12]	SMOTE	5 components with linear kernel (data supplied with all features)	Pick first 150 and last 150 commits from the bellwether project.
Data-lite	CPDP	$*E_{Bellwether}^+$	Steps illustrated in §4.1 in [70], [74], [75]	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).
		$*E_{TCA+}^+$	Steps illustrated in §4.1	2 components with linear kernel (data supplied only with LA and LT)	
Data-lite	WPDP	E [54] $*E^+$	CFS and steps illustrated in §4.1 Steps illustrated in §4.1	Selected by CFS LA and LT	(same as above)

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

TABLE 3: This table lists three baselines approaches (where two of them are CPDP and the remaining one is a WPDP) along with three data-lite variants to support this study. Note many of these methods used the steps advised in [70], [74], [75]. Those steps are discussed in §4.1. Also, for a discussion on CFS, see §4.2.

NUC (number of unique changes to the modified files before) dividing by NF (number of modified files) [74], [77]. 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 [78].

Similar to the back-trace approach by SZZ algorithm [79] each commit was labeled “defective” (based on certain defect-related keywords) or “clean” (otherwise) internally by *Commit Guru*. We walked back in the code to find the changes associated with that commit.

This empirical study uses three sets of algorithms:

- The six classification algorithms described in §4.1.1;
- Pre-processing algorithms (for some sampling policies) described in §4.2;
- The seven sampling methods are described in §4.2.1.

4.1.1 Classifiers

We use all the six classifiers used in the early WPDP study [54]. The six classifiers are prevalent in the SE literature chosen about Ghotra et al. [80] work that extensively compared 30+ defect prediction algorithms in four ranks.

Those classifiers were:

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

4.1.2 Pre-processors

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 1, some frequency counts of different methods published in [54] and using own engineering judgement, the following methods are representative of much of the prior research in this field:

- The steps advised by [74], [77], [78] (see §4.1);
- Correlation-based Feature Selection (CFS, see §4.2);
- Synthetic Minority Over-Sampling (SMOTE, see §4.2).

Not all combinations of pre-processing steps are valid. Table 3 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 [68]).

4.2 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 [81]. That is often used in building supervised defect prediction

models, especially while building defect predictors [77]. 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:

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

- *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 *rff*.

Synthetic Minority Over-Sampling (SMOTE): A sampling technique proposed by Chawla et al. [82] 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 [68], [83].

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 3. Early ‘E’ based policies are balanced with an equal number of defective and clean commits by definition.

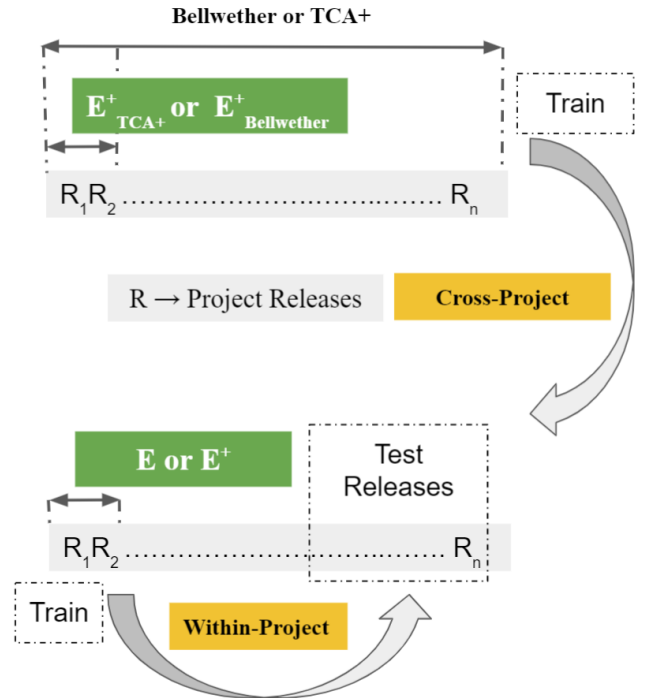


Fig. 4: A visual map of our methodology to gauge the sampling policies listed in Table 3.

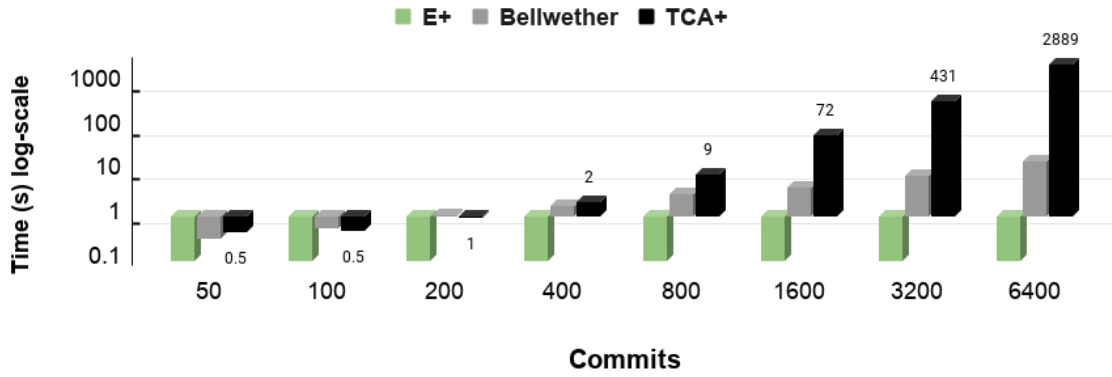


Fig. 5: Time taken by different sampling policies to train a model based on the number of commits.

4.2.1 Sampling Methods

Figure 4 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 1 for standard sampling policies, CPDP techniques, projects, and features. We found numerous CPDP techniques reported in the past decades; for example, Sousuke’s work recently (2020) explored 24 CPDP approaches [3]. In the context of WPDP, Shrikanth et al. identified many sampling strategies (based on recent releases, recent months of data, and all past data) to build predictors [54]. However, in the context of CPDP, we found only two (‘whole’ or ‘part’). Nevertheless, both ‘whole’ or ‘part’ are data-hungry. The focus of this study is to check the efficacy of using data-lite approaches in prevalent CPDP approaches and not rank numerous CPDP techniques. We chose two representative CPDP techniques, specifically *TCA+* and *Bellwether*, because:

- *TCA+* is an active CPDP technique in this space based on the seminal work by Nam et al. [12].
- *Bellwether*, a recent (2018) baseline approach by Krishna et al. that performed better than *TCA+* [45]

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

Bellwether Project (*Bellwether*): All the commits from a *bellwether* project are used as the training data [45]. 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.* [45]”. According to Krishna et al. [45]” 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 [84]. Extending that, Nam et al. stacked many normalization rules on top of *TCA* to form *TCA+* [85]. 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 [45] report *TCA+* to perform better than other transfer methods, namely Transfer Naive Bayes [6] and Value Cognitive Boosting Learner [32].

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

- Choose a bellwether project randomly from all the bellwether projects.
- Figure 5 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 six 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 restrict to only 300 training commits (2X of 150 commits) rather than consider all the project commits.
 - 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 WPDP Shrikanth et al. showed that using all the project commits is needless for predicting defects [54].

For *TCA+* related policies listed in Table 3 we reused the implementation from their replication package³ by Kondo

3. <https://sailhome.cs.queensu.ca/replication/featred-vs-featsel-defectpred/>

et al., which is an EMSE'19 article about feature reduction techniques on defect prediction models [77].

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

Early Bellwether ($E_{Bellwether}^+$): 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. [54]). All the data features are removed except two size-based features, 'LA' and 'LT.' The rationale for choosing two features are discussed later in §4.5.

Early TCA+ (E_{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. [54]). All the data features are removed except two size-based features, 'LA' and 'LT.' The rationale for choosing two features are discussed later in §4.5.

Then, to compare CPDP with WPDP techniques, we chose E that is endorsed by Shrikanth et al. in [54] 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 $E_{Bellwether}^+$, we created E^+ .

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 [54].

Early Sampling with few features (E^+): Training data is curated same as E but it is restricted to two size based features 'LA' and 'LT' listed in Table 2. The rationale for this is discussed later in §5.

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 §4.2.

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

4.3 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 [1], [21], [70], [75], [83], [86]–[93] 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 [86].

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 have 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").

4.3.1 Brier

Brier is the absolute predictive accuracy measure. Numerous defect prediction papers [1], [75], [83], [89] 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 [94]).

Then:

$$Brier = \frac{1}{C} \sum_{i=1}^C (y_i - \hat{y}_i)^2 \quad (1)$$

4.3.2 Initial number of False Alarms (IFA)

Based on the observation by Parnin and Orso [95] 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.

4.3.3 Recall

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

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (2)$$

4.3.4 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).

$$PF = \frac{False\ Positives}{False\ Positives + True\ Negatives} \quad (3)$$

4.3.5 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.

4.3.6 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 $Recall = 1$ & $PF = 0$ [96].

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

4.3.7 G-measure (GM)

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

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

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 [54] and in this work shown later in §5, 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*.

4.4 Statistical Tests

Predictors built using each of the sampling policies listed in Table 3 are tested on all appropriate (future) project releases. Then we compute each of the evaluation criteria discussed above in §4.3. 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 [97]. 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 [98] 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*.

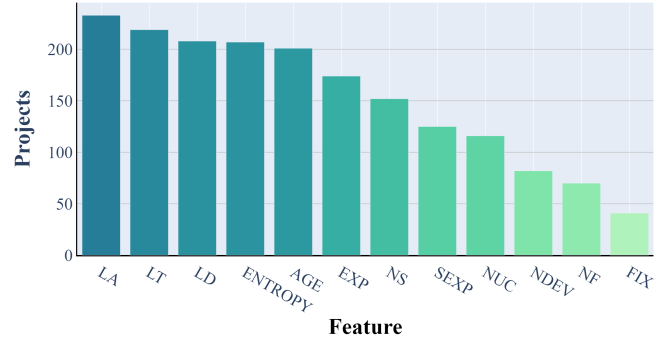


Fig. 6: Frequency of features chosen by the CFS feature selector (using just the first 150 commits) sampled using *E* in all the 240 projects

4.5 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.

4.5.1 Finding Important Features:

Shrikanth et al.’s early sampling rule ‘E’ shown to be adequate for WPDP [54]. 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 6 shows the frequency of features selected by the CFS algorithm (see §4.2) 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

4.5.2 Finding Representative Bellwethers:

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 7 shows that *more* of the models found via the *E*⁺ method was “satisfactory” than those found using all

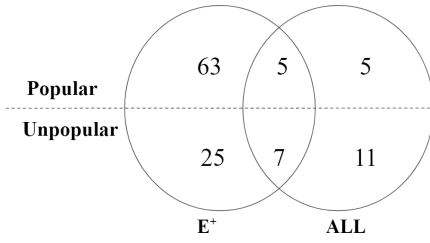


Fig. 7: Number of “satisfactory” bellwethers identified independently by two different sampling policies in all 240 projects.

the data. That effect is particularly marked in the popular projects⁴.

Later in this paper, we will present results comparing models built in this way to a variety of other treatments.

5 RESULTS

Table 4 shows our results. In the first row of that table “+” 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 “data-lite” methods (and all other rows are “data-hungry”). The ‘wins’ column (see column #1) counts how often a particular sampling policy/classifier achieves a top score. 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 §4.4).

Looking at the pattern of gray cells, some of the criteria (IFA and Brier and PF) turn out to be uninformative:

- IFA turns out to tie nearly all the time (as witnessed by the numerous gray cells in the column).
- Also, achieving best ‘PF’ and best ‘Brier’ comes at the expense of low ‘Recall’ results (see *Bellwether+SVM* or *RF*, one third the way down the table).

Looking at the top row of that table (marked in yellow), we see that a cross-company model early life cycle model ($E^+_{Bellwether}$, LR) achieves the best results on all the criteria (except Brier). Hence we say:

Conclusion1: For defect prediction, if you have access to many other projects, use any bellwether both identified and built using just LA and LT from the first 150 commits.

Of course, not all projects have access to data from many other projects. In that case, we note the result on row5 of Table 4 that shows (E^+ , LR). This within-project treatment is statistically not as good as our top-ranked treatment. That

4. 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 ($i \leq 12$) different bellwether projects with nearly equal frequency (see the seven unpopular + 5 popular bellwethers in the middle of Figure 7). The statistical analysis of the 12 $E^+_{Bellwether}$ using §4.4 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.

said, it ties with three other near-top treatments, and its performance is close to our best. Hence we say:

Conclusion2: For defect prediction, if you have only access to your own project, wait for 150 commits, then build a model using your own data having only the LA and LT features.

For completeness, we offer two more observations:

- Firstly, prior state-of-the-art transfer learning methods (*Bellwether*, *TCA+*) appear below $E^+_{Bellwether}$. This result further endorses the efficacy of early life cycle reasoning.
- Secondly, this paper is an extension of prior work [54] that used features selected by CFS. As shown in the yellow line at row 20 of that table, models learned via LA and LT perform better than models learned using features selected by CFS on every release. Hence, we no longer recommend that part of our prior results.

5.1 Threats to Validity

5.1.1 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 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.

5.1.2 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 [54]. Secondly, in Table 4 we compared our results with multiple classifiers, and Logistic-Regression based predictors were in-par and better than five other classifiers listed in §4.1.1.

Further, an empirical study can only focus on a handful of representative classifiers. Thus we chose six classifiers (Logistic Regression, Nearest neighbor, Decision Tree, Random Forrest, and Naïve Bayes). These six classifiers cover a broad range of classification algorithms [80].

5.1.3 Evaluation bias

This paper uses both ‘Recall’ and ‘PF’ to identify bellwethers and seven evaluation measures (Recall, PF, IFA, Brier, GM, D2H, 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 [86].

TABLE 4: 36 defect prediction models (both WPDP and CPDP) tested in 12,379 applicable releases from 155 popular and 85 unpopular GitHub projects. 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 “data-lite” methods (and all other rows are “data-hungry”). 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 §4.4). The ‘wins’ column (see column #1) counts how often a particular sampling policy/classifier achieves a top score.

Type	Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	GM+	IFA-	
CPDP	$E^+_{Bellwether}$	LR	5	0.85	0.34	0.73	0.32	0.31	0.76	1.0	← best transfer learning
CPDP	$E^+_{Bellwether}$	SVM		0.9	0.4	0.71	0.35	0.35	0.74	1.0	
CPDP	E^+_{TCA+}	SVM		0.88	0.39	0.7	0.35	0.34	0.74	1.0	
CPDP	E^+_{TCA+}	LR	4	0.82	0.33	0.72	0.32	0.31	0.75	1.0	
WPDP	E^+	LR		0.77	0.26	0.73	0.31	0.27	0.73	1.0	← best within-project learning
WPDP	E^+	NB		0.89	0.44	0.68	0.39	0.38	0.71	2.0	
CPDP	E^+_{TCA+}	KNN		0.8	0.33	0.7	0.35	0.32	0.72	1.0	
WPDP	E^+	SVM		0.8	0.29	0.71	0.35	0.29	0.71	1.0	
CPDP	$E^+_{Bellwether}$	KNN	3	0.78	0.3	0.7	0.34	0.3	0.72	1.0	
CPDP	E^+_{TCA+}	RF		0.75	0.32	0.69	0.35	0.32	0.71	1.0	
CPDP	$Bellwether$	SVM		0.22	0.04	0.58	0.56	0.19	0.26	1.0	
CPDP	$Bellwether$	RF		0.2	0.03	0.57	0.57	0.19	0.23	1.0	
CPDP	$E^+_{Bellwether}$	NB		0.88	0.5	0.64	0.42	0.42	0.67	2.0	
CPDP	E^+_{TCA+}	NB		0.83	0.44	0.67	0.39	0.39	0.7	2.0	
WPDP	E^+	KNN	2	0.7	0.26	0.7	0.34	0.29	0.68	1.0	
CPDP	$Bellwether$	KNN		0.25	0.06	0.59	0.53	0.2	0.29	1.0	
CPDP	$Bellwether$	LR		0.77	0.33	0.67	0.39	0.32	0.66	1.0	
CPDP	E^+_{TCA+}	DT		0.71	0.32	0.67	0.36	0.33	0.68	2.0	
WPDP	E	SVM		0.71	0.31	0.68	0.36	0.31	0.67	1.0	
CPDP	$E^+_{Bellwether}$	RF		0.71	0.29	0.68	0.36	0.3	0.68	1.0	
CPDP	$TCA+$	LR		0.7	0.5	0.62	0.42	0.45	0.64	2.0	
WPDP	E	LR		0.7	0.31	0.67	0.37	0.32	0.66	1.0	← prior result [54]
WPDP	E	KNN		0.67	0.31	0.66	0.39	0.33	0.63	1.0	
CPDP	$E^+_{Bellwether}$	DT		0.67	0.3	0.65	0.38	0.33	0.64	1.0	
WPDP	E^+	RF		0.67	0.27	0.68	0.36	0.29	0.65	1.0	
WPDP	E^+	DT	1	0.64	0.3	0.65	0.38	0.32	0.62	1.0	
WPDP	E	RF		0.62	0.27	0.63	0.42	0.32	0.56	2.0	
WPDP	E	DT		0.6	0.36	0.58	0.48	0.38	0.52	2.0	
CPDP	$TCA+$	DT		0.5	0.33	0.56	0.48	0.38	0.48	2.0	
CPDP	$TCA+$	RF		0.5	0.3	0.58	0.48	0.36	0.49	2.0	
CPDP	$TCA+$	SVM		0.5	0.22	0.61	0.45	0.31	0.51	2.0	
CPDP	$Bellwether$	NB		0.5	0.15	0.64	0.41	0.25	0.53	1.0	
CPDP	$TCA+$	KNN		0.38	0.18	0.58	0.49	0.29	0.41	2.0	
CPDP	$Bellwether$	DT		0.33	0.13	0.59	0.48	0.25	0.38	1.0	
WPDP	E	NB	0	0.55	0.31	0.54	0.57	0.37	0.37	3.0	
CPDP	$TCA+$	NB		0.22	0.19	0.5	0.61	0.33	0.23	3.0	

KEY: All data-lite sampling methods are shaded (■).

5.1.4 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.

6 DISCUSSION

6.1 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 [4]. 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 [4], [66], [99].

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 [100] 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 [101]. From Kim et al. [102] perspective, software analytics is a way to obtain fruitful insights that guide practitioners to accomplish software development goals, whereas for Tan et al. [103] such insights are a central goal. From a practitioner’s perspective, Bird et al. [104] 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.

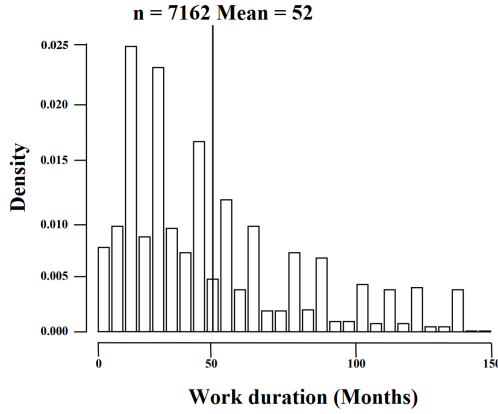


Fig. 8: Work duration histograms on particular projects; from [105]. Data from: Facebook, eBay, Apple, 3M, Intel and Motorola.

Tool development and Training: Previously [99] 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 8. 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 8, in seven years (84 months), the majority of workers and managers would first 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.

6.2 Systems Implications

Figure 9 shows the run-time required to fund bellwether projects in all 240 projects using *Bellwether* or $E_{Bellwether}^+$ 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 7 show when sampled using E^+ 30% “more projects”

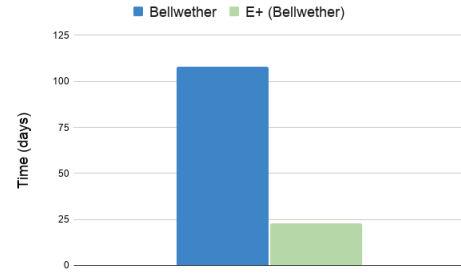


Fig. 9: Time taken to identify qualifying bellwether projects in all 240 projects using *Bellwether* and $E_{Bellwether}^+$ policies. Note that in our experiments, we ran on a multi-core cloud-based CPU farm. To collect the data here (which assumes we are running on a single-core machine), we summed 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).

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 data-lite nature of our methods can also help to gauge sophisticated techniques like TCA+ faster.

6.3 Research Implications

We have shown that when defect data contains information, that information may be densest in 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?”.

As to other future work, given the simplicity of the data-lite approach, we are keen to explore hyper-parameter optimization (HPO) [106]–[108]. HPO is often not applied in software analytics due to its computational complexity. Perhaps that complexity can be avoided by focusing only on small samples of data early in the life cycle.

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” [109].
- 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 [1], [75], [107], [110].

More generally, before researchers focus just on later life cycle data, they need to first first, check that that their (e.g.) buggy commit data occurs at equal frequency across the life cycle.

ACKNOWLEDGEMENTS

This work was partially supported by an NSF-CISE grant #1908762.

REFERENCES

- [1] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2017.
- [2] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 2013, pp. 147–157.
- [3] S. Amasaki, "Cross-version defect prediction: use historical data, cross-project data, or both?" *Empirical Software Engineering*, pp. 1–23, 2020.
- [4] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 343–351.
- [5] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.
- [6] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.
- [7] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.
- [8] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on software engineering*, vol. 39, no. 6, pp. 822–834, 2012.
- [9] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 60–69.
- [10] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the"" imprecision"" of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.
- [11] B. Turhan, A. T. Misirlı, and A. Bener, "Empirical evaluation of the effects of mixed project data on learning defect predictors," *Information and Software Technology*, vol. 55, no. 6, pp. 1101–1118, 2013.
- [12] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 382–391.
- [13] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 409–418.
- [14] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 252–261.
- [15] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1054–1068, 2013.
- [16] S. Herbold, "Training data selection for cross-project defect prediction," in *Proceedings of the 9th international conference on predictive models in software engineering*, 2013, pp. 1–10.
- [17] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *2013 ACM/IEEE international symposium on empirical software engineering and measurement*. IEEE, 2013, pp. 45–54.
- [18] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 432–441.
- [19] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. 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*, 2014, pp. 172–181.
- [20] A. Panichella, R. Oliveto, and A. 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)*. IEEE, 2014, pp. 164–173.
- [21] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 182–191.
- [22] D. Ryu, J.-I. Jang, and J. Baik, "A hybrid instance selection using nearest-neighbor for cross-project defect prediction," *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 969–980, 2015.
- [23] L. Chen, B. Fang, Z. Shang, and Y. Tang, "Negative samples reduction in cross-company software defects prediction," *Information and Software Technology*, vol. 62, pp. 67–77, 2015.
- [24] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 264–269.
- [25] F. Peters, T. Menzies, and L. Layman, "Lace2: Better privacy-preserving data sharing for cross project defect prediction," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 801–811.
- [26] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Defect prediction as a multiobjective optimization problem," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 426–459, 2015.
- [27] X. Jing, F. Wu, X. Dong, F. Qi, and B. 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*, 2015, pp. 496–507.
- [28] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, vol. 59, pp. 170–190, 2015.
- [29] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 309–320.
- [30] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.
- [31] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 157–168.
- [32] D. Ryu, O. Choi, and J. Baik, "Value-cognitive boosting with a support vector machine for cross-project defect prediction," *Empirical Software Engineering*, vol. 21, no. 1, pp. 43–71, 2016.
- [33] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on software Engineering*, vol. 42, no. 10, pp. 977–998, 2016.
- [34] X.-Y. Jing, F. Wu, X. Dong, and B. Xu, "An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 321–339, 2016.
- [35] D. Ryu and J. Baik, "Effective multi-objective naïve bayes learning for cross-project defect prediction," *Applied Soft Computing*, vol. 49, pp. 1062–1077, 2016.
- [36] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model with rank transformed predictors," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2107–2145, 2016.
- [37] R. Krishna, T. Menzies, and W. 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*, 2016, pp. 122–131.
- [38] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.
- [39] D. Ryu, J.-I. Jang, and J. Baik, "A transfer cost-sensitive boosting approach for cross-project defect prediction," *Software Quality Journal*, vol. 25, no. 1, pp. 235–272, 2017.
- [40] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 874–896, 2017.
- [41] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project soft-

- ware defect prediction," *Journal of Computer Science and Technology*, vol. 32, no. 6, pp. 1090–1107, 2017.
- [42] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? an examination of cross-project defect prediction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 1, pp. 1–51, 2018.
- [43] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," *Information and Software Technology*, vol. 93, pp. 1–13, 2018.
- [44] S. Hosseini, B. Turhan, and M. M^aantyl^a, "A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction," *Information and Software Technology*, vol. 95, pp. 296–312, 2018.
- [45] R. Krishna and T. Menzies, "Bellwethers: A baseline method for transfer learning," *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1081–1105, 2018.
- [46] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," *Automated Software Engineering*, vol. 25, no. 2, pp. 201–245, 2018.
- [47] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2018.
- [48] F. Wu, X.-Y. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, and Y. Sun, "Cross-project and within-project semisupervised software defect prediction: A unified approach," *IEEE Transactions on Reliability*, vol. 67, no. 2, pp. 581–597, 2018.
- [49] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *arXiv preprint arXiv:1802.00921*, 2018.
- [50] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.
- [51] J. Chen, Y. Yang, K. Hu, Q. Xuan, Y. Liu, and C. Yang, "Multiview transfer learning for software defect prediction," *IEEE Access*, vol. 7, pp. 8901–8916, 2019.
- [52] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect number prediction: Unsupervised vs supervised methods," *Information and Software Technology*, vol. 106, pp. 161–181, 2019.
- [53] C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang, "A two-phase transfer learning model for cross-project defect prediction," *Information and Software Technology*, vol. 107, pp. 125–136, 2019.
- [54] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 448–459. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00050>
- [55] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.
- [56] F. Akiyama, "An example of software system debugging," in *IFIP Congress (1)*, vol. 71, 1971, pp. 353–359.
- [57] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *IEEE Transactions on Software Engineering*, 2018.
- [58] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.
- [59] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, vol. 33, no. 1, pp. 2–13, 2006.
- [60] A. T. Misirli, A. Bener, and R. Kale, "Ai-based software defect predictors: Applications and benefits in a case study," *AI Magazine*, vol. 32, no. 2, pp. 57–68, 2011.
- [61] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 151–160.
- [62] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 424–434. [Online]. Available: <https://doi.org/10.1145/2568225.2568269>
- [63] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE transactions on Software Engineering*, vol. 28, no. 7, pp. 706–720, 2002.
- [64] E. Kocaguneli, T. Menzies, and E. Mendes, "Transfer learning in effort estimation," *Empirical Software Engineering*, vol. 20, no. 3, pp. 813–843, Jun. 2015. [Online]. Available: <http://link.springer.com/10.1007/s10664-014-9300-5>
- [65] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings - International Conference on Software Engineering*, 2013, pp. 382–391.
- [66] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. 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*, 2009, pp. 91–100.
- [67] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, 2008.
- [68] A. Agrawal and T. Menzies, "Is"" better data"" better than"" better data miners""?" in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1050–1061.
- [69] G. Mathew, A. Agrawal, and T. Menzies, "Finding trends in software research," *IEEE Transactions on Software Engineering*, 2018.
- [70] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.
- [71] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 966–969.
- [72] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.
- [73] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, 2020.
- [74] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.
- [75] M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 890–939, 2020.
- [76] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.
- [77] M. Kondo, C.-P. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno, "The impact of feature reduction techniques on defect prediction models," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1925–1963, 2019.
- [78] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. 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*, 2010, pp. 1–10.
- [79] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: <http://doi.acm.org/10.1145/1082983.1083147>
- [80] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *37th ICSE-Volume 1*. IEEE Press, 2015, pp. 789–800.
- [81] M. A. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Transactions on Knowledge and Data engineering*, vol. 15, no. 6, pp. 1437–1447, 2003.
- [82] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.
- [83] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on

- defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2018.
- [84] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Transactions on Neural Networks*, vol. 22, no. 2, pp. 199–210, 2010.
- [85] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings - International Conference on Software Engineering*, 2013, pp. 382–391.
- [86] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of the 4th international workshop on Predictor models in software engineering*, 2008, pp. 47–54.
- [87] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.
- [88] K. E. Bennin, J. W. Keung, and A. Monden, "On the relative value of data resampling approaches for software defect prediction," *Empirical Software Engineering (ICSE)*, IEEE, 2019, pp. 602–636, 2019.
- [89] 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*, pp. 1–1, 2018.
- [90] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: should we consider affected releases?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 654–665.
- [91] M. Yan, X. Xia, D. Lo, A. E. Hassan, and S. Li, "Characterizing and identifying reverted commits," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2171–2208, 2019.
- [92] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 476–491, 2016.
- [93] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.
- [94] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [95] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 2011, pp. 199–209.
- [96] D. Chen, W. Fu, R. Krishna, and T. Menzies, "Applications of psychological science for actionable analytics," *FSE'19*, 2018.
- [97] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *IEEE Trans SE*, vol. 39, no. 4, pp. 537–551, Apr. 2013.
- [98] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
- [99] N. Shrikanth and T. Menzies, "Assessing practitioner beliefs about software defect prediction," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2020, pp. 182–190.
- [100] A. Hassan, "Remarks made during a presentation to the ucl crest open workshop," Mar. 2017.
- [101] R. Sawyer, "Bi's impact on analyses and decision making depends on the development of less complex applications," in *Principles and Applications of Business Intelligence Research*. IGI Global, 2013, pp. 83–95.
- [102] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 96–107. [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884783>
- [103] S.-Y. Tan and T. Chan, "Defining and conceptualizing actionable insight: a conceptual framework for decision-centric analytics," *arXiv preprint arXiv:1606.03510*, 2016.
- [104] C. Bird, T. Menzies, and T. Zimmermann, *The Art and Science of Analyzing Software Data*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.
- [105] A. Sela and H. Ben-Gal, "Big data analysis of employee turnover in global media companies, google, facebook and others," 12 2018, pp. 1–5.
- [106] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *ICSE 2016*. ACM, 2016, pp. 321–332.
- [107] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135–146, 2016.
- [108] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to" dodge" complex software analytics," *IEEE Transactions on Software Engineering*, 2019.
- [109] 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)*, 2019, pp. 34–45.
- [110] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.