Early Life Cycle Software Defect Prediction. Why? How?

N.C. Shrikanth, Suvodeep Majumder and Tim Menzies

Department of Computer Science
North Carolina State University
Raleigh, USA
snaraya7@ncsu.edu, smajumd3@ncsu.edu, timm@ieee.org

Abstract—Many methods in defect prediction are “data-hungry”; i.e. (1) given a choice of using more data, or some smaller sample, researchers assume that more is better; (2) when data is missing, researchers take elaborate steps to transfer data from another project; and (3) given a choice of older data or some more recent sample, researchers usually ignore older data.

Based on the analysis of hundreds of popular Github projects (with 1.2 million commits), we suggest that for defect prediction, there is limited value in such data-hungry approaches. Data for our sample of projects last for 84 months and contains 3,728 commits (median values). Across these projects, most of the defects occur very early in their life cycle. Hence, defect predictors learned from the first 150 commits and four months perform just as well as anything else.

This means that, contrary to the “data-hungry” approach, (1) small samples of data from these projects are all that is needed for defect prediction; (2) transfer learning has limited value since it is needed only for the first 4 of 84 months (i.e. just 4% of the life cycle); (3) after the first few months, we need not continually update our defect prediction models.

We hope these results inspire other researchers to adopt a “simplicity-first” approach to their work. Certainly, there are domains that require a complex and data-hungry analysis. But before assuming complexity, it is prudent to check the raw data looking for “short cuts” that simplify the whole analysis.

I. INTRODUCTION

In some domains, there is much evidence for the superiority of a data hungry analysis. In his famous talk “The Unreasonable Effectiveness of Data” Google’s Chief Scientist Peter Norvig argues that “billions of trivial data points can lead to understanding” [1]. To support that claim, he offers numerous examples through vision research where adequate learners grow superior, merely by learning from gigabytes more data.

But what if some Software Engineering (SE) data was not like vision data? What if SE needs its own AI methods, based on what we learned about the specifics of software projects? If that were true, then data-hungry methods might be needless over-embellishments of a fundamentally simple process.

The paper argues that 96% of the time, we neither want nor need data-hungry defect prediction. Defect prediction uses data miners to input static code attributes and output models that predict where the code probably contains most bugs [2], [3]. Wan et al. [4] reports that there is much industrial interest in these predictors since they can guide the deployment of more expensive and time-consuming quality assurance methods (e.g., human inspection). Misirili et al. [5] and Kim et al. [6] report considerable cost savings when such predictors are used in guiding industrial quality assurance processes. Also, Rahman et al. [7] show that such predictors are competitive with more elaborate approaches.

In defect prediction, data-hungry researchers assume that if data is useful, then even more data is much more useful. For example: “…as long as it is large; the resulting prediction performance is likely to be boosted more by the size of the sample than it is hindered by any bias polarity that may exist” [8]. “It is natural to think that a closer previous release has more similar characteristics and thus can help to train a more accurate defect prediction model. It is also natural to think that accumulating multiple releases can be beneficial because it represents the variability of a project” [9]. “Long-term JIT models should be trained using a cache of plenty of changes” [10]. Data hungry in defect prediction researchers go to great lengths to avoid data starvation. For example, early on in a project, there may be insufficient local data for data mining. In that circumstance, data-hungry researchers apply transfer learning to map data from other projects. To say the least, transfer learning is a very active area of software engineering [11]. Not only are researchers hungry for data, they are most hungry for the most recent data. For example: Hoang et al. say ‘We assume that older commits changes may have characteristics that no longer effect the latest commits” [12]. 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 [10], [13], [15]. In a project with multiple releases, recent validation ignores any insights that are available from older releases.

The rest of this paper compares data-hungry defect prediction against a much more “data-lite” early life cycle defect prediction method. In §2 we note that there are negative consequences for data-hungry methods that constantly revise defect prediction models whenever new data arrives. After that (in §3) we show that for hundreds of Github [16] projects, the defect data from the latter life cycle defect data is relatively uninformative. This leads to the definition of experiments in
the early life cycle defect prediction (see §4, §5). Results from those experiments (in §6) show that for hundreds of popular Github projects, some data is useful, but more data is not more useful. Specifically: our Github projects usually last 84 months and make 3,728 commits (median values). Defect predictors after just four months of work just as well as anything else. This means that techniques like transfer learning are useful only for 4% of the life cycle (i.e., 4 months out of 84).

Before beginning, we digress to make three points. Firstly, to encourage replication, all our data and scripts are on-line. Secondly, to be clear, our early life cycle defect predictors are competitive, but not necessarily “better”, than other methods (where “better” is assessed via the standard measures shown in §V-C like AUC, D2H, IFA, Brier, Recall, G-measure, and PF). Rather, we recommend early life cycle defect prediction methods (where “better” is assessed via the standard measures shown in §V-C like AUC, D2H, IFA, Brier, Recall, G-measure, and PF). Rather, we recommend early life cycle defect products since they are easier to build and do not need continual adjustment for the rest of the life cycle (and this has positive business implications— see next section).

Thirdly, we are not arguing that all SE domains can be simplified with a little early life cycle sampling. Some domains need extensive data collection (e.g. autonomous cars need much data to train their vision systems). Certainly, there are domains that require a complex and data-hungry analysis. Before assuming complexity, it is prudent to check the raw data looking for “short cuts” that simplify the whole analysis.

II. PROBLEMS WITH DATA HUNGRY DATA MINING

This section argues that, at least for defect prediction, there are good reasons to avoid data-hungry defect prediction methods. After this, the rest of this paper will argue that we do not need such methods.

One perennial problem with defect prediction is “conclusion instability” where new data leads to different models. Conclusion instability is well documented. For example, Zimmermann et al. [17] learned defect predictors from 622 pairs of projects project1, project2. In only 4% of pairs, predictors from project1 worked on project2. Also, Turhan et al. [18] studied defect prediction results from 28 recent studies, most of which offered widely differing conclusions about what most influences software defects. Menzies et al. [19] reported experiments where data for software projects are clustered, and data mining is applied to each cluster. They report that very different models are learned from different parts of the data, even from the same projects.

In our own past work, we have found conclusion instability, meaning there we had to throw years of data. In one sample of Github data, we sought to learn everything we could from 700,000+ commits. Roughly, the web slurping required for that process took 500 days of CPU (using five machines with 16 cores, over 7 days). Within that data space, we found significant differences in the models learned from different parts of the data. So even after all that work, we were unable to offer our business users a stable predictor for their domain.

Is that the best we can do? Are there general defect prediction principles we can use to guide project management, software standards, education, tool development, and legislation about software? Or is software engineering some “patchwork quilt” of ideas and methods where it only makes sense to reason about specific, specialized, and small sets of related projects? Note that if the software was a “patchwork” of ideas, then there would be no stable conclusions about what constitutes best practice for software engineering (since those best practices would keep changing as we move from project to project). Such conclusion instability would have detrimental implications for trust, insight, training, and tool development.

Trust: Conclusion instability is unsettling for project managers. Hassan [20] warns that managers lose trust in software analytics as its results keep changing. Such instability prevents project managers from offering 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 insights are essential to catalyze business initiative [21]. From Kim et al. [22] perspective software analytics is a way to obtain fruitful insights that guide practitioners accomplish software development goals, whereas for Tan et al. [23] such insights are a central goal. From a practitioners perspective Bird et al. [24] report, insights occur when users respond to software analytics models. Frequent model generation could exhaust users abilities on confident conclusions from new data.

Tool development and Training: Shrikanth and Menzies [25] warn that unstable models make it hard to on-board novice software engineers. Without knowing what factors most influence local project, it is hard to design and build appropriate tools for quality assurance activities.

All these problems with trust, insight, training, and tool development could 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. As mentioned in the introduction, we study here Github projects spanning 84 months and containing

https://github.com/anonymoussresearcher/EarlyDefectPrediction

Fig. 1: Work duration histograms from [26]. Data from: Facebook, eBay, Apple, 3M, Intel and Motorola.
3,728 commits (median values). Within that data, we have found that models learned after just 150 commits (and four months of data collection) perform just as well as anything else. In terms of resolving conclusion instability, this is a very significant result since it means that for $4/84 = 96\%$ 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 1. That plot shows that software employees usually change jobs every 52 months (either moving between companies or changing projects within an organization). This means that 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.

III. Why Early Defect Prediction Might Work

A. Github Results

Figure 2 shows data from 1.2m Github commits from 155 popular Github projects (the criteria for selecting those particular projects is detailed below). Note how the frequency defect data (shown in red) starts collapsing early in the life cycle (after 12 months). This observation suggests that it might be relatively uninformative to learn from later life cycle data. This was an interesting finding since, as mentioned in the introduction, 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 [10], [13], [14]. In terms of Figure 2, that strategy would train on red dots taken near the right-hand-side, then test on the most right-hand-side dot. Given the shallowness of the defect data in that region, such recent validation could lead to results that are not representative of the whole life cycle.

Accordingly, we sat out to determine how different training and testing sampling policies across the life cycle of Figure 2 affected the results. After much experimentation (described below), we assert that if data is collected up until the vertical green line of Figure 2 then that generates a model as good as anything else.

B. Related Work

Before moving on, we first discuss related work on early life cycle defect prediction. In 2008, Fenton et al. [61] explored the use of human judgment (rather than data collected from the domain) to handcraft a causal model to predict residual defects (defects caught during independent testing or operational usage) [61]. Fenton needed two years of expert interaction to build models that compete with defect predictors learned by data miners from domain data. Hence we do not explore those methods here since they were very labor-intensive.

In 2020, Arokiam & Jeremy [62] explored bug severity prediction. They show it is possible to predict bug-severity early in the project development by using data transferred from other projects [62]. Their analysis was on the cross-projects, but unlike this paper, they did not explore just how early in the life cycle did within project data became effective.

In similar work to Arokiam & Jeremy, in 2020, Sousuke [9] explored another early life cycle, Cross-Version defect prediction (CVDP) using Cross-Project Defect Prediction (CPDP) data. Their study was not as extensive as ours (only 41 releases). CVDP uses the project’s prior releases to build defect prediction models. Sousuke compared defect prediction models trained using three within project scenarios (recent project release, all past releases, and earliest project release) to endorse recent project release. Sousuke also combined CVDP scenarios using CPDP (24 approaches) to recommend that the recent project release was still better than most CPDP approaches. However, unlike Sousuke, we offer contrary evidence in this work, as our endorsed policy based on earlier commits works similar to all other prevalent policies (including most recent release) reported in the literature. Notably, we assess our approach on 1000+ of releases and evaluate on seven performance measures.

In summary, as far as we can tell, ours is the first study to perform an extensive comparison of prevalent sampling policies practiced in the defect prediction space.

IV. Sampling Policies

One way to summarize this paper is that we evaluate a novel “stop early” sampling policy for collecting the data needed for defect prediction. This section describes a survey of sampling policies in defect prediction. Each sampling policy has its own way of extracting training and test data from a project. As shown below, there is a remarkably diverse number of policies in the literature which, prior to this paper, have not been systematically and comparatively evaluated.

In April 2020, we found 737 articles in Google Scholar using the query (“Software” AND “Defect prediction” AND...
TABLE I: Papers discussing different sampling policies. All (papers that utilize all historical data to build defect prediction models, shaded in gray).

| Paper Year | Citations | Sampling | Projects |
|------------|-----------|----------|----------|
| 2008       | 172       | All      | 12       |
| 2010       | 21        | All      | 5        |
| 2010       | 347       | Percentage | 10     |
| 2011       | 94        | All      | 8        |
| 2012       | 264       | All      | 11       |
| 2012       | 387       | All      | 10       |
| 2013       | 322       | All      | 10       |
| 2014       | 105       | All      | 1,403    |
| 2014       | 44        | Release  | 1        |
| 2014       | 93        | Release  | 5        |
| 2015       | 129       | All      | 7        |
| 2016       | 87        | All      | 10       |
| 2016       | 42        | Release  | 23       |

“just in time”, “software” AND “defect prediction” AND “sampling policy”). “Just in time (JIT)” defect prediction is a widely-used approach where the code seen in each commit is assessed for its defect prone-ness [13], [14], [38], [63].

From the results of that query, we applied some temporal filtering: (i) we examined all articles more recent than 2017; (ii) for older articles, we examined all papers from the last 15 years with more than 10 citations per year. After reading the title and abstracts and the methodology sections, we found the 39 articles of Table I that argued for particular sampling policies.

Figure 3 shows a high-level view of the sampling policies seen in the Table I papers:

- **All**: When the historical data (commits/files/modules etc) is used for evaluation within some cross-validation study (where the data is divided randomly into N bins and the data from bin ‘i ∈ N’ is used to test a model trained from all other data) [64].
- **Percentage**: When the historical data is stratified by some percentage, like 80-20%. The minimum % we found was 67% [31].
- **Release**: When models are trained on the immediate or more past releases in order to predict defects on the current release [55].
- **Month**: When 3 or 6 months of historical data is used to predict defects in future files, commits, or release [10].
- **Slice**: When an arbitrary stratification is used to divide the data based on a specific number of days (like 180 days or six months in [45]).

It turns out Figure 3 is only an approximation of the diverse number sampling policies we see in the literature. A more comprehensive picture is shown in Figure 4 where we divide software releases $R_i$ that occur over many months $M_j$ into some train and test set.

Using a little engineering judgement, and guided by the frequency of different policies (from Figure 3) we elected to focus on four sampling policies from the literature and one ‘early stopping’ policy, see Table II. The % share in Figure 3 show ‘ALL and RR’ are prevalent practices whereas ‘M3 and M6’ though not prevalent are used in related literature [10], [12]. We did not consider separate policies for ‘Percentage’ and ‘Slice’ as the former is similar to ‘ALL’ (100% of historical data), and the later is least prevalent and similar to M6 (180 days or six months).

Note the “magic numbers” in Table II:

- **3 months, 6 months**: these are thresholds often seen in
V. METHODS

A. Data

This section describes the data used in this study as well as what we mean by “clean” and “defective” commits.

All our data comes from open source Github projects [16] that enables other researchers to reproduce our results. But SE researchers have warned against using all Github data since this website contains many projects that might be categories as “non-serious”. Accordingly, following the advice of prior researchers [65], [66], we ignored projects with

- Less than 1000 stars;
- Less than 1% defects;
- Less than two releases;
- Less than one year of activity;
- No license information.

Less than 5 defective and 5 clean commits.

This resulted in 155 projects developed in many languages (including Python, Java, C, C++, C#, JavaScript, Ruby, Kotlin, PHP, Fortran, Go, Shell, etc.). This selection includes popular projects like postgres, elasticsearch, and scikit-learn.

For features from these projects, we used Commit Guru [67]. This is a publicly available tool based on a 2015 ESEC/FSE paper, which has been used in numerous works [14], [68]. Commit Guru provides a portal where it takes the Github repository (project) URL (input), and it internally mines from Github repositories

| Policy | Method |
|--------|--------|
| ALL    | Train using all past software commits ([0, R_i]) in the project before the first commit in the release under test R_i. |
| M6     | Train using the recent six months of software commits ([R_i - 6months]) made before the first commit in the release under test R_i. |
| M3     | Train using the recent three months of software commits ([R_i - 3months]) made before the first commit in the release under test R_i. |
| RR     | Train using the software commits in the previous release R_{i-1} before the first commit in the release under test R_i. |
| E      | Train using early 50 commits (25 clean and 25 defective) randomly sampled within the first 150 commits before the first commit in the release under test R_i. |

| Feature | Definition |
|---------|------------|
| NS      | Number of modified subsystems |
| ND      | Number of modified directories |
| NF      | Number of modified files |
| ENTROPY | Distribution of modified code across each file |
| LA      | Lines of code added |
| LD      | Lines of code deleted |
| LT      | Lines of code in a file before the change |
| FIX     | Whether the change is defect fixing ? |
| NDEV    | Number of developers that changed the modified files |
| AGE     | The average time interval from the last to the current change |
| NUC     | Number of unique changes to the modified files before |
| EXP     | Developer experience |
| EXP     | Recent developer experience |
| NUC     | Developer experience on a subsystem |

With those changes were then summarized by Commit Guru using the attributes of Table III. These attributes became the independent attributes used in our analysis. Note that the use of these particular attributes has been endorsed by prior studies [38], [70]. Each such data point was then labeled “defective” or “clean” via Commit Guru’s categorization of the commit associated with that change.

Figure 5 shows information on our selected projects. As shown in that figure, our projects have:

- Median life spans of 84 months with 59 releases;
- The projects have (265, 3,728, 83,409) commits (min,
Figure 6 focuses on just the data used in the early life cycle (except the boolean variable ‘FIX’) to alleviate skewness [73].

Another pre-processor that was applied to some sampling policies was Synthetic Minority Over-Sampling, or SMOTE. When the proportion of defective and clean commits (or modules, files, etc.) is not equal, learners can struggle to find the target class. SMOTE, proposed by Chawla et al. [75] is often applied in defect prediction literature to overcome this problem [28], [52]. To achieve balance, SMOTE artificially synthesizes examples (commits) extrapolating using K-nearest neighbors (minimum five commits required) in the data set (training commits in our case) [75]. Note that:

- We do not apply SMOTE to policies that already guarantee class balancing. For example, our preferred early life-cycle method selects at random 25 defective, and 25 non-defective (clean) commits from the first 150 commits.
- Also, just to document that we avoided a potential methodological error [28], we record here that we applied SMOTE to the training data, but never the test data.

C. Evaluation Criteria

Defect prediction studies evaluated their model performance using a variety of criteria. From the literature, we used what we judged to be the most widely-used measures [10], [14], [27], [31], [38], [40], [42], [45], [47], [52], [53], [55], [76]. For the following seven criteria:

- Nearly all have the range 0 to 1 (except Initial number of False Alarms, which can be any positive number);  
- Four of these criteria need to be minimized: D2H, IFA, Brier, PF; i.e., for these criteria less is better.
- Three of these criteria need to be maximized: AUC, Recall, G-Measure; i.e., for these criteria more is better.

One measure we do not apply is precision since prior work has shown that this measure has significant issues for unbalanced data [27].

1) Brier: Recent defect prediction papers [10], [14], [31], [52] measure the model performance using the Brier absolute predictive accuracy 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 (calculated from the loss functions in scikit-learn classifiers [77]). Then:

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

(1)

2) Initial number of False Alarms (IFA): Parnin and Orso [78] report that developers lose faith in this kind of analytics if they encounter many initial false alarms. Thus IFA is simply the number of false alarms encountered after sorting the commits in the order of probability of being defective, then counting the number of false alarms before finding the first true alarm.

3) Recall: Recall is the proportion of inspected defective commits among all the actual defective commits.

\[ \text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \]  

(2)

4) False Positive Rate (PF): The proportion of predicted defective commits those are not defective among all the predicted defective commits.

\[ \text{PF} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \]  

(3)

5) Area Under the Receiver Operating Characteristic curve (AUC): AUC is the area under the curve between the true positive rate and false-positive rate.

6) Distance to Heaven (D2H): D2H or “distance to heaven” aggregates on two metrics Recall and False Positive Rate (PF) to show how close to “heaven” where Recall is 1 & PF is 0 [79].

\[ D2H = \frac{\sqrt{(1 - \text{Recall})^2 + (0 - \text{PF})^2}}{\sqrt{2}} \]  

(4)

7) G-measure (GM): A harmonic mean between Recall and the compliment of PF measured, as shown below.

\[ G - \text{Measure} = \frac{2 \times \text{Recall} \times (1 - \text{PF})}{\text{Recall} + (1 - \text{PF})} \]  

(5)

Even though GM and D2H combined the same underlying measures, we include both here since they both have been used separately in the literature. Also, as shown below, it is not necessarily true that achieving good results on GM means that good results will also be achieved with D2H.

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

- A learner can achieve 100% recall just by declaring that all examples belong to the target class. This method will incur a high false alarm rate.
- A learner can achieve 0% false alarms just by declaring that no examples belong to the target class. This method will incur a very low recall rate.
- Similarly, Brier and Recall are also antithetical since reducing the loss function also means missing some conclusions and lowering recall.

D. Statistical Test

Later in §VI we compare distributions of evaluation measures of various sampling policies that may have the same median while their distribution could be very different. Hence to identify significant differences (rank) among two or more populations, we use the Scott-Knott test recommended by Mittas et al. in TSE’13 paper [80]. This test is a top-down bi-clustering approach for ranking different treatments, sampling policies in our case. This method sorts a list of l sampling policy evaluations with ls measurements by their median score. It then splits l into sub-lists m, n in order 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 ∪ n, the “best” division maximizes E(∆); i.e. the difference in the expected mean value before and after the split:

\[ E(\Delta) = \frac{m_s}{l_s} \text{abs}(m.\mu - l.\mu)^2 + \frac{n_s}{l_s} \text{abs}(n.\mu - l.\mu)^2 \]

We also employ the conjunction of bootstrapping and A12 effect size test by Vargha and Delaney [81] to avoid “small effects” with statistically significant results.

Important note: we apply our statistical methods separately to all the evaluation criteria; i.e., when we compute ranks, we do so for (say) false alarms separately to recall.

E. Experimental Rig

By definition, our different sampling policies have different train and different test sets. But, methodologically, when we compare these different policies, we have to compare results on the same sets of releases. To handle this we:

- First, run all our six policies, combined with all our six learners. This offers multiple predictions to different commits.
- Next, for each release, we divide the predictions into those that come from the same learner:policy pair. These divisions are then assessed with statistical methods described above.

VI. RESULTS

Tables IV and V show results when our six learners applied our five sampling policies. For two reasons, we split these results into two tables:

- One of our research questions needs to compare “very early” to “most recent” sampling; hence Table V.
- Our policies lead to results with different samples sizes: the recent release, or “RR”, the policy uses data from just two releases while “ALL” uses everything.

In the first row of those tables, “+” and “-” denote criteria that need to be maximized or minimized, respectively.

Within the tables, gray cells show results from our statistical tests (conducted separately on each criterion). Anything ranked “best” is colored gray and all other results are have white backgrounds.

Columns one and two show the policy/learners that lead to these results. Rows are sorted by how often policy/learners
TABLE IV: 24 defect prediction models tested in all 4,876 applicable project releases. In the first row “+” and “−” denote the criteria that need to be maximized or minimized, respectively. ‘Wins’ is the frequency of the policy found in the top #1 Scott-Knott rank in each of the seven evaluation measures (the cells shaded in gray).

Note that we do explore a fourth research issue: are different learners better at learner from a little, a lot, or all the available data. Based on our results, we have nothing definitive to offer on that issue. That said, if we were pressed recommend a particular learning algorithm, then we say there are no counterexamples to the claim that “it is useful to apply CFS+LR”.

**RQ1: Is more data, better?**

**Belief1**: Our introduction included examples where proponents of data-hungry methods advocated that if data is useful, then even more data is much more useful.

**Prediction1**: If that belief was the case, then in Table IV data-hungry sampling policies that used more data should defeat “data-lite” sampling policies.

**Observation1**: In Table IV our “data hungriest” sampling policy (ALL) loses on on most criteria. While it achieves the highest Recall (83%), it also has the highest false alarm range (40%). As to which other policy is preferred in the "wins=4" zone of Table IV there is no clear winner. What we would say here is that our preferred “data-lite” method called “E” (that uses 25 defective and 25 non-defective commits selected at random from the first 150 commits) is competitive with the rest. Hence:

**Answer1**: For defect prediction, it is not clear that more data is inherently better.
TABLE V: 12 defect prediction models tested on 3,704 project releases. In the first row “+” and “-” denote criteria that need to be maximized or minimized, respectively. ‘Wins’ is the frequency of the policy found in the top #1 Scott-Knott rank in each of the seven evaluation measures (the cells shaded in gray).

RQ2: When is “transfer learning” useful?

Belief2: “Transfer learning” is the art of taking data from other projects and applying it to the local project [11]. As discussed in our introduction, this is an active and prominent area of software analytics research. The premise of transfer learning is that there is a need to transfer; i.e., there is insufficient local data to build a satisfactory model.

Observations2: Figure 5 of this paper showed that within our sample of projects, we have data lasting a median of 84 months. Figure 6 noted that by the time we get to 150 commits, most projects are 4 months old (median values). The “E” results of Table IV showed that defect models learned from that 4 months of data are competitive with all the other policies studied here. Hence we say,

Answer2: The region of time when transfer learning is required is usually 4/84 months (i.e. 4%) of a project.

This is not to say that we should not study transfer learning—it is certainly useful for software that completes in less than four months. However, when software becomes useful, it can be maintained and repaired for many years. As shown here, for useful software that is not built then discarded before four months’ time, there are methods other than transfer learning for learning useful defect prediction models.

RQ3: When is more recent data better than older data?

Belief3: As discussed earlier in our introduction, many researchers prefer using recent data over data from earlier periods. For example, 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 [10], [13]–[15]. For a project with multiple releases, recent validation ignores any insights that are available from older releases.

Prediction3: If recent data is comparatively more informative than older data, then defect predictors built on recent data should out-perform predictors built on much older data.

Observations3: We observe that:

- Figure 5 of this paper showed that within our sample of projects, we have data lasting a median of 84 months.
- Figure 6 noted that by the time we get to 150 commits, most projects are 4 months old (median values).
- Table V says that “E” wins over “RR” since it falls in the best wins=4 section.
- Hence we could conclude that older data is more effective than recent data.

That said, we feel a somewhat more the circumspect conclusion is in order. When we compare E+LR to the next learner in that table (RR+NB) we only a very small difference in their performance scores. Hence we make a somewhat humbler conclusion:

Answer3: Recency based methods perform no better than results from early life cycle defect predictors.

This is a startling result, for two reasons:

- Compared to the “RR” training data, the “E” training data is very old indeed. For projects lasting 84 months long, “RR” is trained on information from recent few months, with “E” data comes from years before that.
- This result calls into question any conclusion made in a paper that used recent validation to assess their approach; e.g. [10], [13]–[15].

VII. THREATS TO VALIDITY

A. Sampling Bias

Generalizability of the conclusion will depend upon the samples considered; i.e., what matters here may not be true everywhere. To improve the generalizability of our conclusion, we mined 155 popular (>1000 stars) OS projects (having
over 1.2m commits) developed in numerous programming languages.

B. Learner bias

Any single study can only explore a handful of classification algorithms. For building the defect predictors in this work, we elected six learners (Logistic Regression, Nearest neighbor, Decision Tree, Random Forrest, and Naïve Bayes). These six learners represent a plethora of classification algorithms [71].

C. Evaluation bias

This paper uses seven evaluation measures (Recall, PF, IFA, Brier, GM, D2H, and AUC). Other prevalent measures in this defect prediction space include precision. However, as mentioned earlier, precision has issues with unbalanced data [27].

D. Input Bias

Our proposed sampling policy ‘E’ randomly samples 50 commits from early 150 commits. Thus it may be true that different executions could yield different results. However, this is not a threat, because each time the early policy ‘E’ randomly samples 50 commits from early 150 commits to test sizeable 8,490 releases (from Table IV and Table V) across all the six learners. In other words, our conclusions about ‘E’ hold on a large sample size of numerous releases.

VIII. Conclusion

The paper argues that for 96% of the time, we neither want nor need data-hungry defect prediction. The case for not wanting to apply data-hungry methods was made in §2. To repeat: when data keep changing, the models keep changing, so the conclusions become unstable. In such a fluid environment, conclusion instability means no one has a stable basis for making decisions or communicating insights.

Issues with conclusion instability disappear if, early in the life cycle, we can learn a defect prediction model that is effective for the rest of the project. We showed that, indeed, it is possible to learn such a predictive model. In our sample of projects lasting seven years, or more, we found that after four months, we could learn a predictor that works for the rest of the project. This means that, contrary to the “data-hungry” approach, (1) small samples of data from these projects are all that is needed for defect prediction; (2) transfer learning has limited value since it is needed only for the first 4 of 84 months (i.e. just 4% of the life cycle); (3) after the first few months, we need not continually update our defect prediction models.

As for future work, we have many suggestions:

- This paper offers a refutable conclusion: later life-cycle data performs no better than very early life cycle data. We hope that researchers regard this result as a research challenge which they should try to refute (perhaps using the latest generation of deep learning algorithms that promise better automated feature engineering).
- It would be useful to explore other samples of projects to check if (a) the pattern of Figure 2 exists in other data sets and, if so, (b) is it possible to build effective early life cycle defect predictors in those domains.
- This paper has only been about defect prediction. Perhaps there are other SE domains that respond well to a “data-lite” early life cycle prediction approach.
- We need to revisit all prior results that used recent validation to assess their approach; e.g. [10], [13], [15] since our RQ3 suggests they may have been working in a relatively uninformative region of the data.
- Proponents of transfer learning might consider developing a cost/benefit model that clearly identifies when to use that method (or, alternatively, when to use something else like our methods).
- While the performance scores of Tables IV and V are reasonable, there is still much room for improvement. Perhaps if we augmented early life cycle defect predictors with a little transfer learning, then we could generate better performing predictors.
- Further to the last point, another interesting avenue of future research might be hyper-parameter optimization [15], [82], [83]. Such optimization can be very slow due to the large space of possible parameters. One way to find better predictors would be to speed up that optimization, perhaps by focusing only on very small data sets. Since early life cycle predictors work so well, we conjecture that optimization that only uses small samples from early in the life cycle might be very effective indeed.

Acknowledgements

This work was partially supported by a NSF grant.

References

[1] P. Norvig. (2011) The Unreasonable Effectiveness of Data. Youtube. [Online]. Available: https://www.youtube.com/watch?v=ycDCZhjYWsc
[2] 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.
[3] T. Menzies, J. Greenwald, and A. Frank, “Data mining static code attributes to learn defect predictors,” IEEE transactions on software engineering, no. 1, pp. 2–13, 2007.
[4] 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.
[5] 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.
[6] 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.
[7] 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. ACM, 2014, pp. 424–434.
[8] F. Rahman, D. Posnett, I. Herreza, 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.
[9] S. Amasaki, “Cross-version defect prediction: use historical data, cross-project data, or both?” Empirical Software Engineering, pp. 1–23, 2020.
[10] 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.
