Better Predictors for Issue Lifetime

Mitch Rees-Jones
NC State, USA
mwreesjo@ncsu.edu

Matthew Martin
Colby College, USA
mjmartin@colby.edu

Tim Menzies
NC State, USA
tim.menzies@gmail.com

***Abstract*—Predicting issue lifetime can help software developers, managers, and stakeholders effectively prioritize work, allocate development resources, and better understand project timelines. Progress had been made on this prediction problem, but prior work has reported low precision and high false alarms. The latest results also use complex models such as random forests that detract from their readability.**

We solve both issues by using small, readable decision trees (under 20 lines long) and correlation feature selection to predict issue lifetime, achieving high precision and low false alarms (medians of 71% and 13% respectively). We also address the problem of high class imbalance within issue datasets - when local data fails to train a good model, we show that cross-project data can be used in place of the local data. In fact, cross-project data works so well that we argue it should be the default approach for learning predictors for issue lifetime.

***Index Terms*—Issue lifetime prediction, issue tracking, effort estimation, prediction.**

I. Introduction

We propose an improvement to recent results by Kikas, Dumas, and Pfahl (hereafter, KDP) at MSR'16 [8]. In their paper, an *issue* is a bug report, and the time required to mark it closed is the *issue close time*. Predicting this time has multiple benefits for the developers, managers, and stakeholders. It helps software developers better prioritize work, helps managers effectively allocate resources and improve consistency of release cycles, and helps project stakeholders understand changes in project timelines and budgets. It is also useful to predict issue close time when an issue is created; e.g. to send a notification if it is predicted that the current issue is an easy fix.

Much of the machinery of this paper is the same as KDP. However, prior to learning, we run Hall's CFS feature selector to prune irrelevant features [7]. Hall et al. note that the effect of feature selection can be quite dramatic, particularly for decision tree learners. If a data set has, say, four features that are strongly associated with the target class, a binary decision tree learner would quickly use those features to divide the data $2^4 = 16$ ways. This means that subsequent learning, at lower levels of tree, can only reason about $1/16$-th of the data. On the other hand, if those four features are strongly associated with each other, feature selection would remove all but one of them prior to learning. The decision tree learner would then use that single feature to divide the data twice, after which point subsequent learning at lower levels of tree could reason better (since it can access more data).

This prediction by Hall et al. proved to be the key to dramatically improving the KDP results:

- KDP report precisions under 25%, and false alarm rates of over 60%. On the other hand, using CFS, we can report precisions over 66% and false alarms under 20%.
- KDP use Random Forests to build their models. Random Forests are hard to read and reason from. In our approach, CFS plus single decision tree learning yielded easily comprehensible trees (under 20 lines).
- We also find that if local data fails to build a good model, then cross-project data filtered by CFS can be used to build effective predictors. In fact, cross-project data works so well, that we argue it should be the default approach for learning predictors for issue lifetime.

The success of cross-project learning for lifetime prediction was quite unexpected. The cross-project results of this paper were achieved without the data adjustments recommended in the defect prediction transfer learning literature (see relevancy filtering [9], [15] or dimensionality transform [11]). That is, while transfer learning for some software analytics tasks can be complex, there exist other tasks (such as predicting issue lifetimes) where it is a relatively simple task.

Overall, the contributions of this paper are:

- A new "high water mark" in predicting issue close time;
- A new "high water mark" in cross-project prediction.
- A method for repairing poor local performance using cross-project learning.
- Evidence for that some software analytics tasks allow for the very simple transfer of data between projects.

This paper is organized as follows. The *Background* section summarizes the current state of issue lifetime prediction. Next, in the *Methods* section, we describe our learners and data (we use issues, and code contributors come from GitHub and JIRA projects, from which we extracted ten issue datasets with a minimum, median, and maximum number of issues of 1,434, 5,266, 12,919 per dataset, respectively). Our *Results* section presents experimental results that defend three claims:

- *Claim #1*: Our predictors had low false alarms and higher precisions than KDP.
- *Claim #2*: Cross-project learning works remarkably effectively in this domain.
- *Claim #3*: Our predictors are easily comprehensible.

To assist other researchers, a reproduction package with all our scripts and data is available in Github[1] and in archival form, tagged with a DOI[2] (to simplify any future reference).

[1] github.com/reesjones/issueCloseTime
[2] doi.org/10.5281/zenodo.197111

II. Related Work

Panjer [12] explored predicting the time to fix a bug using data known at the beginning of a bug's lifetime. He used logistic regression models to achieve 34.9% accuracy in classifying bugs as closing in 1.4 days, 3.4 days, 7.5 days, 19.5 days, 52.5 days, 156 days, and greater than 156 days.

Giger, Pinzger, and Gall [4] used decision tree learning to make prediction models for issue close time for Eclipse, Gnome, and Mozilla bugs. They divided the time classes into thresholds of 1 day, 3 days, 1 week, 2 weeks, and 1 month, using static features for initial predictions that achieved a mean precision and recall of 63% and 65%. They also extended their models to include non-code features, which resulted in a 5-10% improvement in model performance. Their predictions using non-code features were validated with 10-fold cross validation, meaning their train/test splits in the cross validation could have used an issue's data at a future point in time to predict its close time in the past. Our methods (both local and cross-project approaches) avoid this potential conflation since all features used in our prediction are static, not temporal.

Bhattacharya and Neamtiu [1] studied how existing models "fail to validate on large projects widely used in bug studies". In a comprehensive study, they find that the predictive power (measured by the coefficient of determination R^2) of existing models is 30-49%. That study found that there is no correlation between bug-fix likelihood, bug-opener reputation, and time required to close the bug for the three datasets used in their study. Our results agree and disagree with this study. Like Bhattacharya and Neamtiu, we find that no single feature is always most predictive of issue close time. That said, we do find that in different projects that different features are highly predictive for issue close time.

Guo, Zimmerman, Nagappan, and Murphy [5] evaluated the most predictive factors that affect bug fix time for Windows Vista and Windows 7 software bugs. Unlike Bhattacharya and Neamtiu's work [1], they found that bug-opener reputation affected fix time; an issue with a high-reputation creator was more likely to get fixed. Bugs are also more likely to get fixed if the bug fixer is in the same team as or in geographical proximity of the bug creator. Guo et. al. conclude that the factors most important in bug fix time are social factors such as history of submitting high-quality bug reports and trust between teams interacting over bug reports. This conclusion of process metrics over product metrics is endorsed by [13]. Our results could be viewed as a partial replication of that study. Like Guo et al. we find that non-code features are most predictive of issue close time (and this is a *partial replication* since our findings come from different projects to those used by Guo et al.). Also, our results extend those of Guo et al. since we also check how well our preferred non-code features work in across 10 projects.

Marks, Zou, and Hassan [10] found that time and location of filing the bug report were the most important factors in predicting Mozilla issues, but severity was most important for Eclipse issues. Priority was not found to be an important factor for either Mozilla or Eclipse. Their models produced lower performance metrics (65% misclassification rate) than subsequent work.

Zhang, Gong, and Versteeg [17] reviewed the above work and concluded that standard methods were insufficient to handle predicting issue lifetimes. Hence, they developed an interesting, but intricate, new approach:

- A Markov-based method for predicting the number of bugs that will be fixed in the future;
- A method for estimating the total amount of time required to fix them;
- A kNN-based classification model for predicting a bug's fix time (slow or fast fix).

One reason to prefer our approach is that our methods can be implemented with a very simple extension to current data mining tool kits. Zhang et al.'s approach is interesting, but we find that a very small change to standard methods (i.e. prepending learning with CFS) leads to comparable results. Also, from our experience, the mathematical formalisms used in Zhang, et al. are difficult for some users to understand and thus gain deeper insight from. Therefore, we would prefer to use human-readable methods; e.g. data miners that learn single trees.

At MSR'16, Kikas, Dumas, and Pfahl (the KDP team) [8] built time-dependent models for issue close time prediction using Random Forests with a combination of static code features, and non-code features to predict issue close time with high performance. We regard this paper as the prior state-of-the-art in predicting issue lifetime. Our results, reported below, achieve similar performance figures to those of KDP, but we do so:

- *Using cross-project data.* This is an important extension to KDP since, as will show, sometimes projects have very bad local data about project issues. In those cases, our cross-project results would be preferred to the local-learning methods of KDP.
- *Using fewer features.* KDP argue that certain features are overall most important for predicting issue lifetime. Our results suggest that their conclusion on "most important feature" need some adjustment since the "most important" features different from project to project.
- *Using simpler learners.* KDP used Random Forests which can be too large for humans to read. Our approach, that combines feature selection with learning a single decision tree, leads to very small, easily-readable models.

While we adopt much of the framework of KDP, we do make some simplifications to their technique. Rather than build 28 *different* models for specific time periods in the issue's lifetime, we only build five. We do this since, in our experience, the simpler the modeling, the more the commercial acceptance.

III. Methods

A. Data

Figure 1 shows the ten projects used in this study. These projects were selected by our industrial partners since they use,

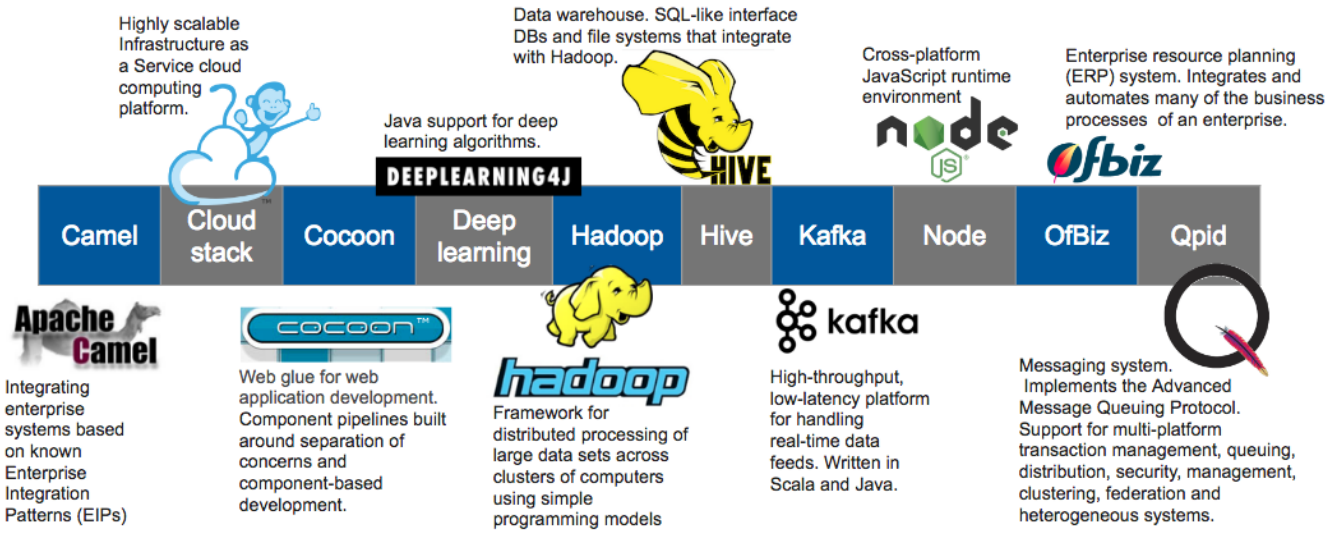

Fig. 1: The open-source projects that were used in this study. We extracted one issue lifetime dataset for each project listed above.

or extend, software from these projects. The raw data dumps came in the form of commit data, issues, and code contributors from GitHub and JIRA projects, and we extracted ten issue datasets from them with a minimum of 1,434 issues, maximum of 12,191, and median of 5,267 issues per dataset. We also created a combined dataset, which aggregated all instances from the ten datasets into one, for a total of eleven datasets used in the study.

One issue in preparing this data is that a small number of issues were *sticky*. A sticky issue is one which was not yet closed at the time of data collection. In the KDP paper, sticky issues were handled by approximating the close time to be a chosen set date in the future. KDP's method for doing that is innovative, but somewhat subjective. Hence, we simply omitted sticky issues from our train/test sets. It should be noted that for local learning (where training and test data come from the same source), it is unresolved as to whether or not our handling of *sticky* issues is to preferred to that of KDP. However, in the sequel, we will strongly recommend the use of cross-project data; i.e. experience from completely different projects will be used to build predictors. In that cross-project context, there is no need to complete the sticky issues using the KDP method since we are only training on the issues for which we had certain knowledge (rather than make use of KDPs educated guess approach)

In raw form, our data consisted of sets of JSON files for each repository, each file containing one type of data regarding the software repository (issues, commits, code contributors, changes to specific files), with one JSON file joining all of the data associated with a commit together: the issue associated with a commit, commit time, the magnitude of the commit, and other information. In order to extract data specific to issue close time, we did some preprocessing and feature extraction on the raw datasets pertaining to issue close time.

Our feature engineering is where we diverge primarily from KDP's study. We chose to make our model simpler and smaller with the hopes of making models easier to read and understand. Of the 21 features used by KDP, we first eliminated those from Table I if they:

- Were not available in our raw data.
- Could not calculated. For example, KDP do not specify how to calculate their *textScore* feature.
- Needed data generated after issue creation.

This brought our feature list down to just seven features, as detailed in Table II. Note that our target class was *timeOpen*.

B. Selecting Target Classes

In the issue lifetime prediction literature, there are several ways to handle the learning target:

- Bhattacharya and Neamtiu use multivariate regression to predict an exact fix time [1]. Note that such regression models output one continuous value.
- Both Panjer and Marks et al. build one classifier that predicts for N classes [10], [12].
- KDP, Zhang et. al., and Giger et. al. build one binary classifier for N goals [4], [8], [17]. This "N-binary" approach is standard practice when using support vector machines [16].

For two reasons, we adopt this "N-binary" approach. Firstly, as KDP is the current state-of-the-art, we adopt their approach. Accordingly, instead of asking what will be the issue close time, we discretized the issue report time into five lifetimes: *day, week, 2weeks, month,* and *3months*; then built five predictors for the following two-class problems:

- Predictor1: closes in 1 day vs more than 1 day;
- Predictor2: closes in 1 week vs more than one week;
- Predictor3: closes in 2 weeks vs more than 2 weeks;
- Predictor4: closes in 1 month vs more than 1 month;
- Predictor5: closes in 3 months vs more than 3 months.

Feature name	Feature description
nCommentsT	Number of comments the issue has received before the time of data collection
nActorsT	Number of unique persons who have commented, referenced or subscribed to the issue before the time of data collection
nAssignmentsT	Number of assignment events before the time of data collection
nLabelsT	Number of labels added before the time of data collection
nMentionedByT	Number of times issue was mentioned from other issues before the time of data collection
nReferencedByT	Number of times issue was mentioned in commit messages using the issue id, before time of data collection
nSubscribedByT	Number of persons subscribing to receive updates on the issue before time of data collection
meanCommentSizeT	Average comment size of the comments received before the time of data collection
textScore	Classification score obtained from cleaned issue title and content
nCommitsByActorsT	Total number of commits done by actors who committed code to the project repository during the period from two weeks before the issue creation to the time of data collection
nCommitsByUnique-ActorsT	Number of unique actors who committed code to the project repository during the period from two weeks before the issue creation to the time of data collection
nIssuesCreatedProjectT	Number of issues created in the project during the period of 2 weeks before the issue creation until the time of data collection
nIssuesCreatedProject-ClosedT	Number of issues created and closed in the project during the period of 2 weeks before the issue creation until the time of data collection
nCommitsProjectT	Number of commits in the project during the period of 2 weeks before the issue creation until the time of data collection

TABLE I: Features from KDP's study [8] that we did not use to build our models, either because they were time sensitive (indicated by the *T* suffix) or our feature selector did not select them.

Feature name	Feature Description
issueCleanedBodyLen	The number of words in the issue title and description. For JIRA issues, this is the number of words in the issue description and summary
nCommitsByCreator	Number of commits made by the creator of the issue in the 3 months before the issue was created
nCommitsInProject	Number of commits made in the project in the 3 months before the issue was created
nIssuesByCreator	Number of issues opened by the issue creator in the 3 months before the issue was opened
nIssuesByCreatorClosed	Number of issues opened by the issue creator that were closed in the 3 months before the issue was opened
nIssuesCreatedInProject	Number of issues opened in the project in the 3 months before the issue was opened
nIssuesCreatedIn-ProjectClosed	Number of issues in the project opened and closed in the 3 months before the issue was opened
timeOpen	Close time of the issue (target class).

TABLE II: Features used in this study (prior to feature selection). A *creator* is the person who opened the issue. A *project* is a software repository that has associated issue and commit data.

Our second reason for using "N-binary" classifier is that it fit the needs of our industrial partners. We work with a group of developers in Raleigh, NC that attend a monthly "open issues" report with their management. The sociology of that meeting is that the fewer the open issues, the less management will interfere with particular projects. In this context, developers are motivated to clear out as many issues as possible before that meeting. Therefore, the question these developers want to know is what issues might be closed well before that monthly meeting (i.e. can this issue be closed in one week or two?).

Generalizing from the experience with our industrial partners, we say that "N-binary" learning is appropriate when the local user population has "activity thresholds" where new activity is required if some measure reaches some threshold point. In such a context, the more general and harder question of "is it class X or Y or Z" can be replaced by the simpler and more specific question of "is it a result that crosses our thresholds?"

C. Class Re-Balancing

The *timeOpen* class distribution for each dataset is shown in Figure 2. Note that many issues were closed within a day or before 7 days, as well as between 365 and 1000 days, while very few issues were closed between 14 and 90 days.

For some of our datasets on certain time thresholds, the minority to majority class ratio was over 300:1, which created difficulty for the learners to train themselves effectively. To handle this problem of highly imbalanced class distributions, we tried applying SMOTE (Synthetic Minority Over-Sampling [2]) which an is oversampling technique for equalizing class distributions. The results of SMOTEing were inconclusive since no statistically significant difference was detected between our SMOTEd and non-SMOTEd results. Accordingly, this report makes no further mention of SMOTE.

D. Feature Subset Selection

Hall's CFS feature selector [7] was used to determine which features were most important.

Unlike some other feature selectors (e.g. Relief, InfoGain), CFS evalautes and hence ranks feature subsets rather than individual features. and hence CFS is based on the heuristic that "good feature subsets contain features highly correlated

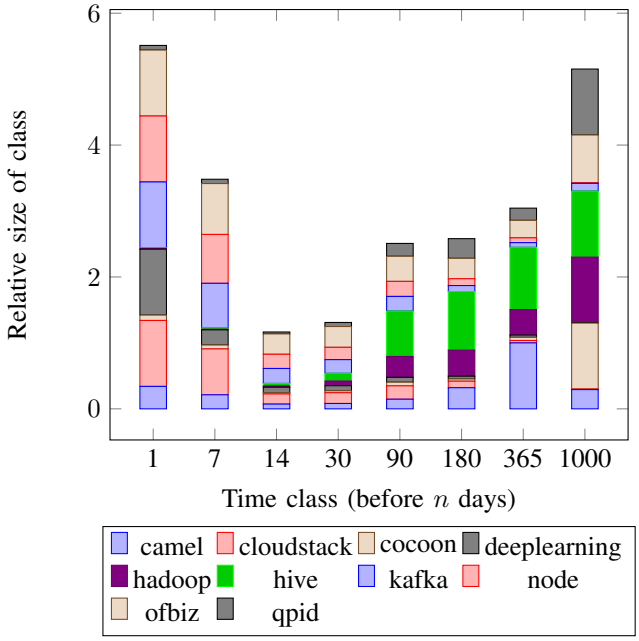

Fig. 2: Class distribution for the *timeOpen* feature for each of the ten issue lifetime datasets (plus one dataset combining all 10) used in the study. Many issues were closed between 0 and 1 day, and between 365 and 1000 days.

with the classification, yet uncorrelated to each other". Using this heuristic, CFS performs a best-first search (with a horizon of five[3]) to discover interesting sets of features
Hall et al. scores each feature subsets as follows:

$$merit_s = \frac{k\overline{r_{cf}}}{\sqrt{k + k(k-1)\overline{r_{ff}}}}$$

where:

- $merit_s$ is the value of some subset s of the features containing k features;
- r_{cf} is a score describing the connection of that feature set to the class;
- and r_{ff} is the mean score of the feature to feature connection between the items in s.

Note that for this to be maximal, r_{cf} must be large and r_{ff} must be small. That is, features have to connect more to the class than each other.

The above equation is actually Pearson's correlation where all variables have been standardized. To be applied for discrete class learning (as done by KDP and this paper), Hall et al. employ the Fayyad Irani discretizer [3] then apply the following entropy-based measure to infer r (the degree of associations between discrete sets X and Y):

$$r_{xy} = 2 \times \left[\frac{H(x) + H(y) - H(x,y)}{H(y) + H(x)}\right]$$

[3](1) The initial "frontier" is all sets containing one different feature. (2) The frontier of size n (initially $n = 1$) is sorted according to *merit* and the best item is grown to all sets of size *n+1* containing the best item from the last frontier. Go to step (3). Halt when last five frontiers have not seen an improvement in $merit$. On halt, return the best subset seen so far.

where H is the standard information gain measure used in decision tree learning.

E. Decision Tree Learning

Decision trees are standard classification models that use the concept of entropy in information theory to partition data into classes in a way that either minimizes entropy or maximizes homogeneity in each partition. Decision tree learners attempt to predict the value of the target variable (in this case, *timeOpen*) by recursively partitioning the data on features that most decrease the information entropy of each partition, until a stopping criterion is reached (such as number of instances in a partition being less than a chosen threshold).

Decision tree learners by nature are prone to overfitting the training set, usually when the stopping criterion is not aggressive enough, since a decision tree can be built to perfectly classify the training set. Overfitting can be avoided with a number of approaches, by setting a minimum number of instances per partition, which stops partitioning when the partition's size is less than the threshold size, or by pre- or post-pruning the tree, which is the process of replacing sub-trees with single leaf nodes when doing so doesn't significantly increase the error rate.

We used the C4.5 decision tree learner, using an aggressive pruning parameter M to stop partitioning the tree when the partition size was less than M, where we defined M as:

$$M = \frac{\text{number of instances in the dataset}}{25} \quad (1)$$

The "magic number" of 25 was set after a pre-study that tried values of 100, 50, 25, 12, 6, and 3, and reported no significant differences in performances between 25 and 3. Note that this definition of M means that no sub-tree of our data will ever be built from small samples of our data.

F. Model Generation and Evaluation Loop

We used the open-source data mining tool WEKA [6], developed at the Machine Learning Group at the University of Waikato, for all of our data mining operations, including dataset filtering, model generation, and model evaluation. We generated our models as follows:

- First, we first split each dataset into five datasets, one for each of our chosen prediction thresholds (1, 7, 14, 30, and 90 days).
- Next we applied correlation-based feature subset selection (CFS) on each data set to find what features are relevant to each data set.
- Finally, we built C4.5 decision trees on the resulting filtered and oversampled datasets. Note that, within WEKA, C4.5 is called J48 (short for "Java port of C4.5 release 8").

The resulting models were evaluated in one of two ways:

- Local learning with cross-val; i.e. performing stratified 10-fold cross validation where, 10 times, we test on 10% of the data while training on the other 90%.

Time threshold	Round Robin prec	recall	pf	KDP prec	recall	pf
1 day	.58	.19	.02	N/A	N/A	N/A
7 days	.66	.68	.12	.26	.82	.63
14 days	.71	.73	.13	.13	.80	.64
30 days	.77	.71	.12	.16	.80	.65
90 days	.78	.73	.17	.25	.79	.64

TABLE III: Comparison between the performances of our Round Robin approach and KDP's results. We achieve better precisions and false positive rates, but slightly lower recall.

- Cross-project learning with round-robin; i.e. start with $N = 10$ projects, for each project $p \in N$, train on $N - p$ then test on project p.

Note that, the round-robin studies were repeated for each target time period. For example, when trying to predict for "within 1 week" vs "more than one week" in project p, we only collected "within 1 week" vs "more than one week" data in the other $N - p$ data sets.

As shown below, in some cases the local learning generated poor results. For all those cases, much better results were seen using round-robin. Hence, for predicting issue lifetimes, we recommend the use of the round-robin approach.

G. Performance Measures

Precision, recall, and false alarms are three performance measures for binary classification problems, where a data point is classified as "selected" or "not selected" by the model:

$$prec = \frac{TP}{TP + FP},\ recall = \frac{TP}{TP + FN},\ pf = \frac{FP}{TN + FP}$$

Here, $\{TP,\ FP,\ TN,\ FN\}$ are the count of true positives, false positives, true negatives, false negatives, respectively.

IV. Results

This section offers results that support the claims made in the introduction:

- *Claim #1*: Our predictors had lower false alarms and higher precisions than KDP.
- *Claim #2*: Cross-project learning works remarkably effectively in this domain.
- *Claim #3*: Our predictors are easy to read & comprehend.

A. Claim #1: Better Precisions and False Alarms

Table III shows mean results averaged over all data sets for this paper and KDP. As can be seen:

- KDP has slightly better recall results (pd);
- Our methods have much higher precision (median 71%);
- Our methods have much lower false alarms (median 13%);

In our engineering judgement, our false alarm and precision results more than compensate with the slightly lower recalls.

Dataset	Time Class	Crossval (local learning) *prec*	*recall*	*pf*	Round Robin (cross-project) *prec*	*recall*	*pf*
camel	1	65	40	4	56	23	3
	7	74	70	7	66	70	12
	14	73	78	10	74	70	11
	30	82	80	7	77	74	12
	90	89	70	5	77	81	21
cloudstack	1	66	60	24	57	18	2
	7	76	93	77	65	66	11
	14	81	96	83	70	71	11
	30	85	100	100	78	65	9
	90	94	100	100	74	75	20
cocoon	1	0	0	0	60	17	2
	7	0	0	0	66	68	13
	14	0	0	0	71	75	13
	30	53	96	14	77	71	11
	90	67	95	11	82	64	12
deeplearning	1	78	77	41	51	17	3
	7	80	100	100	65	66	11
	14	86	100	100	70	73	12
	30	91	100	100	77	67	10
	90	96	100	100	76	77	19
hadoop	1	0	0	0	57	22	4
	7	0	0	0	66	70	18
	14	0	0	0	70	81	22
	30	0	0	0	76	80	20
	90	32	2	0	80	83	24
hive	1	0	0	0	61	15	2
	7	0	0	0	68	67	13
	14	52	35	1	73	71	13
	30	0	0	0	80	69	10
	90	62	53	9	81	76	16
kafka	1	63	48	17	58	16	2
	7	78	83	43	65	66	11
	14	81	90	56	69	73	12
	30	83	97	76	76	69	10
	90	91	98	71	81	62	11
node	1	56	31	16	60	21	2
	7	69	95	89	64	55	7
	14	76	100	100	68	55	7
	30	84	100	100	74	56	7
	90	93	100	100	69	69	18
ofbiz	1	0	0	0	63	21	2
	7	54	43	27	66	79	12
	14	56	70	57	73	78	10
	30	62	87	77	79	72	9
	90	67	100	100	83	64	9
qpid	1	0	0	0	60	16	2
	7	0	0	0	66	71	14
	14	0	0	0	70	78	16
	30	0	0	0	74	82	17
	90	53	19	5	79	75	18

TABLE IV: Median predictive performance of each model created. Each row corresponds to the performance statistics of a dataset split by a certain time threshold. Cells marked with **red** indicate "bad" results; i.e. false alarms over 33% or precision or recall results under 33%.

B. Claim #2: Cross-Project Learning Works Well

Table IV show median results for precision, recall, and false alarm seen in our local learning and round-robin experiments. Cells marked with **red** show "bad" results; i.e. very low precisions or low recalls or high false alarms. Three things to note about Table IV are:

- In many cases, the local results have many "bad" results. This result explains many of the results described in *Releated Work*; i.e. learning issue lifetimes is hard using just data rom one project.
- With one exception, the cross-project results are not "bad"; i.e. cross-project learning performs very well for lifetime prediction.
- The one exception is predicting for issues that close in 1 day. By its very nature, it is a challenging task since it

```
HADOOP-7:
nissuescreatedinprojectclosed <= 33     :7
nissuescreatedinprojectclosed > 33
|   nissuescreatedinprojectclosed <= 199
|   |   issuecleanedbodylen <= 27       :not7
|   |   issuecleanedbodylen > 27        :7
|   nissuescreatedinprojectclosed > 199 :not7

CAMEL-14:
nIssuesCreatedInProjectClosed <= 12 :14
nIssuesCreatedInProjectClosed > 12
|   nCommitsInProject <= 596
|   |   nCommitsInProject <= 524    :not14
|   |   nCommitsInProject > 524     :14
|   nCommitsInProject > 596         :not14
```

Fig. 3: Learned Decision Trees. These are nested if-then-else statements. For example, the last line of the the above is part of a branch saying the following kind of issue will not close in 14 days: *if nIssuesCreatedInProjectClosed is over 12 and if nCommitsInProject is over 596*.

Feature name \ Dataset name	node	hive	camel	kafka	ofbiz	combined	qpid	cloudstack	deeplearning	hadoop	cocoon	Total
nCommitsInProject	○	○	○	○	○			○	○	○		8
nIssuesCreatedInProjectClosed	○	○	○	○	○	○	○				○	8
nIssuesCreatedInProject	○	○	○	○		○	○	○		○		8
nIssuesByCreatorClosed	○	○		○	○		○					5
nCommitsByCreator	○	○	○			○						4
issueCleanedBodyLen	○				○				○			3
nIssuesByCreator	○											1
Total	7	5	4	4	4	3	3	2	2	2	1	

TABLE V: Features selected by CFS for each dataset are denoted with the ○ symbol. *nIssuesByCreator* appeared in only 1 dataset suggesting it is not a good predictor of issue close time, while *nCommitsInProject*, *nIssuesCreatedInProject*, and *nIssuesCreatedInProjectClosed* appeared in 8, suggesting they are clearly important in predicting issue close time.

relies on a very small window for data collection. Hence, it is not surprising that even our best round-robin learning scheme has "bad" performance for this hard task.

- Interestingly, just because local learning has problems with a data set does not mean that that cross-project learning will also be challenged. As shown in Table IV, cross-project learning works very well in nearly all cases.

The exception to this last point are the cross-project recalls for *node*. These are quite low: often 50% to 60%. That aid, the the precisions for cross-project *node* are respectable and the false alarms for cross-project *node* are far superior to the local learning *node* results.

C. Claim #3: Our Models Are Easy to Explain

Compared to KDP, our results are easier to explain to business users:

- Since we do use single-tree decision tree learning rather than Random Forests (as done by KDP), it is not necessary to browse across an ensemble of trees in a forest to understand our models.
- Since we use the early stopping rule of Equation 1, our trees are very small in size (median of 9 nodes in round-robin). Figure 3 shows two trees learned during the round-robin when predicting for issues that close in 7 or 14 days for *hadoop* or *camel*. Each tree is six lines long. No other tree learned in this study was more than 20 lines long.
- Since we use feature selection, our data miner has fewer features from which they can learn trees. Accordingly, our trees contain fewer concepts.

Further to the last point, our CFS tool selects very few features per project. The results of the CFS selection are shown in Table V. Not shown in that figure is *timeOpen* since that is the target class used by every dataset. In that figure, columns are sorted according to how many features were selected within a data set and rows are sorted according to how many data sets used an feature. That figure shows that:

- *node* uses all of the features defined in Table II.
- Most data sets use a small minority of the features: the median number is 3 and often it is much less; e.g. *cocoon* only uses one feature.
- The number of selected features is not associated with the success of the learning. For example, consider the datasets *hive* and *cocoon* which make use of many and few features (respectively). We can see in Table IV that both these data sets had poor cross-val results even though Table V shows that both these data sets used a different number of features.

From the above, we cannot say that any particular set of features is always "best" (since CFS selects different features for different data sets).

V. Threats to Validity

As with any empirical study, biases can affect the final results. Therefore, any conclusions made from this work must be considered with the following issues in mind.

Sampling bias threatens any classification experiment; i.e., what matters there may not be true here. For example, the data sets used were selected by our industrial partners (since they use and/or enhance these particular projects). Even though these data set covers a large scope of applications (see Figure 1), they are all open source systems and many of them are concerned with Big Data applications.

Learner bias: For building the defect predictors in this study, we elected to use a single decision tree learner. We chose the this learner since past experience showed it generates simple models and we were worried about how to explain our learned models to our industrial partners. That said, data mining is a large and active field and any single study can only use a small subset of the known classification algorithms.

Goal bias: We used the same "N-binaries" approach as used in the prior state-of-the-art in this work (the KDP paper) but whereas they built $N = 24$ different models, we only built $N = 5$ models. Given our current results, that decision seems justified but further work is required to check how many N time thresholds are most useful.

VI. Conclusions

Our results support several of the KDP conclusions. Firstly, we endorse their use of "N-binary" learners. Our explanation for some of the poor results seen in prior studies is that they were trying to make one model do too much (i.e. predict for too many classes). At least for predicting issue lifetime, it would seem that N learners each predicting "before/after" for a particular time threshold performs very well indeed.

Secondly, we endorse KDP's conclusion that it is best to use contextual features for predicting issue close time. All the results of this paper, including our excellent cross-project results, were achieved without reference to static code measures (except *issueCleanedBodyLen*, used in models for 3 projects). Further, just a handful of contextual features (see right-hand-side of Table V), are enough to predict issue lifetimes. Like KDP, these results say that issue lifetimes can be characterized by the temporal pattern of issues and commits associated with the team working those issues. Our models seem to act like a distance function that find the closest temporal pattern to that seen in some current issue. Once that closest pattern is found, we need only report the average close time for that group of issues and commit patterns.

That said, we recommend changing some aspects of the KDP analysis. As shown in Table III, using the following methods, we achieved lower false alarm rates and higher precisions than KDP:

- Prior to learning, use CFS to find the best features.
- Use simpler learners than Random Forests. As shown in Figure 3, we can produce very simple decision tree using C4.5, which are much simpler to show and explain to business users than Random Forests.
- The best way to learn issue lifetime predictors is to use data from other projects.

As evidence for the last point, the results of Table IV are very clear: round-robin learning (where the training data comes from other projects) does better than local learning. Recall that in that table, we had many data sets where the local learning failed spectacularly (*cocoon, hadoop, hive, kafka, node, ofbiz, qpid*). Yet in each of those cases, the cross-project results offered effective predictions for issue lifetime.

To end this paper, we repeat a note offered in the introduction. The success of cross-project learning for lifetime prediction was quite unexpected. These results were achieved without the data adjustments recommended in the defect prediction transfer learning literature [9], [11], [15]. That is, while transfer learning for some software analytics tasks can be complex, we have discovered that there exist some tasks such as predicting issue lifetimes where it is relatively simple. Perhaps, in 2017, it is timely to revisit old conclusions about what tasks work best for what domains.

Acknowledgments

This work is funded by the National Science Foundation under REU Grant No. 1559593 and via a faculty gift from the DevOps Analytics team in the Cloud Division of IBM, Research Triangle Park, North Carolina.

References

[1] Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 207–210, New York, NY, USA, 2011. ACM.

[2] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[3] U M Fayyad and I H Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[4] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM.

[5] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 495–504, May 2010.

[6] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[7] Mark A Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[8] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using dynamic and contextual features to predict issue lifetime in github projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 291–302, New York, NY, USA, 2016. ACM.

[9] Ekrem Kocaguneli, Tim Menzies, and Emilia Mendes. Transfer learning in effort estimation. *Empirical Software Engineering*, 20(3):813–843, 2015.

[10] Lionel Marks, Ying Zou, and Ahmed E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 11:1–11:8, New York, NY, USA, 2011. ACM.

[11] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proceedings of The European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*, 2015.

[12] Lucas D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 29–, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[14] Mitch Rees-Jones, Matt Martin, and Tim Menzies. Replication package, available for download. Issue close time: datasets + prediction classifiers, December 2016. DOI: 10.5281/zenodo.197111; URL: https://doi.org/10.5281/zenodo.197111.

[15] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[16] Zhe Yu, Nicholas A Kraft, and Tim Menzies. How to read less: Better machine assisted reading methods for systematic literature reviews. *arXiv preprint arXiv:1612.03224*, 2016.

[17] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: An empirical study of commercial software projects. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1042–1051, Piscataway, NJ, USA, 2013. IEEE Press.