Enhanced Issue Lifetime Prediction Using Contextual Features

Matthew Martin
Colby College
Waterville, Maine USA
Email: mjmartin@colby.edu

Mitch Rees-Jones
North Carolina State University
Raleigh, NC USA
Email: mwreesjo@ncsu.edu

Tim Menzies
North Carolina State University
Raleigh, NC USA
Email: 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. Previous studies on issue lifetime prediction present models that use attributes not known at issue creation time, making it a less viable predictor for newer issues. Further, previous studies use many models and do not attain high levels of accuracy.**

In this work, we present a prediction model which uses only attributes known at creation time and achieves high accuracy. The results show that issue close time can be predicted with fewer features and higher accuracy than the state-of-the-art, while only using features known at issue creation time.

Keywords*—*Issue lifetime prediction, issue tracking, effort estimation, prediction.

I. Introduction

During the software development lifecycle, accurately predicting the time required for an issue to close (here, we define an *issue* as a bug report, and the amount of time required to mark it as closed as *issue close time*) has multiple benefits for the developers, managers, and stakeholders involved in a software project. Predicting issue lifetime 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 be able to predict issue close time specifically when the issue is created. An immediate prediction can be used, for example, to auto-categorize the issue or send a notification if it is predicted to be an easy fix.

Issue lifetime prediction has gained significant interest in academia in recent years. Researchers have employed a number of techniques, including Markov chains and kNN-based models [11], and simple logistic regression [10] [7], to build predictive models with varying degrees of success. Recently, Kikas, Dumas, and Pfahl (hereafter *KDP*) produced high-performance models in issue lifetime prediction by taking a contextual approach to the feature selection process. KDP used time-dependent features, such as number of comments on an issue, and contextual features, such as commit frequency for the software project, to produce the highest-accuracy issue close time model to date. Still, current state-of-the-art prediction models are neither stable nor accurate enough for practical industry use. If the performance of these prediction models can be improved to a high enough degree of accuracy and stability across software projects, then they can be implemented in issue tracking systems and large-scale benefits can be reaped by users of the systems.

Current approaches to issue close time prediction include using textual similarity to other issues, using combinations of dynamic and contextual features, and building many models each trained for a specific software project. Accuracy metrics of these approaches suggest that issue lifetime prediction is a challenging task, and is even more challenging for issues in open source projects.

They key component of our approach is using contextual features such as measurements of project activity (frequency of commits, number of issues created and closed, etc.) Using contextual features makes the prediction a measurement of absolute time rather than effort, but this is still a relevant and useful metric. In this study, we build prediction models via decision tree learners to predict issue close time, using contextual features. We also investigate research questions in issue lifetime prediction.

Our resulting models achieve an F1 accuracy measure about 20 percent higher on average than the current state-of-the-art, using fewer and simpler models and fewer features.

We investigated the following research questions during this study:

- **RQ1: Can we accurately predict issue close time with fewer total attributes and no time-dependent attributes?** To answer this question, we build decision trees to predict issue close time with no time-dependent attributes, while maintaining high predictive performance. We also build decision trees without specifically excluding time-dependent attributes to compare performance.
- **RQ2: Are there attributes that are especially strong predictors of issue close time?** To answer this research question, we analyze our own datasets as well as the similarities and discrepancies of related studies. For our own datasets, we quantify the strength of each attribute for each dataset with the CFS feature subset selection algorithm. (todo - also use other methods) For related work, we show that each software project finds different features to be more important.
- **RQ3: Does a general model trained for many software projects perform better than many**

individually-trained models? To answer this question, we use the same process to build a model, but we train and test it on all datsets rather than an individual one. We then compare the performance of the general model to the performance of the individuals to conclude if there is a significant difference.

- **RQ4: Does the choice of time classes significantly affect performance of issue close time models?** Issue close time prediction is typically implemented with a classifier that predicts an issue will close within some time range. We investigate if the choice of time ranges can significantly affect the performance of our classifier by choosing multiple time ranges to classify issues.

The rest of the paper is organized as follows. In the *Background* section, we more formally introduce the motivation behind issue lifetime prediction and prior research work. In the *Data Collection* section, we describe the source and character of our data sources and the pre-processing required to get the data into a format with the features we were interested in. In the *Methods* section, we present the methods used to build our prediction models and in the *Results* section we present the experimental results of the models and evaluate their performance. We answer our research questions and address threats to validity in the *Discussion* section, and make final statements on the outcome of the study in the *Conclusions* section.

II. Background

A. Motivation

Mention Company A's and Company B's interest in our models.

B. Related Work

Recently, there has been significant interest in issue lifetime prediction models. Recent research efforts have focused on building high-accuracy models to predict issue close time, studying the methodologies used to generate such prediction models, and quantifying which issue features are most predictive across software projects and how stable they are.

Panjer [10] explores the viability of predicting the time to fix a bug given only basic information 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 [5] performed used decision tree learning to make prediction models for issue close time for Eclipse, Gnome, and Mozilla bugs with 65% precision. He divided his time classes into thresholds of 1 day, 3 days, 1 week, 2 weeks, and 1 month.

Bhattacharya and Neamtiu [1] study 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%. They 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. They find 3 open research questions: (1) assessing whether bug-opener's reputation is useful for prioritizing bugs, (2) which attributes are useful in bug prediction time, and (3) finding prediction models for bug fix time.

Guo, Zimmerman, Nagappan, and Murphy [7] 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 (more likely to get fixed). Bugs are also more likely to get fixed if the bug fixer is in the same team or in geographical proximity. 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. XXX raised possibiltiy that social ¿ product (a thesis also endorsed by https://goo.gl/7IPYw5

Marks, Zou, and Hassan [9] found that the most important factors in Mozilla were time of filing and location, but severity for Eclipse, but priority wasn't an important factor for either. Also did some analysis on effort vs. calendar time. XXX still low values

Zhang, Gong, and Versteeg [11] perform an empirical study of bug-fixing time - proposing 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, and a kNN-based classification model for predicting a bug's time horizon by which it will be fixed.

KDP summary here.

KDP. model not readable. random forests. an important attribute used that study not defined (textscore). textscore comes from such an elaborate computation that we need to ask if it some other option is better. Also, they split their train and test data on a chosen temporal split rather than using k-fold cross validation. While this strategy matches how issue prediction models are trained and used in real-world scenarios, they do not allow for the extensive validation that comes with cross validation.

In summary, the two state-of-the-art papers in the are of issue lifetime prediction are Zhang, et al. [11] and KDP [8]. We therefore set out to try and reproduce their results (since state-of-the-art papers should always be assessed by other authors). In that process, we found several areas that could be improved.

- Zhang, et al. [11] used a non-standard learner and we commend the authors for making that exploration. But the use the non-standard tools does compromise the widespread usability of the method.
- Both Zhang, et al. and KDP reported on what attributes were most important using a single-attribute feature selector (e.g. InfoGain or mean decrease in impurity). *Multiple feature selectors* such as the CFS method used in this study can select sets of features that, in conjunction, are useful.
- From our experience, the mathematical formalisms used in Zhang, et al. are difficult for some users to understand. Therefore, we would prefer to use human-readable methods; e.g. data miners that learn single trees.

- One of our biggest hesitations with KDP's work is that we believe their models are "too smart". By this, we mean they rely heavily on data generated after issue creation, and then tailor-make 28 *different* models for specific time periods in the issue's lifetime. This is excessive, and an unrealistic model for real software development teams to use practically.

In summary, prior results have demonstrated that the possibility of reasoning about issue lifetimes is possible, but several open issues remain. First, the predictive power of the models across studies is not consistently high. Models should report high enough accuracy across many studies to be feasible in industry. Second, decision trees have significant practical use for real-world application due to their human-readable nature, but the Zhang et al. result reported negatively on their use in issue classification problems. To solve this, we apply newer methods in software analytics to improve the performance of decision trees; specifically SMOTE and CFS. (todo: can we say "newer" when SMOTE and CFS are from 2000? according to the papers I found)

III. Data collection

The data used in this study was derived from two large industry leading companies. Company A provided data regarding several open source projects that they orchestrate, and Company B offered data regarding several of their proprietary 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 47,515, and median of 5,475 issues per dataset.

A. Feature selection

In raw form, the datasets consisted of separate sets of JSON files for each software 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 provided by Companies A and B to extract features pertaining to issue close time. We were able to extract ten datasets and used them to produce our prediction models.

Our feature selection is where we diverge primarily from KDP's study. We chose to make our model simpler and smaller with the hopes of maintaining high performance while making the models easier to replicate and use in practical scenarios. Of the 21 features used by KDP, we first eliminated those that either were not supported by our data or involved unclear methods of calculation (*textscore*). This brought the feature list to 18. Then, in order to address our primary concern with KDP's work - that their features were "too smart" - we eliminated any features that relied on data generated after issue creation. This brought our feature list down to just seven features.

Another key difference between our datasets and KDP's is we excluded *sticky issues* from our analysis. 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, whereas we did not include them when building our datasets. We concluded that their method of dealing with *sticky issues* was presumptuous and imperfect, and simply eliminating these issues was the cleanest approach.

TABLE I. Descriptions for each feature extracted for the datasets.

Feature name	Feature Description
issueCleanedBodyLen	The number of words in the issue ti
nCommitsByCreator	Number of commits made by the cr
nCommitsInProject	Number of commits made in the pro
nIssuesByCreator	Number of issues opened by the issu
nIssuesByCreatorClosed	Number of issues opened by the issu
nIssuesCreatedInProject	Number of issues opened in the proj
nIssuesCreatedInProjectClosed	Number of issues in the project oper
timeOpen {*1,7,14,30,90,180,365,1000*}	Close time of the issue. For example

B. Feature descriptions

To understand the attributes related to issue close time which we extracted, the following terms are defined. A *creator* is the person who created the issue. A *project* is a software repository which has associated issue and commit data.

Our decision trees classify the $timeOpen$ attribute. The value of the attribute indicates that the issue closed before that amount of time, but after the closest smaller time class. For example, an issue with a $timeOpen$ class of 30 closed in at or fewer than 30 days, but more than 14 days.

The features we extracted are described in Table II.

We wrote scripts to transform the data from the raw form to an pcrarff format containing the seven features we selected. Then, we were able to use those 10 datasets to build our models.

C. Close time distribution

The datasets had the following distribution of class frequencies shown in Figure 2. Many issues were closed within a day or before 7 days, as well as between 365 and 1000 days.

IV. Data collection

The data used in this study was derived from two large industry leading companies. Company A provided data regarding several open source projects that they contribute to, and Company B offered data regarding several of their proprietary 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 47,515, and median of 5,475 issues per dataset.

A. Feature selection

In raw form, the datasets consisted of separate sets of JSON files for each software 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 provided by Companies A and B to extract features pertaining to issue close time. We were able to extract ten issue close time datasets and used them to produce our prediction models.

The features we extracted from the raw datasets were those used in KDP's study. We chose these features because they were comprehensive, they produced relatively high-performance models as shown in the KDP study, and our data was complete enough to extract all of the features described by KDP.

One key difference between our datasets and KDP's is we excluded *sticky issues* from our analysis. 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, whereas we did not include them when building our datasets. We concluded that

B. Feature descriptions

To understand the attributes related to issue close time which we extracted, the following terms are defined. An *actor* is a person who has interacted with an issue - by creating, closing, commenting, referencing, or being assigned to the issue. A *creator* is the person who created the issue. A *project* is a software repository which has associated issue and commit data. A *label* is a tag assigned to an issue - such as "bug", "feature", or "duplicate". An attribute's dependence on time is indicated with a t at the end of the attribute name. The value of each time-dependent attribute is calculated at the time of observation, i.e. when the data was collected. For example, $nCommentsT$ is calculated as the number of comments on the issue when the data was collected.

Our decision trees classify the $timeOpen$ attribute. The value of the attribute indicates that the issue closed before that amount of time, but after the closest smaller time class. For example, an issue with a $timeOpen$ class of 30 closed in at or fewer than 30 days, but more than 14 days.

The features we extracted are described in Table II.

We wrote scripts to transform the data from the raw form to an pcrarff format containing the 18 features we selected. Then, we were able to use those 10 datasets to build our models.

C. Close time distribution

The datasets had the following distribution of class frequencies shown in Figure 2. Many issues were closed within a day or before 7 days, as well as between 365 and 1000 days.

V. Methods

We used the open source data mining tool WEKA [citation] for our data mining operations. It is available for use both as a GUI and as a Java API accessible via the command line or as a library. We ran preprocessing algorithms on the datasets to balance the minority classes of time-to-close, and used a C4.5 decision tree learner to build and validate decision trees models. We used 10-fold cross validation to train and test the models.

TABLE II. Descriptions for each feature extracted for the datasets.

Feature name	Feature Description
issueCleanedBodyLen	The number of words in the issue ti
meanCommentSizeT	Average number of words per comm
nActorsT	Number of actors on the issue at tim
nCommentsT	Number of comments on the issue a
nCommitsByActorsT	Number of commits made by actors
nCommitsByCreator	Number of commits made by the cr
nCommitsByUniqueActorsT	Number of unique commiters in the
nCommitsInProject	Number of commits made in the pro
nCommitsProjectT	Number of commits made in the pro
nIssuesByCreator	Number of issues opened by the issu
nIssuesByCreatorClosed	Number of issues opened by the issu
nIssuesCreatedInProject	Number of issues opened in the proj
nIssuesCreatedInProjectClosed	Number of issues in the project oper
nIssuesCreatedProjectClosedT	Number of issues in the project oper
nIssuesCreatedProjectT	Number of issues in the project oper
nLabelsT	Number of labels assigned to the iss
nSubscribedByT	Number of actors subscribed to upda
timeOpen {*1,7,14,30,90,180,365,1000*}	Close time of the issue. For example

A. Pre-processing

Our datasets required some pre-processing to generate the most performant models. It has been shown that preprocessing datasets improves models trained on those datasets [3], employing methods such as oversampling or removing features that don't have high predictive value.

We first ran each dataset through the CFS feature subset selection algorithm, then filtered the datasets to include only the features the CFS algorithm found to be most useful for prediction. An example is shown in Figure **??**

We then split each dataset into 4 separate datasets, each of which had issues classified as closing either before or after a time threshold corresponding to that dataset.

We then ran each dataset through SpreadSubsample and SMOTE.

A common problem across the datasets was one of imbalanced classes. This is a common occurrence in real world datasets and can have detrimental effects on classifier performance [6]. In our datasets, many issues tended to close almost immediately or remain open for a long period of time, as shown in Figure 2. In decision tree learners such as C4.5, the learner we used to generate our models, performance and stability issues caused by class imbalances can be mitigated by under- and oversampling techniques. We chose the SMOTE algorithm (Synthetic Minority Over-Sampling), an oversampling technique for class imbalance problems shown by Chawla [2] to be an effective method of improving performance in C4.5 decision trees. It should be noted that arguments have been made in favor of undersampling and in opposition to oversampling techniques for mitigating C4.5 decision tree class imbalances [4].

The SMOTE algorithm operates by creating synthetic data points in the minority class of a dataset based on existing data

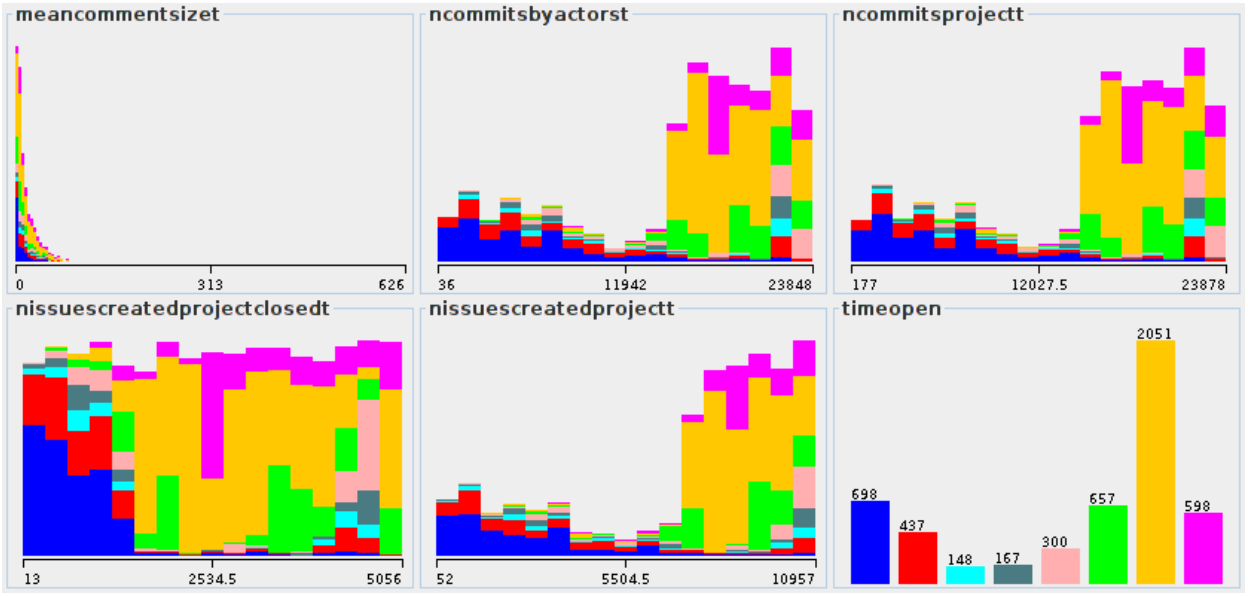

Fig. 1. A visualization of the distribution of time classes for each attribute selected by CFS. Each graph corresponds to an attribute, with the x axis representing the range of values the attribute takes and the y axis representing the distribution of each issue over that value, colored by time class.

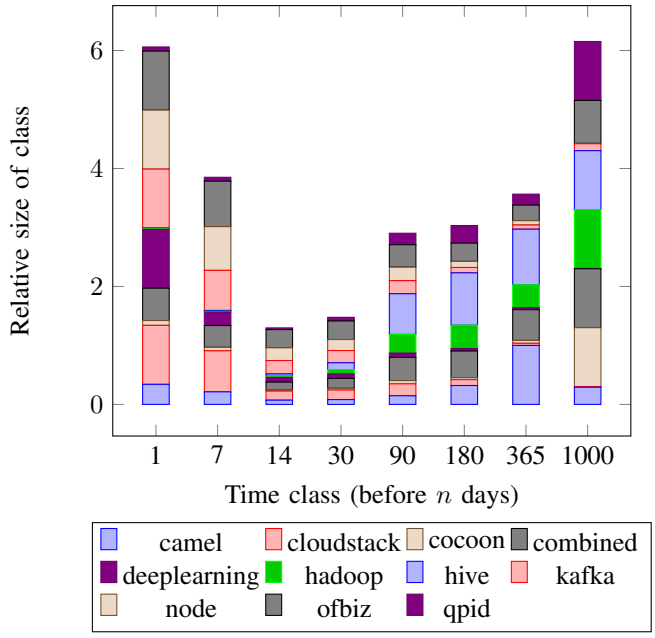

Fig. 2. Frequencies of each time class in the ten issue lifetime datasets used in the study. Many issues were closed between 0 and 1 day, and between 365 and 1000 days.

points in the minority class. This creation process is done until the desired balance is achieved - for example, to achieve a 200% increase in the size of the minority class, two synthetic data points must be created for each existing data point in the dataset. SMOTE picks a sample of instances in the minority class to generate synthetic data points for. Then, for each instance I (a feature vector) in the set, k nearest neighbors of the instance (usually 5) are calculated, where "nearness" is defined as Euclidean distance d between two instances a and b in the feature space with n features, where x_i denotes the ith feature in x:

$$\mathrm{d}(\mathbf{p}, \mathbf{0}) = \sqrt{\sum_{i=0}^{n} (a_i - b_i)^2}$$

To create a synthetic data point, one of the five nearest neighbors nn (also a feature vector) is randomly chosen, a random number $r \in [0, 1]$ and a synthetic data point $p = I + r \cdot (nn - I)$ is generated as a random point on the line segment between I and nn. This process of creating data points is repeated as needed to re-balance the minority class to a reasonable degree.

Our usage of SMOTE was dependent on the nature of the class imbalances in each dataset. Some datasets were not highly imbalanced and models on those datasets performed well, so we conditionally SMOTEd our datasets if the ratio of majority class to minority class was at least $4:1$.

B. Decision tree learning

We built our models in the form of decision trees. Decision trees are standard classification models that use the information theory concept of entropy to partition data into classes in a way that minimizes entropy, i.e. maximizes homogeneity in each partition. Decision tree learners attempt to predict the value of a target variable (in this case, issue close time) 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 are prone to overfitting the training set, especially when the stopping criterion is not aggressive enough. This is usually avoided with x approaches, including setting a minimum number of instances per partition, which stops partitioning when the partition's size is less than the

threshold, and a pruning confidence threshold, that prunes only when blah or stops pruning when blah.

We used the C4.5 decision tree learner using an aggressive pruning parameter M, which we defined to be:

$$M = \frac{\text{number of instances in the dataset}}{25}$$

C. Model performance and stability improvements

We tried changing the M parameter, but found little change in model performance.

We tried SMOTEing only if the model was unbalanced, for various thresholds for "unbalanced", and again found little change in model performance.

We trained and tested on one big combined dataset (todo) and found XXXX improvement in model performance.

We trained and tested on different time classes, and found XXXX improvement in model performance.

D. Model evaluation

We evaluated our models using three common performance statistics: precision, recall, and the F1 metric. We also used 10-fold cross validation to validate our models, a technique not employed by some prior results [citations].

Precision and recall are two common performance measures for binary classification problems, where a data point is classified as "selected" or "not selected" by the model. Let TP = Number of true positives and FP = Number of false positives. Precision is the fraction of those data points that were correctly selected, i.e. the fraction:

$$prec = \frac{TP}{TP + FP},$$

and recall is the the fraction of true positives that were actually selected:

$$rec = \frac{TP}{TP + FN}.$$

VI. Results

Results show that for the software projects we used in this study, we can accurately predict issue close time with practical time classifications.

A. All datasets combined

We generated models using the same methods on all datasets combined into one and measured its performance. We achieved a median F1 score of .735 with an IQR of .198, comparable to the average F1 scores of the individual datasets' models. Precision and recall had medians .773 and .743 respectively, which is also similar to the average medians of the individual datasets' models. The results are shown in Table [**?**].

TABLE III. Performance for all datasets combined

	Median	IQR
Precision	0.773	0.097
Recall	0.743	0.289
F1	0.735	0.198

B. Feature selection with CFS

The importance of features in issue lifetime prediction is an active research question with differing results [citations]. To answer this question for our own datasets, we ran CFS on each of the datasets to determine which features were most important. We deemed the feature to be important for the dataset if CFS measured the foobar performance to be at least 50%. The results of the analysis are shown in Table [**?**].

VII. Discussion

In this section, we discuss the performance of our results and answer the research questions presented in the introduction.

A. Research Questions

RQ1: Can we accurately predict issue close time with fewer total and no time-dependent attributes?

Result 1
We created models with an average F1 score averaging 20% higher than KDP's model, both with and without time-dependent attributes.

RQ2: Are there attributes that are especially strong predictors of issue close time?

Result 2
We found that CFS feature subset selection yielded no features that had unanimous predictive power across all datasets as shown in Table 1.

RQ3: Does a general model trained for many software projects perform better than many individually-trained models?

Result 3
We found that a general model trained and tested on all of our datasets had similar median precision, recall, and F1 scores compared to the individual models, but with lower variance.

- **RQ4: Does the choice of time classes significantly affect performance of issue close time models?**

Result 4
Need to do this

B. Threats to Validity

- We used a small number of datasets (10). Using a larger number of datasets, especially from more

Feature name \ Dataset name	camel	cloudstack	cocoon	combined	deeplearning	hadoop	hive	kafka	node	ofbiz	qpid	Total
issueCleanedBodyLen					×							1
meanCommentSizeT	×	×			×			×	×			5
nActorsT		×			×			×	×	×	×	6
nCommentsT		×	×	×	×			×	×	×		7
nCommitsByActorsT	×	×	×	×			×	×			×	7
nCommitsByCreator				×			×					2
nCommitsByUniqueActorsT		×		×								2
nCommitsInProject		×										1
nCommitsProjectT	×							×		×	×	4
nIssuesByCreator		×							×			2
nIssuesByCreatorClosed								×			×	2
nIssuesCreatedInProject												0
nIssuesCreateInProjectClosed			×	×				×				3
nIssuesCreatedProjectClosedT	×	×				×	×				×	5
nIssuesCreatedProjectT	×			×		×	×				×	5
nLabelsT					×				×			2
nSubscribedByT		×						×	×		×	4

Fig. 3. CFS classification of issue features. In this table, a feature is denoted as highly predictive with the × symbol, if CFS measured it to be at least 50%. todo: reword that

diverse origins than two companies, may yield more information regarding the stability of our models across different software projects.

- We discarded all issues that were marked as open at the time of data collection to avoid arbitrarily assigning a close date.
- We do not analyze differences in issue prediction for open- and closed-source software projects, or other types of software projects whose social contexts may affect a model's predictive performance.

VIII. Conclusion

Acknowledgment

The authors would like to thank...

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. C4. 5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure. In *Proceedings of the ICML*, volume 3, 2003.

[3] Sven F Crone, Stefan Lessmann, and Robert Stahlbock. The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing. *European Journal of Operational Research*, 173(3):781–800, 2006.

[4] Chris Drummond, Robert C Holte, et al. C4. 5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on learning from imbalanced datasets II*, volume 11. Citeseer, 2003.

[5] 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.

[6] Q. Gu, Z. Cai, L. Zhu, and B. Huang. Data mining on imbalanced data sets. In *2008 International Conference on Advanced Computer Theory and Engineering*, pages 1020–1024, Dec 2008.

[7] 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.

[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] 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.

[10] 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.

[11] 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.