Using Dynamic and Contextual Features to Predict Issue Lifetime in GitHub Projects

Riivo Kikas
University of Tartu
riivokik@ut.ee

Marlon Dumas
University of Tartu
marlon.dumas@ut.ee

Dietmar Pfahl
University of Tartu
dietmar.pfahl@ut.ee

ABSTRACT

Methods for predicting issue lifetime can help software project managers to prioritize issues and allocate resources accordingly. Previous studies on issue lifetime prediction have focused on models built from static features, meaning features calculated at one snapshot of the issue's lifetime based on data associated to the issue itself. However, during its lifetime, an issue typically receives comments from various stakeholders, which may carry valuable insights into its perceived priority and difficulty and may thus be exploited to update lifetime predictions. Moreover, the lifetime of an issue depends not only on characteristics of the issue itself, but also on the state of the project as a whole. Hence, issue lifetime prediction may benefit from taking into account features capturing the issue's context (contextual features). In this work, we analyze issues from more than 4000 GitHub projects and build models to predict, at different points in an issue's lifetime, whether or not the issue will close within a given calendric period, by combining static, dynamic and contextual features. The results show that dynamic and contextual features complement the predictive power of static ones, particularly for long-term predictions.

CCS Concepts

•**Software and its engineering** → *Software creation and management; Maintaining software; Open source model;*

Keywords

issue lifetime prediction, issue tracking, mining software repositories

1. INTRODUCTION

Open source projects usually rely on publicly accessible issue tracking systems to manage unresolved bugs and development tasks. In contemporary open source code hosting sites, such as GitHub and Bitbucket, the barrier for contributing issue reports is minimal. Entering a new issue in GitHub only requires two fields – title and textual description. Everyone can create an issue, but only a limited set of stakeholders actually deal with these issues. This tension between the ease of creating issues and the limited resources of the core project team leads to situations where issues receive sparse attention from the project team or do not even receive a preliminary screening upon their creation.

In a recent study [17], we showed that a considerable proportion of issues in GitHub issue trackers are left open for several months or even over a year. Yet, knowing when an issue will be closed is important from two viewpoints. First, it has been found that timeliness is an important determinant of contributor engagement and community contribution acceptance in GitHub [12]. If there is high uncertainty regarding the timeframe when the development team will address a given issue, the stakeholder who submitted it might be discouraged from making further contributions or even from using the software product. Having an estimate of issue closing time can help to reduce this uncertainty and provide greater transparency to all stakeholders. Second, an estimate of issue closing time provides core team members with a basis to prioritize their efforts and plan their contributions. In this respect, a recent study of long lived bugs in different projects [22] observes that over 90% of such bugs impact user experience and that automatic prioritization and assignment can minimize the impact of such bugs on end users. It is also observed that in some cases, bugs can be resolved earlier thanks to automatic prioritization and assignment.

In this setting, this paper addresses the problem of predicting, at a given time point during an issue's lifetime, whether or not the issue in question will close after a given time horizon, e.g. predicting if an issue that has been open for one week will remain open one month after its creation. The general problem of issue (or bug) lifetime prediction has received significant attention in the research literature. The focus of this study differs from previous work in four respects. First, the bulk of previous work has focused on analyzing a small number of hand-picked projects. In contrast, this paper studies this prediction problem based on a large sample of projects hosted in GitHub. Second, most previous work has focused on exploiting static features, i.e. characteristics extracted for a given snapshot of an issue – typically issue creation time. In contrast, the present study combines static features available at issue creation time, with dynamic features, i.e. features that evolve throughout an issue's lifetime. Third, previous approaches focus on predicting lifetime based on characteristics of the issue itself. In contrast,

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901751

the present study combines characteristics of the issue itself with contextual information, such as the overall state of the project or recent development activity in the project. Finally, most previous studies do not employ temporal splits to construct prediction models. In other words, models are trained on future data and then evaluated on past data. In this study, we construct models predictively using strict temporal splits such that predictions are always made based only on past data, which reflects how such predictive models would be used in practice.

Within the scope of the problem of issue lifetime prediction, this study seeks to answer the following specific research questions:

- RQ1: What level of accuracy is achieved by classification models trained to predict issue lifetime at different calendric time points in an issue's lifetime and for different calendric periods (day, week, month, quarter, semester and year) using both static and dynamic features of an issue as well as contextual features?
- RQ2: What features are most important when predicting issue lifetime?

The rest of the paper is organized as follows. In Section 2 we discuss previous work on issue lifetime prediction. In Section 3 we describe the dataset and analyze its characteristics, particularly with respect to issue lifetime. In Section 4 we present the features and methods employed to construct classification models for issue lifetime prediction. Next, we present the experimental results in Section 5. We finish with a discussion of the findings in Section 6 and conclusions and directions for future work in Section 7.

2. RELATED WORK

Issue lifetime prediction models have received significant attention in the literature. Several previous studies explicitly deal with problem of predicting how long it will take to close a given issue report.

Weiss et al. [28] predict issue resolution time for the JBoss project. Their approach enables early prediction by finding a set of textually similar issues for a newly entered issue and using this set to make a prediction of closing time. Their estimated resolution times deviate on average by 7 hours from the actual resolution time, and half of the estimations are in the +/-50% range of the original issue lifetime. This study shows the feasibility of predicting issues based on models trained for one specific project, where the project in question has a large set of issues that share common patterns.

Similarly, Giger et al. [9] predict bug fix times for Mozilla, Eclipse and Gnome projects. Their approach consists of extracting features for each bug report and training a decision tree model to predict if the fix time will be lower or higher than the project median fix time. They also experiment with dynamic features calculated at different points during a bug report lifetime, such as number of comments an issue has received and number of actions performed on an issue. They conclude that the use of dynamic features improves accuracy. Their reported Area Under the ROC Curve (AUC) scores fall in the range [0.65..0.83]. A shortcoming of their work is that it does not apply temporal splits to separate training and testing data – hence the models may use data 'from the future' to predict issue lifetime at a particular point in time.

Panjer [21] predicts resolution time for Eclipse bugs using a set of static features and a machine learning approach. He divides the bug lifetime distribution into seven ranges and uses these as classes. Using different classifiers, around 30% of issues are classified correctly. Cross validation is used for evaluation of the classifiers, but no temporal split is used to segregate training and test data. No dynamic features are used.

Francis & Williams [8] study the prevalence of long living bugs in an open-source Apache HTTP server and a closed source private project. They train a decision tree model to predict whether an issue will be closed by the time when 85%, 90% and 95% of issues are closed. Their model achieve F-scores in the range [0.63..095] for the closed-source project and [0.21..0.59] for the Apache open source project, suggesting that accurate issue lifetime prediction is more difficult in the context of open source projects.

Besides issue lifetime prediction, there are other related lines of research that seek to predict some attribute or future action on an issue report. Examples are predicting whether an issue will be fixed [13], delayed [6], will be reopened [30, 29, 23], as well as estimating its priority [24], assignment or category [32, 14, 1]. In general, these studies use the same general approach: extract features from issue reports; train a machine learning model; and evaluate the model. A similar framework has also been used for predicting when contributed code patch or pull request will be accepted [11, 15, 31].

Table 1 summarizes the above review of related work in terms of six attributes that delimit the scope of the present study. Specifically, for each referenced study, the table indicates: (i) if the study relies on a large dataset – where "large" is defined as encompassing more than 6 projects (cf. column LD); (ii) whether or not the study relies on dynamic features in addition to static ones (column DF); (iii) whether or not the study in question relies on contextual features about the surrounding project in addition to features extracted from individual features (column CF); (iv) whether or not the study applies temporal splits to separate training data from testing data (column PT); (v) whether the constructed models can be used for predicting lifetime at issue creation time and during an issue's lifetime (column MA);[1] and (vi) whether the study addresses the problem of issue lifetime prediction (IL), acceptance time of a contribution (CL), or other issue attribute (IA) – cf. column TYPE.

We note that only Giger et al. [9] rely on dynamic features and make issue lifetime predictions at different time points during the issue's lifetime. Also, only Ye et al. [31] and Gousios et al. [11] rely on a large dataset when building their models; however this latter study is exploratory rather than intended to construct and test predictive models, and focusing on the problem of contribution acceptance prediction (pull requests). The use of contextual features has been considered in several previous studies, but not in conjunction with dynamic features. A few related studies use predictive splitting and are designed to be used at issue creation time.

[1]This criterion is included because some previous studies calculate features "as of the closing time" of each issue – e.g. they calculate the total number of comments received by an issue throughout its lifetime. Such studies are useful for post-mortem analysis of issue closing time, but not for predictive purposes.

In summary, the scope of the present study is unique in that it uses static, dynamic and contextual features on a large issue dataset to construct issue lifetime models predictively (i.e. all predictions are strictly based on data available at the time the prediction is made).

Table 1: Synthesis of related work.

Paper	LD	DF	CF	PT	MA	TYPE
Weiss et al. [28]				✓	✓	IL
Giger et al. [9]		✓			✓	IL
Panjer [21]						IL
Francis and Williams [8]					✓	IL
Assar et al. [2]				✓	✓	IL
Guo et al. [13]			✓		✓	IL
Marks et al. [19]			✓			IL
Gousios et al. [11]	✓		✓			CL
Jiang et al. [15]						CL
Ye et al. [31]	✓		✓			CL
Tian et al. [24]			✓	✓	✓	IA
Choetkiertikul et al. [6]			✓	✓		IA
Antionol et al. [1]					✓	IA
Xia et al. [29]			✓			IA
Shihab et al. [23]			✓			IA
Guo et al. [14]			✓		✓	IA
Our approach	✓	✓	✓	✓	✓	IL

3. DATASET

GitHub makes the data about public repositories available through public APIs. In this work we use data collected by a third-party project, namely GHTorrent[10]. GHTorrent collects GitHub data through public event streams and enhances it with data obtained via additional API calls. The data used in this work originates from GHTorrent's MySQL database dump, dated 1 April 2015. In addition, we used GHTorrent's MongoDB service to query the textual information (queries issued in January 2016).

3.1 Filtering

At the time of the data dump, GitHub had more than 7 million project repositories (not counting forked ones). Not all repositories in GitHub are software projects [16] and many of them use GitHub for code hosting but not for issue tracking. In order to avoid analyzing non-software projects (e.g. pure documentation projects), projects that do not use GitHub for issue tracking, and other special cases such as one-man projects or projects with little issue activity, we filtered the dataset using the following rules:

- Projects must have been created between January 1, 2012 and December 31, 2014. We limited our observation period to this interval, because the data of older projects is only partially available in GHTorrent. Even tough the dataset also contains events until April 2015, we chose the ending date to be in 2014 as due to the delayed crawling behavior of GHTorrent not all changes are instantly visible.

- Projects must not be forks of existing GitHub projects.

- Projects must have at least 100 opened issues and one closed issue. This criterion guarantees that we only include projects that actively use the issue tracker.

- Projects must have at least five commits to the main repository. This criterion guarantees that we only analyze projects where there is some development activity.

- Projects must not show any activity before the repository creation date. In GitHub, it is possible to fork a repository and therefore inherit an already existing code base which technically shows up as code committed before the project creation.

An examination of the selected data revealed that some projects had unexpectedly high issue creation activity in short periods, such as several thousand created issues in a single day. This phenomenon indicates a data import from an older tracking system or the automatic creation of issues via GitHub's API. To get rid of possible import behavior, we additionally filtered out projects that created or closed more than 2000 issues in any single month or created more than 500 issues in any single day.

As the data ranged between 2012-2014, the selection includes projects with maximum 3 years of history and minimum of 1 month of history. We decided to remove projects shorter than 8 months to have enough time to observe issue closing in every retained project.

Issues can also be reopened and closed multiple times. This affects about 4% of the issues in our sample. We decided to remove issues that get reopened since the scope of the present study is to predict "first-closing time". The phenomenon of issue reopening is a question that deserves a separate treatment. Note also that an issue being closed in the dataset does not necessarily imply that it has been "fixed" to the satisfaction of the issue creator. An issue may be closed for a variety of reasons, such as it being a duplicate issue or because someone in the project team deems it irrelevant or unresolvable.

The dataset obtained after the above filtering contains 4024 projects, comprising 967 037 issues in total of which 675 970 (69.9%) have been closed and 291 067 (30.1%) have not been closed during the observation period. The number of issues per project varies by a factor of almost 50: the smallest project having 100 issues and the largest project having 4885 issues in total. The mean number of issues per project is 240 and the median is 163.

3.2 Analysis of Issue Lifetime

Figure 1 shows the issue lifetime distribution for the 69.9% of issues that get closed in the observation period (bottom box-plot) and for the set of remaining "sticky issues" (top box-plot). We use the term *sticky issue* to refer to issues that do not get closed in our observation period. For the sticky issues, the lifetime is calculated with the assumption they are all be closed on 1 January 2015 (recall that we only retained issues created in 2014 or before).

The median lifetime for the closed issues is 3.7 days, the mean lifetime is 32.6 days and 90% issues get closed in 96.4 days or less. We observe that the median lifetime for sticky issues is 280 days, which is approximately 75 times larger than for closed issues. This long lifetime gives us confidence that most of the sticky issues are indeed long-lived issues rather than issues that will be closed shortly after the end of the observation period. Note that the maximum theoretical lifetime of any sticky issue is 1092 days (i.e. this is the number of days between the start of the observation period and 1 Jan. 2015).

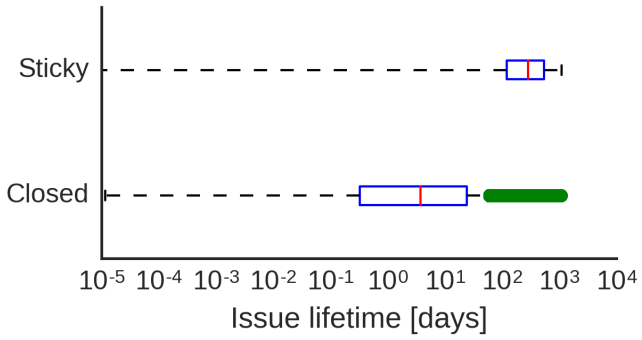


Figure 1: Issue lifetime box-plots for closed and sticky issues. The green-filled line represent outliers not falling into the inter-quantile range

4. MODEL CONSTRUCTION

The overall predictive model construction method involves extracting features to characterize issues in the dataset, training the model, and evaluating the model. In the following subsections, we give detailed information about the first two steps.

4.1 Features

Our approach is based on applying supervised machine learning. Input for learning algorithms is a set of features that would describe each issue as detailed as possible. Next we list the feature we extracted from each issue and justification for this (The features are listed the in Table 2).

During the feature engineering process, we tried to come up with features that would capture the properties of issues, the activity of the project and issue submitter around the time of issue creation. The assumption is that besides individual issue factors, the surrounding context also determines if an issue will be closed.

We initially came up with 35 features. We identified correlated features by calculating Spearman's rank correlation between all pairs of features and manually removed a feature from the each pair of those with a correlation value larger than 0.8. The decision which feature to remove was done manually, but if a feature was correlated with multiple other features, it was removed first. For the remaining 30 features, we calculated the chi-squared score [18] with the training label and removed features that were ranked to last third of all the features. This gave us 21 features. The features removed through the chi-squared analysis were those that were sparse and only had assigned value in a small subset of issues.

To capture the dynamic aspects of open issue reports, we calculate the evolving features at different time points. For example, the number of comments is changing over time, but the issue title is not. In Table 2, the dynamic features have suffix T in their name.

4.1.1 Issue Features

The first group of features describe the issue itself. For example number of comments (`nCommentsT`) can be regarded as a measure of engagement on the issue. Guo et al. have found that more commenting on the bug report can lead to a faster fix [13] and Tsay et al. [25] have shown that more comments on a pull-request makes it more likely to be accepted. Besides the comments itself, the amount of persons interacting with the issue might impact the issue resolution time. Number of actors (`nActorsT`) is the total amount of persons who have had interactions with the issue - opening, closing, commenting, referencing. These features are dynamical in nature - the amount of comments can be different at each observation point.

Other features in this group, such as number of times issue has been assigned, mentioned from another issues, reflect the overall activity of the issue and are dynamical. Besides dynamical features, we extracted the issue content text length (`issueCleanedBodyLen`) to represent the length or possible complexity of the issue.

Issue reports' unstructured textual content has shown to have predictive power itself for estimating the issue lifetime [28]. The typical approach for analyzing text data would be to convert issue reports into bag of words representation. This typically leads to a large sparse representation, as some words are only present in a small set of documents. Adding all these features to our previously defined features would make the classification task harder as the amount of parameters can become very large. In addition, it makes it harder to understand what are the important features.

We decided not to include textual features directly into our model. Instead, we transformed the textual content into a single score, representing the likelihood that issue report with such text will be closed in a period. A similar approach has been previously used for bug classification [32] and clustering [2].

For each issue, we joined issue title and content text into a single text[2]. We parsed the markdown representation, completely removed all source code blocks, tables and links. Next we removed remaining markdown markup and only kept the textual content. We converted all text into lowercase, removed punctuation, English stop words (such a, the, that, etc). We also applied Porter stemming algorithm to extract the stem for each word. For each issue, we kept single words and n-grams of size 2 and we constructed the bag of words representation using feature hashing [27], with 2^{20} features. Each vector is normalized with l_2 norm and is non-negative. Using the hashing trick, we can keep all the words and n-grams and do not have to construct a dictionary during training that contains all allowed words, which is helpful when deriving the score for different parts of the data as the usage of dictionary could also leak information about the target label.

We divided the training data into two random sets. We use the first subset to train a model on text vectors and predict the score on the second subset. The predicted scores will be the corresponding textual score (`textScore`) for the second subset. We repeat the process the other way around - we train on the second subset and predict on the first subset and attach prediction score to the features of the first subset. The scores will be appended to the overall feature set.

For test set, we train on the whole training data and predict for the test set's textual content and add the score as new column. Note that in any case, we do not leak any information about the target as we always derive the score using different subsets and do not compare to actual labels.

For the training, we used Stochastic Gradient Descent based classifier [26] with logistic error, l_2 regularization with

[2]We use the latest version of title and body as GHTorrent only keeps the last version if the fields are updated.

the multiplier 0.001, shuffling before iterations and 5 iterations. This model is suitable for problems with large number of features as the regularization helps to control over-fitting by constraining coefficient values and is fast to train. Other alternatives to consider would be the linear support vector machines (obtaining the probabilities is more costly) and Naive Bayes. We briefly experimented with the latter one and the results were approximately in the same order.

4.1.2 Issue submitter features

Individual reputation has shown to have impact on the time issue will be fixed [13]. The idea of this group of features is to capture the previous interactions that the issue submitter has had prior to the submission in the context of the project. The features extracted reflect the prior activities done by the submitter in the past three months in the context of this project, such as the number of issues created (`nIssuesByCreator`) and the number of commits (`nCommitsByCreator`).

4.1.3 Participant's features

Open source projects have different levels of participation, ranging from core team to irregular contributors. To study the possible effects of individual influence, we extract the number of commits made (`nCommitsByActorsT`) by the people participating in the issue, or in other words, actors. Actors are all people who comment on an issue, change any of its properties such as tags, milestones, assignments. In addition, we count a person to be an actor if they reference it from another issue or commit message. These features are dynamic and we calculate them over the period of two weeks before the observation point, to see if the persons who have had interactions with the issue are still active.

4.1.4 Project (contextual) features

The aim of project features is to capture the overall state of the project. Our hypothesis is that if the project is not active, i.e., there has not been coding activity recently or no issues have been closed, then it is also likely that new issues will not receive attention. We calculate the total amount of commits in the past three months (`nCommitsInProject`), total new issues in the past three months (`nIssuesCreatedInProject`), and the same activity in the past two week with respect of the observation point (`nCommitsInProjectT`, `nIssuesCreatedInProjectT`). We use different period ranges of three months and two weeks to prevent possible overlap and correlated features, as in some cases, the dynamic features can be calculated also close to the issue submission and therefore have overlap with each other.

Note that we do not use any identifier of the specific project to which an issue belongs, as our goal is to study the performance of cross-project models built on large project repositories. However, information about the project characteristics and its state is captured via the above contextual features.

4.2 Model training

Often for prediction tasks, cross-validation is used for evaluating goodness of the model and making sure the model performance is reliable on different subsets of data. Our goal is to train a predictive model that also takes into account the temporal information of issues. This prevents us from using traditional cross-validation. The idea is that for training data, we can only use data from a period that is before the period in which issues contained in the test dataset have been opened. This corresponds to a real world scenario - we can not use future data for training and then test on past data.

Our dataset covers three years, 2012, 2013, 2014. We split the data into two at September 1, 2013. Everything before September 2013 is for training data, and everything after that point in time is for testing. This split leaves 424004(43.9%) issues into training set and 543033(56.1%) into testing set. In addition, the final amount of issues that can be used for training and testing depends on the task as the amount of issues that could be used for estimating if an issue will be closed in a year is smaller than the amount issues used for estimating if an issue will be closed in a month.

We trained classification models for different combinations of an observation point (i.e. the point in an issue's lifetime when the prediction is made) and a prediction horizon (i.e. the timeframe after issue creation by which we predict that an issue will be already closed). For example, an observation point of 7 days means that we make a prediction for an issue that has been open for already 7 days. Meanwhile, a prediction horizon of 30 days means we predict whether an issue will be closed within 30 days of its creation or not (note that this is a binary classification task).

The observation points and prediction horizons are chosen to match calendric periods (day, week, fortnight, month, quarter, semester and year) and of course, the issue creation time itself is taken as one of the observation points. This leads to seven observation points (0, 1, 7, 14, 30, 90, and 180 days) and seven prediction horizons (1, 7, 14, 30, 90, 180, and 365 days).

Note that the models with a zero-day observation point are such that the dynamic features are not meaningful. A small caveat though is that the dataset also has issues where the first comments arrive at exactly the same time as the issue itself. We do count such comments when calculating the zero-day features in order to keep the feature calculation method consistent. Thus the dynamic feature nCommentsT is meaningful for zero-day models.

For each pair (observation point, prediction horizon), we trained a classifier to predict whether an issue will be closed before or after the end of the prediction horizon. Naturally, such a predictive model only makes sense when the prediction horizon ends after the observation point. Hence, there are only 28 valid combinations of an observation point a prediction horizon, and this is the number of models we trained.

In line with the predictive setting, when evaluating a model for a given observation point, we only make predictions for issues that were not yet closed at the observation point in question. Hence, the sizes of the training and the testing sets are different for each (observation period, prediction horizon) combination.

In this paper, we approach the issue lifetime prediction problem using binary classification. The problem could also be approached by training a multi-class classifier, for example where different classes denote if an issue will be closed in a corresponding time range. We choose binary classification as it helps us to understand at which points in time it is feasible to make predictions and how the accuracy of the models changes depending on the chosen observation point

Table 2: Features extracted for each issue. Suffix "T" (short for "Time") in the feature name denotes that this feature is dependent on the observation point.

Feature	Description
Issue features	
nCommentsT	Number of comments issue has received before the observation point T.
nActorsT	Number of unique persons who have commented, referenced or subscribed to the issue before the observation point T.
nAssignmentsT	Number of assignment events before T.
nLabelsT	Number of labels added before T.
nMentionedByT	Number of times issue was mentioned from other issues before T.
nReferencedByT	Number of times issue was mentioned in commit messages using the issue id, before T.
nSubscribedByT	Number of persons subscribing to receive updates on the issue before T.
meanCommentSizeT	Average comment size of the comments received before the observation point T.
issueCleanedBodyLen	Length of the combined title and body with markdown parsed and tags removed.
textScore	Classification score obtained from cleaned issue title and content.
Issue submitter features	
nIssuesByCreator	Number of issues created by the issue submitter in the three months prior to issue opening.
nIssuesByCreatorClosed	Number of issues created by the issue submitter that were closed in the three months prior to issue opening.
nCommitsByCreator	Number of total commits to the issue repository by the issue submitter in the three months before the issue opening.
Participant's features	
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 observation point T.
nCommitsByUniqueActorsT	Number of unique actors who committed code to the project repository during the period from two weeks before the issue creation to observation point T.
Project features	
nIssuesCreatedInProject	Number of issues created in the project during the three months prior to issue creation.
nIssuesCreatedInProjectClosed	Number of issues created and closed in the project in the three months prior to issue creation.
nCommitsInProject	Number of commits created in the project in the three months prior to issue creation.
nIssuesCreatedProjectT	Number of issues created in the project during the period of 2 weeks before the issue creation until the observation point T.
nIssuesCreatedProjectClosedT	Number of issues created and closed in the project during the period of 2 weeks before the issue creation until the observation point T.
nCommitsProjectT	Number of commits in the project during the period of 2 weeks before the issue creation until the observation point T..

and prediction horizon.

4.3 Classification Method

We use Random Forest [4] for classifier construction. The Random Forest classifier trains multiple decision trees on the same data, but for each tree using different random subsets of the data and random subsets of features when creating splits for individual trees. The final predictions are created from finding the mode of classes from each individual tree. The randomization combined with multiple trees helps to avoid over-fitting. Random Forests have shown very good performance on different datasets, even when compared to other well known methods such as logistic regression, support vector machines or gradient boosted decision trees [7]. For the hyper-parameters, we set the number of trees to 1000 and limit the maximum tree depth to 5.

As we train in total 28 classifiers on the different observation and prediction horizon combinations, we do not perform any additional hyper-parameter optimization in order to use the same classifier for all different prediction horizon values and avoid optimizing each task separately. The results reported below should thus be construed as lower-bounds that can be further improved via hyper-parameter optimization.

4.4 Evaluation

To evaluate the classifiers, we use the following technical measures: precision, recall, F1-score and area under the receiver operating characteristic (ROC) curve (AUC). In our classification task, the positive class denotes issues that will be closed in the specified period and the negative class denotes issues that will not be closed in the specified period. We use precision and recall for the positive class.

Precision in our context measures the fraction of correctly classified closing issues over all issues predicted to close. Recall measures the fraction of correctly classified closing issues over all closing issues. The ideal precision and recall scores are 1, the worst case is 0. The F1-measure is the harmonic mean of precision and recall, defined as $F_1 = 2 \cdot (precision \cdot recalll)/(precision + recalll)$. The F1-measure allows us to quantify precision and recall with a single number.

In addition, we make use of the true positive (TP), false

positive (FP), false negative (FN) and true negative (TN) numbers to understand why in some cases the precision and recall are changing.

The AUC measures the probability that a classifier will rank randomly chosen positive instance higher than randomly chosen negative instance. In the context of this work, AUC measures what is the probability with which we can rank randomly chosen closed issue higher than randomly chosen issue that will not close. For a random classifier, the value of AUC will be 0.5, for an ideal classifier it will be 1.

The usage of different measures is due to two reasons. Firstly, we want to compare our results with existing work and all these measures are used in previous work. Secondly, different measure help us to prove the usefulness of the model in different scenarios. If we would like to make an individual prediction for a single issue, we are interested good precision and recall scores as we would like to get a correct prediction for each item. If we were interested finding the ranked list of most likely issues to be closed in a project, then the AUC can reflect how well we can do this.

4.5 Feature importance

The Random Forest classifier lends itself to measuring individual feature importance via *mean decrease in impurity* [5]. For each feature, this method calculates how much it decreases the Gini impurity of a node in a tree and averages this quantity across all trees in the forest. This method enables us to rank the important features in a model. We will use the feature importance to understand which features are useful in making the decision and also compare feature ranking for different observation point scenarios.

5. RESULTS

In this section we report the evaluation results and analyze them with respect to the questions posed in Section 1.

5.1 Classifier performance (RQ1)

We analyze model performance for different observation points to answer our research question RQ1 (*What level of accuracy is achieved by classification models trained to predict issue lifetime using static, dynamic and contextual features?*) Table 3 shows the obtained AUC scores for each combination of observation point and prediction horizon. The scores all fall into the range of 0.64 to 0.707. From the results, it emerges that long-term predictions can be made with higher AUC than short-term predictions. Indeed, for each observation point, the best AUC score corresponds to the longest prediction horizon.

We also calculated F1-scores for each model as shown in Table 4. F1-scores put into evidence the cases of models that fail overall, meaning that while they do identify correctly the issues that are most likely to close (i.e. they have a certain level of ranking accuracy as measured by AUC), they have either low precision or low recall or both. We observe the lowest F1-scores when the gap between the observation point and the end of the prediction horizon is the smallest (cf. the diagonal values in the table). The reason for this phenomenon is that in these cases, the class imbalance is the highest. The best F1-scores are obtained when the gap between the observation point and the prediction horizon is the largest. In other words, we are more accurate when making longer-term predictions. One possible explanation for this is that there are more issues with smaller lifetime

Table 3: AUC scores for different Prediction horizon and observation point (OP) values.

	Prediction horizon(days)						
OP	1	7	14	30	90	180	365
0	0.653	0.660	0.663	0.665	0.666	0.681	0.707
1		0.641	0.644	0.649	0.658	0.680	0.688
7			0.639	0.644	0.653	0.680	0.671
14				0.646	0.653	0.681	0.665
30					0.653	0.681	0.659
90						0.687	0.661
180							0.684

Table 4: F1 scores for different Prediction horizon and observation point (OP) values.

	Prediction horizon(days)						
OP	1	7	14	30	90	180	365
0	0.437	0.604	0.659	0.715	0.781	0.830	0.898
1		0.392	0.478	0.571	0.695	0.766	0.863
7			0.236	0.402	0.587	0.702	0.806
14				0.278	0.513	0.657	0.771
30					0.395	0.574	0.712
90						0.337	0.589
180							0.387

(note that half of the issues have lifetime smaller than 3.7 days) and making predictions for them is harder as there are more different reasons for closing them. Meanwhile, there is a smaller amount of issues with longer lifetime and therefore the features can better capture the corresponding reasons for closing them.

The raw experimental results give us a broad view of how well the models perform. But since the sample sizes differ considerably across different (observation point, prediction horizon) combinations, we cannot directly compare performance across models. Accordingly, in Table 5, we report the results for experiments, where the testing set is always of the same size for all models with a given prediction horizon. This makes it more meaningful to compare models across different observation points. For each prediction horizon, we table gives the performance across all observation points that are before the prediction horizon. For example, for the prediction horizon of 365 days, we only include issues that have not been closed in the first 180 days. For these issues, we can perform the prediction at different observation points. Figure 2 illustrates the concept of observing the same set of issues over different observation points (and thus different observation periods) in order to predict whether or not the issue will close before 365 days or later. Similarly, we do the analysis for other prediction horizons, with each time one less possible observation point and a larger test set.

When looking at the AUC and precision values (Table 5), we observe that in many cases (specifically for prediction horizons of 365, 180 and 90 days) the scores increase as the observation point increases. For the prediction task whether an issue will be closed in 365 days, we observe an 8.7% increase in AUC (from 0.629 to 0.684) and a 33.2% increase in precision (0.187 to 0.249) across the seven corresponding observation points. This supports the hypothesis that observing an issue over an extended period can lead to better predictive power. In contrast, the recall scores are more fluctuating and show a slight negative trend when the ob-

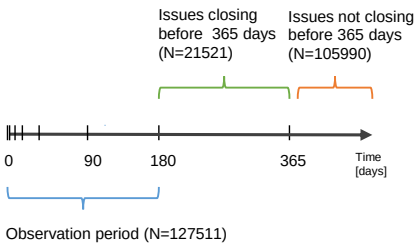


Figure 2: Observing a set of issues for prediction whether they will close before 365 days or after. The numbers correspond to the first group in Table 5.

servation point increases. The reason is that the amount of true positives (correctly classifying closing issues) decreases only slightly with an increasing observation point, while the amount of true negatives (correctly classified issues that will not close) keeps increasing.

To summarize the findings with respect to RQ1, we conclude that when making repeated predictions for issues as they evolves over time, predictions made at later observation points yield higher AUC and precision scores, but lower recall.

5.2 Feature importance (RQ2)

In order to address RQ2 (*What features are most important when predicting issue lifetime?*), we analyze the mean decrease in impurity for each feature as discussed in Section 4.5. We are particularly interested in understanding the role played by dynamic features, hence we compare the feature importance for the models constructed at creation time (the zero-day models) versus one week after issue creation (the 7-days models). This approach allows us to understand the role played by dynamic features early on during the lifetime of an issue. Regarding the prediction horizon, we look at models constructed to predict if an issue will close after 30 days (short-term predictions) and after 180 days and 365 days (long-term predictions).

The ranking of feature importance for the zero-day models is given in Figure 3. In the case of the zero-day model with a 30-days prediction horizon (cf. Figure 3a), the top-ranked features are `nIssuesCreatedProjectClosedT` and `nIssuesCreatedInProjectClosed` (i.e. issue closing activity in the two weeks and the three months before the issue submission respectively). Not far below we also see `nCommitsProjecT` (commit activity). The presence of these features at the top of the ranking suggests that contextual features play an important role when making predictions at creation time. The third feature in the ranking is the number of comments (`nComments`), which in the zero-day models only has two possible values (0 or 1) – some issues come created without any attached comment at issue creation, while others come with a comment having the same timestamp as that of issue creation. Expectedly, the presence of this initial comment carries some information about issue lifetime. We also observe that `textScore` is highly ranked, stressing the potential value of extracting information from text attached to issues.

In the case of the zero-days model with a 180 days prediction horizon (Figure 3b), we observe that the top-3 features are the same as in the model with 30-days prediction horizon. On the other hand, the importance of `textScore` when predicting for 180 days is lower than when predicting for 30 days. In other words, the text attached to the issue is useful for short term prediction, but becomes less important for longer-term prediction.

In both cases (zero-day models with 30-days and 180-days horizons), the features that have least importance are those related to size of comments, commit messages and issue labels. This is simply because these features are dynamic and hence not meaningful for zero-day models.

Let us now compare the feature importance ranking of the zero-day models (Figure 3) against the 7-days models (Figure 4) with a 180-days prediction horizon. We observe that feature `textScore` has less importance in the 7-days models, and instead dynamic features take it over in importance, e.g. number of unique persons (`nActors`), average comment size (`meanCommentSizeT`) and number of references to the issue (`nReferencedByT`). This observation reinforces the hypothesis that dynamic features carry information that complements static features, particularly when making long-term predictions.

One can wonder if a longer-term prediction horizon affects feature importance significantly. To this end, Figure 4b displays the feature importance ranking of the 7-days model with a 365-days prediction horizon. It turns out that this raking is very similar to the one for the 7-days model with 180-days horizon. Although not shown here, we observed a similar ranking in the 7-days model with 90-days horizon, suggesting that the task of predicting closing time with a few months horizon is similar to that of predicting it with a one-year horizon.

In summary, with reference to RQ2, we can say that contextual features complement static features both for short-term and long-term predictions. Dynamic features in turn complement both static and contextual features and their inclusion explains the observed increase in accuracy of models built for later observation points.

6. DISCUSSION AND LIMITATIONS

The observed accuracy of our models (AUC and precision scores) suggests that predictive models of issue lifetime across large sets of open source projects could potentially be used in practice if users were willing to tolerate some fluctuation in their predictive accuracy. Compared to previous research, Giger et al. [9] obtain AUC scores between 0.649 to 0.823 when increasing dynamic feature observation period from 0 days until 30 days (Eclipse JDT project). Their results also show fluctuations in performance, i.e., observing features for longer period does not lead to monotonic increase in model performance. Their model precision scores are also better, ranging in 0.635 to 0.885, but recall values are lower than in our experiments, ranging from 0.485 to 0.661. They also experience high variation across projects, especially with Gnome Gstreamer project dataset where performance decreases with longer post submission data, with the AUC values mostly decreasing from 0.724 to 0.586. Francis & Williams [8] similarly show that a same issue lifetime prediction method can have different performance on an open source and closed source private project. This confirms that issue lifetime prediction varies across projects, and suitable accuracy can not be always obtained.

The experimental setup used by Giger et al. [9] is not

Table 5: Prediction performance of models tested with a constant test size (N) for any given prediction horizon.

Observation point	AUC	Precision	Recall	F1	TP	FP	FN	TN
	Prediction horizon of 365 days (N=127511)							
0	0.629	0.187	0.851	0.306	18311	79679	3210	26311
1	0.631	0.185	0.905	0.308	19484	85702	2037	20288
7	0.629	0.188	0.914	0.311	19674	85136	1847	20854
14	0.635	0.195	0.921	0.322	19811	81582	1710	24408
30	0.637	0.204	0.912	0.334	19635	76484	1886	29506
90	0.654	0.244	0.875	0.381	18831	58433	2690	47557
180	0.684	0.249	0.874	0.387	18802	56839	2719	49151
	Prediction horizon of 180 days (N=196976)							
0	0.618	0.181	0.814	0.297	24757	111782	5661	54776
1	0.633	0.206	0.783	0.326	23830	91756	6588	74802
7	0.656	0.208	0.814	0.331	24767	94540	5651	72018
14	0.664	0.207	0.826	0.331	25117	96029	5301	70529
30	0.675	0.208	0.836	0.333	25416	97003	5002	69555
90	0.687	0.211	0.834	0.337	25374	94884	5044	71674
	Prediction horizon of 90 days (N=280862)							
0	0.615	0.251	0.793	0.381	47477	141823	12375	79187
1	0.616	0.263	0.746	0.389	44643	124887	15209	96123
7	0.630	0.268	0.700	0.388	41912	114349	17940	106661
14	0.639	0.268	0.682	0.385	40801	111222	19051	109788
30	0.653	0.272	0.721	0.395	43179	115474	16673	105536
	Prediction horizon of 30 days (N=352555)							
0	0.625	0.162	0.802	0.270	38331	197918	9490	106816
1	0.620	0.178	0.653	0.280	31228	143910	16593	160824
7	0.632	0.174	0.611	0.271	29195	138131	18626	166603
14	0.646	0.176	0.653	0.278	31248	145930	16573	158804
	Prediction horizon of 14 days (N=404045)							
0	0.623	0.133	0.796	0.227	35243	230365	9057	129380
1	0.620	0.144	0.619	0.233	27443	163398	16857	196347
7	0.639	0.145	0.642	0.236	28437	168189	15863	191556
	Prediction horizon of 7 days (N=516927)							
0	0.644	0.259	0.820	0.394	90343	258308	19829	148447
1	0.641	0.278	0.664	0.392	73135	190227	37037	216528

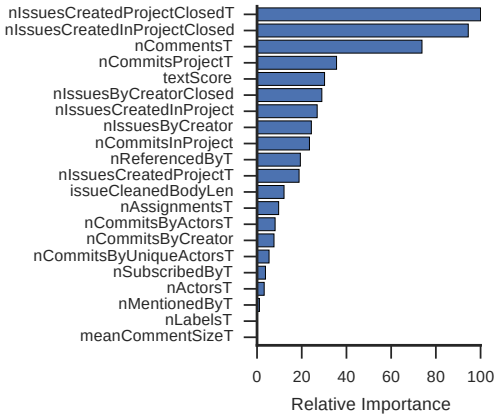
directly comparable to ours, as they used different prediction task and their dataset properties were different, such as considerably larger median issue lifetime. Another aspect that might work in their favor is that they perform cross validation without using any temporal information about the creation of issue reports (i.e. no temporal split), so that "future data" may be used to classify an issue at a given time point. Even tough, their results with using only static features are comparable in terms of AUC, where they had scores ranging from 0.649-0.724 across projects.

With respect to the use of temporal splitting, the reported findings are in line with those of Assar et al. [2], who observe that when using temporal splits and observing issues for a longer period, the prediction error becomes lower compared to shorter observation periods.

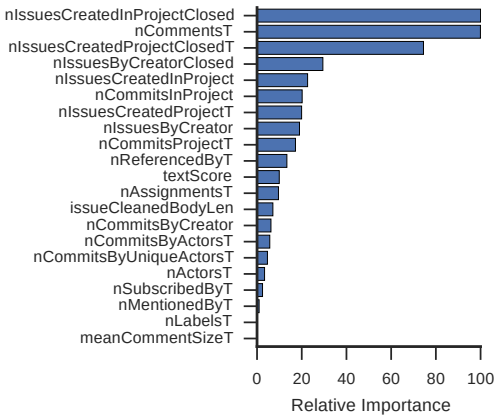
Our work uses issues from more than 4000 projects. The projects have different development practices, backgrounds, resources and goals. Hence the heterogeneity in project properties can affect our results as the models can not make sound generalization based on issues from different projects. It has been shown that features, such has developer reputation that can be important in projects determining the issue lifetime, is not important in another scenario [3]. Similarly, we do not distinguish between different types of issues (e.g. bugs vs. feature requests), although this can have an effect on the resolution time in Github [20].

Another limitation of the study is the lack of cross-validation in the evaluation of the classification models. We have chosen to use temporal splits between training and testing data, as this is exactly how the models are trained when used in real world scenario, and should give a better estimate of issue lifetime. The drawback of this choice is that it does not leave much room for performing multiple test splits, since the period covered by the dataset is just of the length required for one split.

In addition, the precision scores of our models are low –

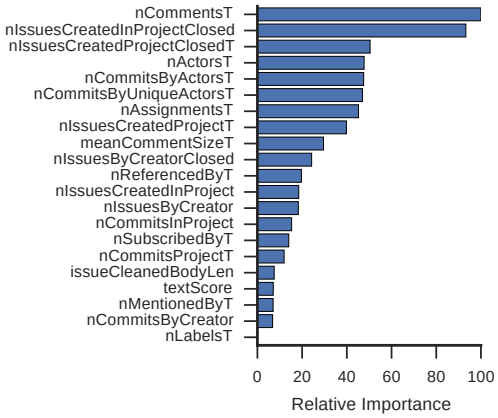


(a) 30-days horizon

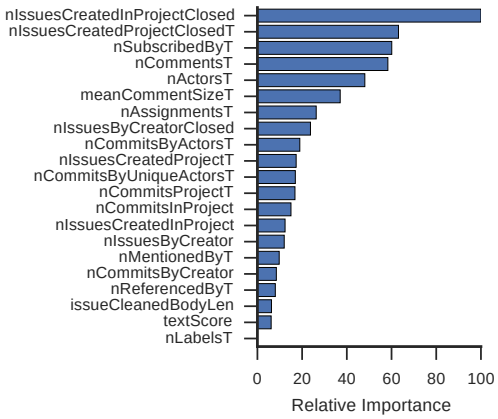


(b) 180-days horizon

Figure 3: Feature importance for the zero-day models with 30-days and 180-days horizon.



(a) 180-days horizon



(b) 365-days horizon

Figure 4: Feature importance for the 7-days models with 180-days and 365-days horizons.

i.e. issues that we predict will close before the horizon often remain open past it. This restricts their direct practical applicability. A critical direction for future work is thus to investigate the reasons for low precision and to improve the models for example by introducing additional features (e.g. from the code commits or from the text of the comments).

A potential threat to validity is that our dataset contains non-software development projects as removing all of them manually would be impractical. To estimate the extent of non-software development projects in the dataset, we manually checked a random sample of 100 projects. We found that 89 of them can be classified as software projects (i.e. projects containing code and build files or deployment guides). Two projects contained only documentation, two contained specifications, two data, one was used purely as an issue tracker for an externally hosted, and four had been since deleted and thus their nature could not be asserted.

7. CONCLUSION AND FUTURE WORK

In this paper we studied the problem of predicting issue lifetime in GitHub projects for different calendric periods, using a combination of static (creation-time), dynamic and contextual features. Based on issues extracted from a sample of 4000 projects, we show that such predictive models exhibit better accuracy when trained with one-day-old or one-week-old issues to predict whether or not an issue will remain open after one month or a longer period. This study highlights the importance of dynamic and contextual features in such predictive models.

The predictive models studied in this paper benefit from being trained on a large dataset. The counter-part of this benefit is that the set of projects in the dataset are very heterogeneous, making the prediction problem more difficult. One possible direction for future work is to study the performance of predictive models trained for specific types of projects (i.e. partial classifiers), such as to strike a tradeoff between volume of projects and homogeneity.

Another direction for future work is to extend the feature set with more dynamic and contextual features, like for example features extracted from the text of the comments added during the lifetime of an issue, or features capturing how busy the developers are handling other concurrent issues in the same or in other projects.

Supplementary Information. Additional results and the list of projects used in this research are available at https://github.com/riivo/github-issue-lifetime-prediction

Acknowledgments. This research was supported by the Estonian Research Council.

8. REFERENCES

[1] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds* (2008), CASCON '08, ACM, pp. 23:304–23:318.

[2] Assar, S., Borg, M., and Pfahl, D. Using text clustering to predict defect resolution time: a conceptual replication and an evaluation of prediction accuracy. *Empirical Software Engineering* (2015), 1–39.

[3] Bhattacharya, P., and Neamtiu, I. Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories* (2011), MSR '11, ACM, pp. 207–210.

[4] Breiman, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[5] Breiman, L., Friedman, J., Olshen, R., and Stone, C. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[6] Choetkiertikul, M., Dam, H. K., Tran, T., and Ghose, A. Predicting delays in software projects using networked classification. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (2015), pp. 353–364.

[7] Fernández-Delgado, M., Cernadas, E., Barro, S., and Amorim, D. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res. 15*, 1 (2014), 3133–3181.

[8] Francis, P., and Williams, L. Determining 'grim reaper'; policies to prevent languishing bugs. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on* (2013), pp. 436–439.

[9] Giger, E., Pinzger, M., and Gall, H. Predicting the fix time of bugs. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering* (2010), RSSE '10, ACM, pp. 52–56.

[10] Gousios, G. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (2013), MSR '13, pp. 233–236.

[11] Gousios, G., Pinzger, M., and Deursen, A. v. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ICSE 2014, ACM, pp. 345–355.

[12] Gousios, G., Zaidman, A., Storey, M.-A., and van Deursen, A. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, IEEE Press, pp. 358–368.

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

[14] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work* (2011), CSCW '11, ACM, pp. 395–404.

[15] Jiang, Y., Adams, B., and German, D. M. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (2013), MSR '13, IEEE Press, pp. 101–110.

[16] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (2014), MSR 2014, ACM, pp. 92–101.

[17] Kikas, R., Dumas, M., and Pfahl, D. Issue dynamics in github projects. In *Product-Focused Software Process Improvement*, vol. 9459 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 295–310.

[18] Liu, H., and Setiono, R. Chi2: feature selection and discretization of numeric attributes. In *Tools with Artificial Intelligence, 1995. Proceedings., Seventh International Conference on* (1995), pp. 388–391.

[19] Marks, L., Zou, Y., and Hassan, A. E. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering* (2011), Promise '11, ACM, pp. 11:1–11:8.

[20] Murgia, A., Concas, G., Tonelli, R., Ortu, M., Demeyer, S., and Marchesi, M. On the influence of maintenance activity types on the issue resolution time. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering* (2014), PROMISE '14, ACM, pp. 12–21.

[21] Panjer, L. D. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories* (2007), MSR '07, IEEE Computer Society, p. 29.

[22] Saha, R., Khurshid, S., and Perry, D. An empirical study of long lived bugs. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on* (2014), pp. 144–153.

[23] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W. M., Ohira, M., Adams, B., Hassan, A. E., and Matsumoto, K.-i. Studying re-opened bugs in open source software. *Empirical Software Engineering 18*, 5 (2012), 1005–1042.

[24] Tian, Y., Lo, D., and Sun, C. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on* (2013), pp. 200–209.

[25] Tsay, J., Dabbish, L., and Herbsleb, J. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ICSE 2014, ACM, pp. 356–366.

[26] Tsuruoka, Y., Tsujii, J., and Ananiadou, S. Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty. In

Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1 (2009), ACL '09, Association for Computational Linguistics, pp. 477–485.

[27] Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning* (2009), ICML '09, ACM, pp. 1113–1120.

[28] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories* (2007), MSR '07, IEEE Computer Society, p. 1.

[29] Xia, X., Lo, D., Shihab, E., Wang, X., and Zhou, B. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering 22*, 1 (2015), 75–109.

[30] Xia, X., Lo, D., Wang, X., Yang, X., Li, S., and Sun, J. A comparative study of supervised learning algorithms for re-opened bug prediction. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on* (2013), pp. 331–334.

[31] Yu, Y., Wang, H., Filkov, V., Devanbu, P., and Vasilescu, B. Wait for it: Determinants of pull request evaluation latency on github. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (2015), MSR '15, IEEE Press, pp. 367–371.

[32] Zhou, Y., Tong, Y., Gu, R., and Gall, H. Combining text mining and data mining for bug report classification. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution* (2014), ICSME '14, IEEE Computer Society, pp. 311–320.