Requirements Eng (2012) 17:117–132
DOI 10.1007/s00766-012-0150-7

RE'11 BEST PAPERS

Early failure prediction in feature request management systems: an extended study

Camilo Fitzgerald · Emmanuel Letier · Anthony Finkelstein

Received: 31 October 2011 / Accepted: 18 March 2012 / Published online: 25 April 2012

Abstract Online feature request management systems are popular tools for gathering stakeholders' change requests during system evolution. Deciding which feature requests require attention and how much upfront analysis to perform on them is an important problem in this context: too little upfront analysis may result in inadequate functionalities being developed, costly changes, and wasted development effort; too much upfront analysis is a waste of time and resources. Early predictions about which feature requests are most likely to fail due to insufficient or inadequate upfront analysis could facilitate such decisions. Our objective is to study whether it is possible to make such predictions automatically from the characteristics of the online discussions on feature requests. This paper presents a study of feature request failures in seven large projects, an automated tool-implemented framework for constructing failure prediction models, and a comparison of the performance of the different prediction techniques for these projects. The comparison relies on a cost-benefit model for assessing the value of additional upfront analysis. In this model, the value of additional upfront analysis depends on its probability of success in preventing failures and on the relative cost of the failures it prevents compared to its own cost. We show that for reasonable estimations of these two parameters, automated prediction models provide more value than a set of baselines for many failure types and projects. This suggests automated failure prediction during requirements elicitation to be a promising approach for guiding requirements engineering efforts in online settings.

Keywords Early failure prediction · Cost-benefit of requirements engineering · Feature requests management systems · Global software development · Open source

C. Fitzgerald (✉) · E. Letier · A. Finkelstein
Department of Computer Science,
University College London, London, UK
e-mail: c.fitzgerald@cs.ucl.ac.uk

E. Letier
e-mail: e.letier@cs.ucl.ac.uk

A. Finkelstein
e-mail: a.finkelstein@cs.ucl.ac.uk

1 Introduction

An increasing number of software development projects rely on online feature request management systems to elicit, analyse, and manage users' change requests [3, 7]. Such systems encourage stakeholder participation in the requirements engineering process, but also raise many challenges [18]: The large number of feature requests and poor structuring of information make the analysis and tracking of feature requests extremely difficult for project managers; this affects the quality of communication between project managers and stakeholders and makes it hard for project managers to identify stakeholders' real needs. Consequently, various problems may arise later in the feature's development life-cycle: An implemented feature may contain bugs caused by ambiguities, inconsistencies, or incompleteness in its description; a newly implemented feature may cause build failures that are caused by unidentified conflicts between the new feature and previous ones; an implemented feature may turn out to be of low value to stakeholders while some other request for a valuable feature may have been wrongly rejected.

There are many ways in which current online feature request management systems could be improved: Techniques have been proposed to cluster similar threads of discussions to detect duplicates [6] and to facilitate the requirements prioritisation process [19]; our first approach to improve such

systems was to allow project stakeholders to annotate discussions with standard requirements defect types (ambiguity, inconsistency, incompleteness, infeasibility, lack of rationale, etc.), providing a more structured way to review and revise feature specifications [9]. This latter approach, however, requires a radical change from the way feature requests management systems are currently used, and the benefits of such a change are hard to demonstrate. The general argument that every defect found and corrected during requirements elaboration saves up to 100 times the cost it would take to correct it after delivery is hard to make in this context: It is not clear whether a requirements defect will actually cause a failure later on in the development process (many features are implemented correctly even if starting from an ambiguous description), and the speed of feature delivery is often viewed as more important than its quality [2].

Our objective in this paper is different: Instead of detecting *defects* in feature requests, we wish to predict *failures*, that is, the possible undesirable consequences of these defects, and we wish to predict these failures automatically from information already present in feature management systems as they are used today. Our approach uses machine learning classification techniques to build failure prediction models by analysing past feature requests with both successful and unsuccessful outcomes. Project managers can then use these prediction models to assess the risk that a decision they are about to make on either assigning a feature request for implementation or rejecting it may later cause a failure. If they find the risk to be too high, they can decide to perform additional upfront requirements analysis on the feature request before making their decision. Performing additional upfront analysis means performing more elicitation, validation, documentation, and negotiation on the feature request before assigning it for implementation [24, 27]. They can also use information on the likelihood of failure to monitor high-risk features more closely during their development.

There is a large volume of work on predicting failures at the code level at later stages of software development life-cycle [31, 32]. Our objective in this paper is to study the extent to which it is possible to predict a variety or product and process failures much earlier in the development process from characteristics of online feature request discussions before a decision is made to implement or reject the feature.

Our specific contributions are the following:

1. We define and present a study of failure types that can be associated with feature requests in 7 large-scale online projects. These failure types extend the usual product failures and integration problems that have been the focus of failure prediction techniques applied at later stage of the life-cycle [23, 31] to include process failures such as abandoned implementation, stalled development, and rejection reversal, which are specific to early failure predictions.
2. We present a cost-benefit model for assessing and comparing the value of early failure predictions. This model provides criteria that are more meaningful to project managers than the standard measures of recall and precision. The model also shows that the random predictor that is often used as a baseline in failure prediction research to generate alerts based on the failure density is not an appropriate baseline for comparing early failure prediction models.
3. We present a tool-supported framework for constructing and evaluating early failure prediction models. The framework supports data extraction from online feature requests management systems, preparation of this data for machine learning classification techniques, and uses an off-the-shelf machine learning tool-set [15] to train and evaluate prediction models using a variety of classification algorithms and predictive attributes.
4. We report experiments in which early failure prediction models for different failures are evaluated using alternate predictive attributes for seven large-scale open-source projects. Our experiments consider simple predictive attributes such as the number of participants in a discussion, the number of posts, the length of discussions, and their textual content. The results show that it is possible to make failure predictions that can be of value at an early stage. The results also tell us which predictive attributes work in this context.

This paper describes in more detail the concepts and techniques originally presented at the 19th International Requirements Engineering Conference [11]. It extends the work presented thereby giving more technical detail on the process of the failure prediction approach and gives the results of 4 new experiments on large open-source projects. These experiments improve the significance of our findings and allow us to make further observations on the contexts in which early failure prediction is likely to be beneficial to a project. Together with the tool-supported framework and evaluation model, they provide the basis for future work into exploring the possibilities for deployment of the approach in the wild and the use of more elaborate predictive attributes to increase the accuracy of predictions.

2 Failures in feature request management systems

2.1 Context

Most open-source projects and an increasing number of enterprise-level projects rely on web-based feature request management systems to collect and manage change

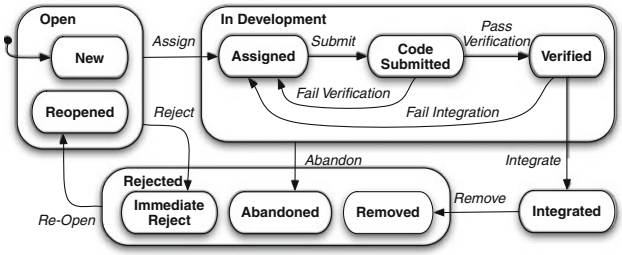

Fig. 1 Life-cycle of a feature request

requests [6, 18]. These systems allow project stakeholders to submit feature requests, contribute to discussions about them, and offer varying possibilities to track their status and development. Stakeholders and developers alike contribute to feature request discussions by submitting posts to an associated discussion thread. Once submitted, these posts may not be edited. Each feature request is associated with meta-data that record its current state (e.g. whether it is opened, in development, or integrated in the product), to which developer it has been assigned, and once developed whether there are bug reports that have been linked to the feature. The meta-data that can be associated with a feature request vary across systems. Some projects rely on general forums where feature requests are discussed in standard discussion threads; others rely on more dedicated collaboration and issue tracking platforms such as JIRA or IBM Jazz. In this paper, we study projects that manage feature requests using the Bugzilla issue tracking system. Bugzilla combines the reporting of bugs and feature requests; in this system, feature requests are distinguished from bug reports by being marked as 'enhancement requests'.

The typical life-cycle of a feature request is shown in the state transition diagram of Fig. 1. When first created, a feature request is in the state New. In this state, a decision needs to be made to either reject the request or assign it for development. When a decision is made to assign the feature for development, it becomes Assigned. The states Code Submitted and Verified denote states where code implementing the feature request has been submitted and verified respectively. Once verified, the code implementing a feature request can be integrated in the product. At different stages of its life-cycle, project managers may decide to reject a feature request, and this may happen before it is assigned to a developer for implementation, during implementation, or after the feature has been integrated in the product. We use the labels Immediate Reject, Abandoned, and Removed to distinguish these three cases. A rejected feature request can be reopened, meaning that it is again open to decide to either reject it or assign it for implementation. This model is not one that is followed rigorously or explicitly in the projects we have studied. It is rather a model that helps clarify the different conceptual states of feature requests and define the different failure types that can be associated with them across a range of feature request management systems.

The state of a feature request in this model can be inferred from meta-data in the Bugzilla issue tracker or similar systems. We explain this mapping for the KDE project[1]—a project concerned with the development of a unix graphical desktop environment. Each project has slightly different ways to label its bug reports, but the general approach is similar across projects. Feature requests in the KDE project correspond to "bug reports" whose severity is labelled as 'enhancement' to distinguish them from bug reports corresponding to system failures. A feature request is in the state 'New', 'Reopened', 'Verified', or 'Assigned' of our model if the status of the bug report has the corresponding value in Bugzilla. A feature request is in the state 'Integrated' if the bug report status is set to 'Resolved' and its resolution type is 'FIXED'. A feature request is in the state 'Rejected' if the bug report status is 'Resolved' and its resolution type is 'WONTFIX', 'INVALID', or 'WORKSFORME'. Our study and analysis of feature requests excludes bug reports marked as 'DUPLICATE' because when this happens, the duplicate request is abandoned and all discussions and decisions continue with the original request.

Project managers and developers are responsible for managing feature requests. This essentially involves making decisions on how to move a feature request forward in its life-cycle and updating its meta-data accordingly. Many projects deal with extremely large volumes of feature requests. The Firefox project contained 6,379 instances, 483 of which were active in the last three months of 2010; Eclipse contained 46,427 feature requests of which 5,155 were active in the same period. Discussions within threads are often quite lengthy, as exemplified in feature request #262459[2] of the Firefox project, which contains 64 posts, approximately 4,600 words and lasted four years. The sheer volume of this data makes managing a feature request management system very difficult for stakeholders, developers, and project managers alike.

2.2 Feature request management failures

Defects in the description of feature requests, such as ambiguities, inconsistencies, and omissions [27], may cause faults in the decisions to either reject a feature or assign it to a developer too early before the actual requirements on the feature are understood well enough. These faults may, in turn, cause different types of failures in the product or development process. It is the occurrence

[1] https://bugs.kde.org/page.cgi?id=fields.html.

[2] https://bugzilla.mozilla.org/show_id=262459.

of these failures that we wish to be able to predict before a decision is made to either accept or reject a feature. The different types of failures that can be potentially caused by faults in the decisions to reject or assign a feature can be defined in terms of a feature request's progress though the life-cycle in Fig. 1 and are the following:

2.2.1 Product failure

A feature request has a product failure if it is in the 'Integrated' state and has at least one confirmed bug report associated with it. This definition does not cover failures that are not reported so might be better understood as 'reported product failure'.

Feature request #451995 [3] of the Thunderbird project is an example of this failure. An archive for autosaved emails is implemented and integrated into the product after which two bugs, #473439 and #474848, are reported as having arisen from the introduction of the feature.

2.2.2 Abandoned development

A feature request is abandoned if it was once assigned for implementation, and the implementation effort has been cancelled before the feature is integrated into the product. These correspond to feature requests that are in the 'Abandoned' state in the model of Fig. 1. A subset of feature requests exhibiting this failure that have been abandoned after code has been submitted (i.e. those that were in state 'Code Submitted' or 'Verified' when rejected). We refer to such failures as '**Abandoned Implementation**'. Our studies and prediction models will focus on abandoned implementation as opposed to the more general case because this failure is more costly since effort has been spent developing implementations. Our study could, however, be extended to cover all abandoned development failures.

Feature request #34868[4] of the Apache project is an example of an abandoned implementation failure, in which a change to a server authentication service is suggested, assigned for implementation, and has two code components and an example developed for it. After this, it is discovered that it is not possible to reach a state when running Apache where this feature would be useful.

2.2.3 Rejection reversal

A feature request has a rejection reversal failure if it was once rejected before eventually being integrated into the product. Rejecting a feature request is an explicit decision to remove it from the backlog of feature requests that can be considered for implementation. This is different from a decision not to assign a feature request for implementation at the moment but leaving it in the backlog. We view the initial decision to wrongly remove a feature request from the backlog as the fault that causes this type of failure. This fault delays the introduction of features of value to stakeholders, can upset stakeholders, and cost them time and effort to argue for the feature request to be reopened.

Feature request #171702[5] of the Firefox project is an example of this failure. A developer rejects a feature request in the first post after it is suggested by simply stating that it is "Not part of the plan" without fully understanding the feature request's full meaning or eliciting stakeholders' needs. Subsequently, stakeholders become frustrated and effort is spent arguing the feature's merits. The feature is eventually re-opened, implemented, and integrated into the product.

2.2.4 Stalled development

A feature request has a stalled development failure when it remains in the 'assigned' state, and no code has been submitted for more than one year. The duration of 1 year is arbitrary; we have chosen it for our studies because it corresponds to a duration within which we would expect code to have been developed for feature requests in the projects studied. Our studies could easily be repeated with shorter deadlines for this failure type. If a feature request management system contains information on the estimated development time for each feature request, it could be used to detect 'late development' failures.

Feature request #49970[6] of the KDE project is an example of this failure. A fullscreen mode option is suggested and assigned for development in late 2005 and does not get fixed until mid 2007. In comment number 17, a developer states that the assigned developer had not been working for some time and apologies that the feature would not make it into the planned release.

2.2.5 Removed Feature

A feature request is removed if it has been rejected after having been integrated into the product. Feature requests rejected in this way signify that a decision was made to discard the feature before shipping the next release of the product. This is a failure because it is caused by the need to remove a feature that introduces undesirable behaviours for stakeholders. Such failures may be caused by insufficient

[3] https://bugzilla.mozilla.org/show_id=451995.

[4] https://issues.apache.org/bugzilla/show_id=34868.

[5] https://bugzilla.mozilla.org/show_id=171702.

[6] http://bugs.kde.org/show_id=49970.

upfront analysis of the feature request before its development.

Feature request #171465[7] of the Netbeans project is an example of this failure. A feature is implemented and integrated into the project, after which it is disabled due to a lack of support in the Netbeans framework for the feature. Finally, a developer states that the feature "Does not fit into our current strategy".

2.3 A study of failures in seven projects

We conducted a study assessing the frequency of failures in seven large-scale open-source projects. We developed a scraper in PHP that automatically traced the life-cycle of all feature requests in these projects and checked for failures against the rules that define them. The purpose of this study was to assess the degree to which failures can be automatically identified and determine their impact on projects. These projects were the Apache web server,[8] the Eclipse development environment,[9] Firefox web browser,[10] the KDE operating system[11] the Netbeans development environment,[12] Thunderbird email client,[13] and the Wikimedia content management system.[14] The projects were all large in size, ranging from 5,000 to 50,000 feature requests.

Table 1 shows occurrences of these failures recorded in the 7 projects. Failures were automatically identified from the meta-data of all feature requests within the projects from their start date until October 2012. The first four rows show the number of years for which the feature request management systems have been in use, the total number of feature requests that have been created in that period, and how many of these features have been assigned or rejected at some stage in their life-cycle. The following five rows show the rate at which failures occur in terms of *failure density*—the percentage of failures among all feature requests to which the particular failure type applies. For product failure, abandoned implementation, stalled development, and removed feature, this corresponds to the number of failures divided by the number of assigned feature requests; for rejection reversal, it corresponds to the number of failures divided by the number of rejected features. In the three cases, where the failure density is shown, no failures were automatically identified. These failure densities are visualised in Fig. 2.

The majority of the failure types identified have a relatively high densities, occurring in abundance within the seven projects. It is important to note, however, that the process used to identify these failures is not infallible as it relies on the users of feature request management systems updating their meta-data consistently.

Product failures will not have been caught using our approach if developers do not place a trace between feature requests and any bugs they may give rise to. In the KDE and Apache systems, for example, the user interface hides the means of placing such traces. The Firefox project, meanwhile, has a culture of maintaining consistency with these links. While many product failures were not caught using our approach, we found that a sample of 100 failures of this type over the 7 projects corresponded to cases that were clearly instances of this failure.

Automated detection of abandoned implementation failures relies on developers submitting code they have developed to the feature request thread. In the Eclipse project, for example, we found that there was a culture of working on code outside of the feature request management system, and therefore, fewer failures of this type could be detected. In the other projects, meanwhile, the feature request management system was the primary means of collaborating on code and a higher failure density for this failure was detected correspondingly.

Rejection reversal failures were caught consistently in these projects since updating a feature request's meta-data is the only means by which a developer can mark it as having been rejected. Features that eventually make it into the final software product will invariably also have their meta-data updated accordingly. One may argue whether all the instances identified are necessarily failures; rejecting the feature request may have been the right decision at the time it was made and the decision to reopen it may be due to new circumstances. Attention should be given to this caveat, and in the projects studied, we sampled 100 feature requests containing this failure and found that in all cases the initial rejection of the feature request appeared to be premature and wrong at the time it was made.

Accurate capture of stalled development failures depends heavily on whether code developed for the feature is submitted to the feature request's ticket and whether features that have been integrated into the product have their meta-data updated in a timely fashion. We found this to be the general case in all the projects apart from Eclipse, where most discussions and code development took place outside of the feature request management system through mailing lists, code repositories, and IRC channels. Further, we found that the Eclipse project did not have a culture of aiming to integrate features that have been assigned for development—a consequence of the project's loose reliance on the

[7] http://netbeans.org/bugzilla/show_id=171465.
[8] https://issues.apache.org/bugzilla/.
[9] https://bugs.eclipse.org/bugs/.
[10] https://bugzilla.mozilla.org/.
[11] https://bugs.kde.org/.
[12] http://netbeans.org/bugzilla/.
[13] https://bugzilla.mozilla.org/.
[14] http://bugzilla.wikimedia.org/.

Table 1 Feature requests and failures in large scale projects

	Apache	Eclipse	Firefox	KDE	Netbeans	Thunderbird	Wikimedia
Duration (years)	10.8	9.2	8.3	11.1	10.3	11.8	7.2
Assigned feature requests	354	7,564	382	1,800	2,152	199	824
Rejected feature requests	1,444	12,247	1,975	11,156	5,008	791	3,129
Total feature requests	5,242	46,247	6,379	55,349	24,167	5,150	13,797
Product failure (%)	–	0.1	21.7	–	2.3	14.1	0.2
Abandoned implementation (%)	4.0	0.8	6.1	1.3	1.4	3.5	4.5
Rejection reversal (%)	5.9	2.5	13.1	4.3	4.4	2.3	9.6
Stalled development (%)	26.1	60.6	12.9	26.0	34.3	14.1	15.6
Removed feature (%)	–	0.1	0.8	0.6	2.3	0.5	1.7

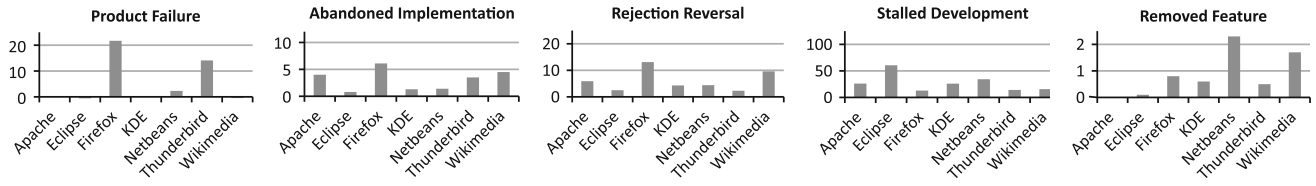

Fig. 2 Failure densities (%)

feature request management system for communicating and collaborating on feature development. Feature requests identified as a failure of this type in Eclipse might therefore not be considered as severe as in the other projects.

Increasing the period of inactivity after which we consider stalled development failure to occur would decrease the failure rate and visa versa. When assessing prediction models for this failure using the cost model, we describe in this paper a project manager can adjust the estimated cost of a failure occurring to reflect the damage that this period of inactivity does to a project.

Identification of removed feature failures is dependent on features removed from the software product having their meta-data updated to reflect the change. We found many examples across the projects studied where this was not done—once a feature request has been implemented developers tend to forget about the thread representing it and do not update its status when it is removed. Our figures are therefore likely to underestimate the true numbers of removed features. However, even if not all removed feature failures are detected and the density of detected failures is low (between 0.1 and 2.3 % in our studies), we believe that detecting and addressing them is valuable to a project because their costs—which includes wasted effort to fully implement, integrate, and then remove the feature—is so big.

One could question whether all removed features are really failures; integrating the feature into the product may have been a good decision at the time it was made, and the feature may have been removed later only because circumstances have changed and it was no longer needed. We have inspected all 86 instances of removed feature failures detected in the 7 projects, and we found that they do correspond to cases where a feature was removed because it conflicted with another feature, stakeholder needs, or the high level goals of the project. In all cases, the feature was removed shortly after it was integrated in the product (within a month). As explained above, features that are removed long after they have been introduced in the product tend to not have the status of their original feature request updated. If such status was updated, we might have found instances of removed feature that does not correspond to failures. To avoid including such good cases of removed feature, we could refine our rule for detecting removed feature failures by including only features that have been removed shortly after they have been integrated, for example 30 days.

An analysis of a sample of approximately 100 automatically identified instances of each failure type across the 7 projects confirmed that they correspond to failures as we have defined them, making the data a strong candidate for the construction of failure prediction models. The exception to this was stalled development failures in the Eclipse project for the reasons discussed above. Inconsistent updates in the meta-data of feature requests do lead to many failures not being caught using our approach. While prediction models constructed with missing data are consistent, we expect that more value could be obtained from predicting failures in projects like Firefox that have a strong culture of consistently updating their meta-data allowing for more instances of failure to be detected.

We have not included the Eclipse project in the prediction experiments that follow in this paper due to the poor quality of the data—inconsistent updating of the meta-data in this project meant that many failures were not identified for our training sets. Further, the feature request management system was not the primary means of collaboration meaning little data are available within the system to provide patterns for the generation of predictive models. We did attempt to generate failure prediction models for Eclipse to validate this claim and found them to perform poorly. Subsequently, we suggest that the approach described in the paper should not be considered for projects where communication, collaboration, and timely updates of meta-data are scarce in the feature request management system.

3 The value of early failure predictions

Some failures of the types characterized in the previous section may be caused, or at least partly caused, by defects in the requirements elicitation and analysis activities carried out in a feature requests discussion thread. Such defects include the equivalent of classic defect types of requirements documents, such as ambiguities, poor structuring, incompleteness, and inconsistencies, as well as process-related defects such as failing to involve the right stakeholders [27]. The purpose of an early failure prediction technique is to generate alerts for the feature requests that are at risk of failure so that the project stakeholders can take counter-measures such as resolving ambiguities or inconsistencies about a desired feature, or monitoring and guiding its later progress.

An early failure prediction model for a failure of type T is a function that generates an alert for each feature request that it believes will result in a failure of this type. Formally, if FR is a set of feature requests for which predictions are sought, the result of applying a prediction model is a set $Alert \subseteq FR$ denoting the set of feature requests that are predicted to result in failure.

The quality of such predictions can be assessed using the standard information retrieval measures of precision and recall. Let $Failure \subseteq FR$ be the set of feature requests that will actually have a failure of type T. This set is unknown at prediction time. The true positives are the alerts that correspond to actual failures; that is, we define the set $TruePostive = Alert \cap Failure$. The precision of a set of predictions is the proportion of true positives in the set of alerts, and its recall is the proportion of true positives in the set of all actual failures, that is

$$precision = \frac{|TruePositive|}{|Alert|} \quad recall = \frac{|TruePositive|}{|Failure|}$$

When designing a prediction model, there's an inherent conflict between these two measures: generating more alerts will tend to increase recall but decrease precision, whereas generating less alerts will tend to increase precision but decrease recall. An important question when designing and evaluating prediction models is to find the optimal trade-off between precision and recall.

A standard measure used in information retrieval for combining precision and recall is an f score, which corresponds to a harmonic mean between precision and recall. This score, however, relies on attaching an arbitrary importance to the two measures and has little meaning in our context.

We instead assess the relative weights to be given to precision and recall by assessing the costs and benefits to a project for a set of predictions. The model is deliberately simple to facilitate its use and comprehension. To use our model, a user need estimate only two parameters: P_s, which denotes the probability that additional upfront analysis on a feature request will be successful at preventing a failure of type T, and $\frac{C_f}{C_a}$, which denotes the relative cost of a failure of type T compared to the cost of the additional upfront analysis. Finer-grained cost-benefit models are possible but would require estimations for a more complex set of model parameters and thereby reduce our confidence in the results.

We evaluate the expected benefit of a set of predictions P as follows. Assuming that each failure of type T imposes a cost C_f to the project when it occurs, if we could prevent all failures for which an alert is generated, the benefit to the project would be $|TruePositive|.C_f$. We have to take into consideration, however, that not all additional upfront analysis will be successful at preventing a failure. If we assume that the probability of success of additional upfront analysis is P_s, then the expected total benefit of a set of predictions is the product $|TruePositive|.P_s.C_f$. Given that $|TruePositive| = |Alert|.precison$ we obtain the following equation characterizing expected benefit:

$$ExpectedBenefit = |Alert|.precision.P_s.C_f$$

We evaluate the expected cost of a set of predictions as follows. Assuming that each alert is acted upon and that the cost of additional requirements elaboration is C_a for each alert, the total expected cost associated with a set of predictions is given by the following equation:

$$ExpectedCost = |Alert|.C_a$$

The expected net value of a set of predictions is then given by the difference between its expected benefit and cost:

$$ExpectedValue = |Alert|.(precision.P_s.C_f - C_a)$$

Since $|Alert| = |Failure|.\frac{recall}{precision}$, the equation can be reformulated as:

$$ExpectedValue = |Failure|.\frac{recall}{precision}.(precision.P_s.C_f - C_a)$$

By simplifying and factoring C_a, the formula is expressed as:

$$ExpectedValue = C_a.|Failure| \cdot recall.\left(P_s.\frac{C_f}{C_a} - \frac{1}{precision}\right)$$

Difficulties lie in estimating absolute values for C_a and C_f, so instead we assume that the cost of additional upfront analysis provides the unit of measure and ask model users to estimate the relative cost of failure with respect to the cost of additional upfront action. This relative measure is often used in empirical studies about the cost of failures in software development projects, although there are caveats about the context of applicability of the different results [26]. The ratio of the cost of fixing a requirement defect after product release to fixing it during upfront requirements analysis is commonly cited to be between 10 and 100 for large projects [4, 21, 27], and between 5 and 10 for smaller projects with lower administrative costs [5, 21].

Fixing the cost of action to 1 gives our final formula characterizing expected value:

$$ExpectedValue = |Failure|.recall.\left(P_s.\frac{C_f}{C_a} - \frac{1}{precision}\right)$$

We have kept the term $\frac{C_f}{C_a}$ instead of simply C_f to make explicit that the cost of failure is relative to the cost of additional analysis. Since the number of failures is a constant for a given set of feature requests, we can compare alternative prediction models by assessing their expected value per failure:

$$\frac{ExpectedValue}{|Failure|} = recall.\left(P_s.\frac{C_f}{C_a} - \frac{1}{precision}\right)$$

In this formula precision and recall are characteristics of the prediction technique that can be estimated from its performance on past feature requests. If one wants to know the expected value per feature request, this can be obtained by multiplying the expected value per failure by the failure density $\frac{|Failure|}{|FR|}$—a constant that can be derived from past feature requests.

The parameters to be estimated by model users are P_s and $\frac{C_f}{C_a}$. The expected value actually depends on the product of these parameters denoted α, i.e. $\alpha = P_s.\frac{C_f}{C_a}$. We therefore obtain the following equation:

$$\frac{ExpectedValue}{|Failure|} = recall.\left(\alpha - \frac{1}{precision}\right) \quad (1)$$

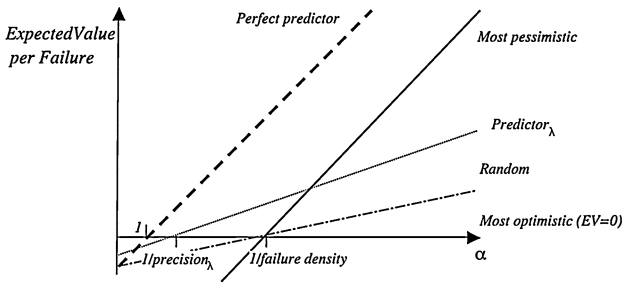

Fig. 3 Expected value per failure as a function of α

In principle, it might be possible to estimate P_s and $\frac{C_f}{C_a}$ empirically from past project data. However, such project specific data are rarely available. In the absence of this data, model users can estimate these numbers based on the general findings from the empirical studies reported in the previous subsection and assess the value of a prediction model for a range of α values rather than a single point. Considering product failures in the Firefox project, for example, if a model user estimates the ratio of the cost of failure to the cost of action to be between 50 and 100, and estimates the probability of success in preventing such failure by additional upfront analysis to be between 0.1 and 0.3, then the α values of interest will be between 5 (50*0.1) and 30 (100*0.3).

The advantage of assessing predictive models using equation 1 as opposed to more commonly used techniques such as an f-score is that the expected value has a clear meaning for the project and the parameters to be estimated for computing it are at least in principle measurable. We argue that it is more meaningful to ask model users to provide estimates for P_s and $\frac{C_f}{C_a}$ than asking them estimate the relative weights of precision and recall.

To understand equation 1 we can look at how the expected value per failure varies with α for some prediction models with known precision and recall, as shown in Fig. 3.

A perfect predictor would be one that generates alerts for all failures and generates no false alerts. Its precision and recall are both 1. We can observe that even for a perfect predictor, the expected value is positive only if $\alpha = P_s.\frac{C_f}{C_a} > 1$. This means that for any upfront activity to have a positive value, the ratio between the cost of the failure it may prevent and its own cost must be higher than the inverse of its probability of success in preventing the failure. For example, if additional upfront requirements analysis may prevent some failure type with an estimated probability of success of 0.1, the cost of this activity (e.g. the number of man-hours it takes) must be at least 10 times smaller than the cost of late correction of the failure it intends to prevent.

The most pessimistic predictor is one that generates alerts for all feature requests. Its recall is equal to 1 and its precision to the failure density. The most optimistic predictor is one that never generates any alerts. Its recall is 0, precision 1, and expected value is therefore always null. The random predictor is one that randomly generates alerts for feature requests using the failure density for past feature requests as the probability of alert. We can observe from equation 1 that the best of these three baseline predictors is the most optimistic predictor when α is less than the inverse of the failure density; it is the most pessimistic predictor when α is more than the inverse of the failure density. The random predictor is always outperformed by one of these two. This means that, unlike other failure prediction models that compare themselves against a random predictor [6, 23, 31], the baselines in our context are the most optimistic and most pessimistic predictors.

Figure 3 also shows the expected values for a predictor λ whose precision is higher than the failure density. Such a predictor outperforms the most optimistic baseline when $\alpha < \frac{1}{precision_\lambda}$ and it outperforms the most pessimistic one when α is below some value α_x (where α_x can be determined to be $\frac{1}{1-recall_\lambda} \cdot \left(\frac{1}{FailureDensity} - \frac{recall_\lambda}{precision\lambda}\right)$. When α is outside of this range—either below or above it—the most optimistic or pessimistic predictors have better expected values. This provides a quantitative justification for the intuition that for a given set of failure predictions with imperfect recall and precision, if the relative cost of failure and the probability of success of additional upfront analysis in preventing the failure are low, then it is more cost-effective not to do any additional analysis; whereas if the cost of failure and probability of success are high, then it is more cost-effective to perform additional analysis on all feature requests.

If a predictor has a precision that is less than the failure density, it is always outperformed by the most optimistic or most pessimistic predictors. Our objective when developing prediction models will therefore be to generate predictors whose precision is higher than the failure density and whose range of α values for which it outperforms the most optimistic and most pessimistic predictors is as large as possible.

4 Predicting failures

The class of machine learning techniques that apply to our problem, known as classification algorithms, first constructs prediction models from historical data and then uses these models to predict classifications for new data. In our case the historical data used to generate prediction models are past feature requests, while the new data consist of feature requests that are about to be assigned or rejected on which we wish to predict future failures.

We have developed a tool-supported framework that generates alternative prediction models from historical data sets. These models vary according to the characteristics of feature requests discussions they take into account for predicting failures and the classification algorithms they rely on for constructing prediction models.

4.1 Generating a prediction model

To generate a single prediction model, a user of our framework must specify the following inputs:

- The feature request management database to be used and the failure type to be predicted.
- The classification algorithm to be used. In our experiments, alternative models were generated using the Decision Table, Naive Bayes, Linear Regression, and M5P-Tree algorithms, which constitute a good representation of the different types of classification algorithm [30]. The Decision Table and Naive Bayes algorithms are the least computationally complex, but have been known to perform well for less computational effort. Decision table constructs a simple decision tree to predict failures from the attributes, and Naive Bayes computes a median value of each attribute for historical failed and non-failed feature requests and matches a new feature request's attributes to the case that they lie most closely to. Linear regression, meanwhile, performs a full correlative analysis by constructing a set of 'best fit' linear functions from historical data to predict failures from attributes. Lastly, the M5P-Tree algorithm creates a decision tree to group feature requests with attributes that result in similar probabilities of failure and then generates a linear regression prediction model for each node. The computational complexity is usually reduced when compared to linear regression since each node typically generates a simpler correlative function.
- A *predictive attribute* of feature requests used to train a predictive model. In our experiments, we use the following 13 attributes that can be automatically extracted to generate a prediction model for each failure type in each project:
  - Attributes relating to discussion participants: the number of participants, the number and percentage of posts by the person who reported the feature request, the number and percentage of posts made by the person who is assigned to develop the feature request.
  - Attributes relating to the structure and development of the discussion: The total number of posts, the

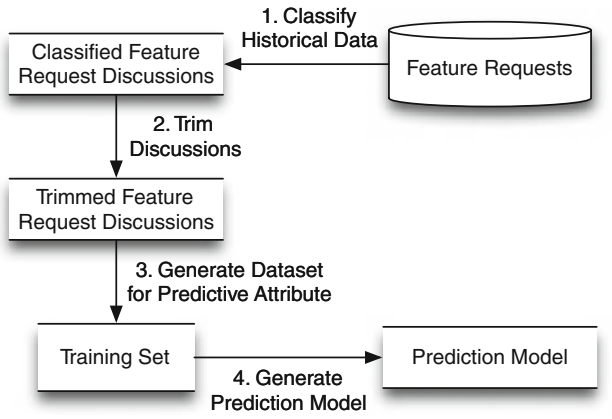

Fig. 4 Framework for generating a prediction model

number of words in each post, the number of words in the whole discussion, the number of code contributions submitted to the ticket during the discussion, the time elapsed between posts, and the total time elapsed in the discussion.

- Textual attributes: Bag of words and term frequency in document frequency (TFIDF), which are two approaches for finding patterns in the textual content of discussions. Bag of words assigns an attribute to each unique word in a feature request with a value corresponding to the number of occurrences of the word. TFIDF, meanwhile, uses the same process, but multiplies this value by the 'uniqueness' of each word (the inverse of the number of occurrences of the word across all feature requests in the historical data set).

- An estimated value for α used by our framework to make trade-offs between precision and recall in predictive models, and to compare the expected value of these models to the baselines for validation.

Using these inputs, our automated failure prediction framework follows the process shown in Fig. 4 performing the following steps to construct a failure prediction model:

4.1.1 *Classify historical data*

A large data set is extracted from the feature request management forum. These data in its raw form include the textual content of discussions, their structure, their associated meta-data, and the history of changes made to this meta-data. To build a prediction model for rejection reversal failures, we extract all feature requests that have been rejected at some point in their lifetime, and for the other four failures we extract all those that have at some time been assigned for implementation. Each feature request is then automatically identified as a positive or negative instance of failure with respect to the rules defined in the Sect. 2.

4.1.2 *Trim discussions*

Posts within each feature request are trimmed to the point in its life-cycle where we want to make predictions. This will be just before it was rejected for the rejection reversal failure, and just before it is assigned for the other four types of failure.

4.1.3 *Generate data set for predictive attribute*

The trimmed discussions are then transformed into a data set that contains the relevant information about the attribute of interest and in a format that can be processed by classification algorithms. For example, if the attribute of interest is the number of participants in the discussion, this set will generate a set of tuples ⟨feature request, number of participants, classification⟩.

4.1.4 *Generate prediction model*

The data set is then used to generate a prediction model using the selected classification algorithm. If the classification algorithm is one that directly classifies a feature request as a failure or not, such at the Decision Table and Linear Regression algorithms, then no further steps are needed. Other classification algorithms such as the Naive Bayes and M5P-Tree algorithms assign to each feature request a numerical scores between -1 and $+1$, indicating whether the feature is more likely to be a failure (if its score close to 1) or not (if close to -1). In these cases, we still need to set a threshold such that all feature requests whose score exceeds the threshold generate an alert. Our framework sets this threshold automatically by estimating the model precision, recall, and expected value (using tenfold cross validation as described in the next subsection) for a range of threshold values and selecting the threshold that yields the highest expected value. For example, Fig. 5 shows how the recall, precision, and expected value vary with this threshold for a set of predictions made on product failures in the Firefox project and an estimated α of 3. In this case, a threshold of 0.36 will be chosen. In cases where a prediction model cannot provide a positive value the threshold is automatically set so as to generate no alerts, thus mimicking the most optimistic baseline. Conversely, in cases where the highest value is provided by generating all alerts, the most pessimistic baseline is mimicked.

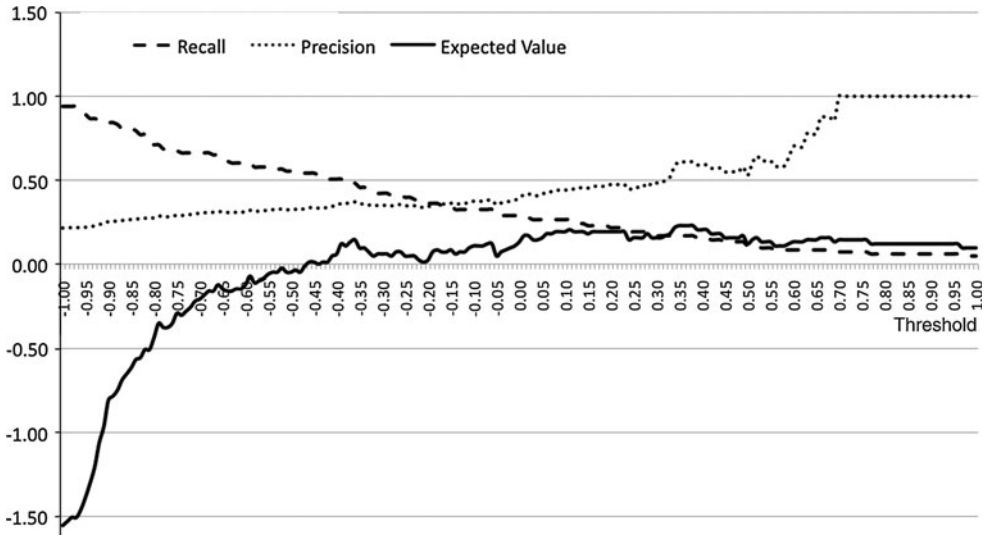

Fig. 5 Recall, precision and expected value as a function of alert threshold

4.2 Evaluating a predictive model

A standard technique for evaluating prediction models consists in performing a tenfold cross-validation [30]. The data set is split into a training set composed of 90 % of feature request selected at random and a testing set composed of the remaining 10 %. A prediction model is built using the training set, and its precision and recall computed for the testing set. This experiment is repeated 10 times with different training sets and testing sets. The precision and recall of the prediction model obtained from the full data set are then estimated as the mean of the precision and recall for the 10 experiments. Equation 1 in Sect. 3 is then used to evaluate the prediction model's expected value from its precision, recall, and assumed α value. The results of such evaluation are reported in Sect. 5.

4.3 Intueri: an implementation

We have developed a research tool, Intueri, that allows users to generate and evaluate predictive models using the framework described in this section. Intueri's front-end is developed in Flash enabling it to be run in a web browser, while it's back-end is powered by ActionScript making use of PHP to communicate with other components and to extract, load, and save data. Data can currently be extracted from Bugzilla-based feature request management systems and is converted to an XML format. The generation of prediction models makes use of the open-source Weka libraries [15], which provide an interface to many classification algorithms. Results are currently stored and analysed in Matlab.

The tool is available online[15] alongside the data we have used for the experiments in the following section. This includes the PHP files used to extract raw data from feature request management systems and automatically identify failures, raw discussion extractions from feature request management systems in XML format, the training sets that can be given as input to the WEKA tool to generate prediction models, and the full results of our experiments.

[15] http://sre-research.cs.ucl.ac.uk/Intueri/.

5 Prediction experiments

We have applied our early failure prediction framework to the seven large-scale open-source projects on which we conducted the failure study in Sect. 2 We do not report here the results from the Eclipse project due to the poor quality of the data it contains as discussed in the Sect. 2 The questions that we wished to answer with our experiments are the following:

1. What classification algorithms, among the four we envisaged, generate the most valuable prediction models?
2. Can feature request failures of the types defined in Sect. 2.2 be predicted from early discussions before the feature request is either assigned or rejected? What expected value can a project hope to obtain from actioning such predictions? Are some of failure types more susceptible to being predicted from early discussions? Are certain types of projects more susceptible to early failure prediction approach?
3. What attributes of early feature requests discussions, if any, can be used as reliable predictors of later failures? Do some attributes perform better for certain types of failures and across projects and failure types?

To answer the first question, we have evaluated the performance of each classification algorithm for all failure

types and all of our 13 feature requests attributes on the data set for the Firefox project only. The experiment revealed that prediction models generated by the Decision Table and Naive Bayes classification algorithms failed to yield meaningful predictions. This might be expected as these algorithms perform a comparatively low amount of correlative analysis on data. The M5P-tree and Linear Regression algorithms, meanwhile, consistently produced similar results in terms of expected value. In all cases, however, the less computationally complex M5P-tree algorithm generated prediction models significantly faster than the Linear Regression approach—in the order of minutes as opposed to hours. For remainder of the experiments, we therefore only used the M5P-tree algorithm.

To answer the second set of questions we have generated and evaluated prediction models for all failure types and predictive attributes in all six projects. The data sets of failures identified in Sect. 2 were used to train these models. Table 2 summarises the results by presenting for each project and failure type the prediction model that yielded the most expected value. That which gave the most expected value for product failure predictions in Firefox, for example, was generated using the 'number of posts in the discussion' attribute and predicted feature requests with more than 12 posts to fail. The table also shows the estimated α value we have used for setting the alert threshold and computing the expected value. When the best prediction model is shown as being the most pessimistic or most optimistic this means that there was no predictive model was generated which performed better than these baseline predictors. In such cases, as you will recall from Sect. 4, the alert threshold for prediction models is automatically set so that it behaves as the best baseline predictor. Fig. 6, meanwhile, shows the expected values per 100 feature requests for the predictive models, and Fig. 7 shows their respective precision and recall.

The results show that it is possible to generate early failure prediction models that provide positive expected value to a project. Rough estimations of the real value expected from actioning predictions can also be made. As you will recall from Sect. 3 the cost of an action was fixed to 1 in our equation, and we can therefore multiply the expected value by an estimation of the real cost of an action in dollars or man-hours to quantify it. For example, if for product failures in Firefox the cost of failure is estimated to be 10 times that of an action and the probability of success is 30 % then acting on failure predictions with a recall of 0.54 and a precision of 0.42 our equation gives us an expected value of $0.54.\left(0.3.\frac{200}{20}-\frac{1}{0.42}\right)=0.33$ per failure. Multiplying this result by an estimated cost of action of 10 hours gives us a quantified value of 3.3 hours saved per failure.

The results suggest that on these six projects our approach is more effective at predicting stalled development and abandoned implementation failures than the other failures. A possible explanation for this is that these failures have higher densities than the others and that they occur earlier (closer to the time of prediction), than removed feature and product failures.

While the Netbeans project could benefit the most from the prediction models constructed, there is no general case for some projects being more susceptible to early failure predictions than others. Our hypothesis that the Firefox project would gain more since it updated its meta-data more consistently was not confirmed.

The recall and precision of the models demonstrate how the cost-benefit equation is used to make the trade-off between these measures by maximising the expected value with respect to the failure type. For abandoned implementation failures, for example, preference is given to recall over precision since the high estimated α value weights the benefits of taking action to avoid values to be far more than the costs. The expected value can also show us where prediction models might be able to provide value to a project in cases where a low precision and recall are achieved, such as for rejection reversal failure predictions in the KDE project.

The α-values we have used for the different failure types have been estimated by ourselves and represent our best informed guesses based on general empirical studies of software projects [4, 5, 21, 27]. In the absence of empirical validation specific for each project, these values are certainly subject to debate. A benefit of our evaluation framework is that it is possible to assess how deviation between the estimated value for α and its real (unknown) value will impact the expected value of a prediction model. Figure 8, for example, shows how the expected value for different product failure prediction models in Firefox (for which α was set to 3 to determine the alert threshold) vary with α. We can see from the figure that the prediction model based on the number of posts in the discussion still performs better than the most pessimistic predictor if the real value for alpha goes up to about 6.5, but above that point the most pessimistic predictor (i.e. the one that suggests performing additional upfront analysis on all feature requests) yields a higher value. Similar graphs resulted from the other cases in which predictive models yielded more expected value than the baselines.

Table 3 helps us to answer the third set of questions. For each of the 13 predictive attributes, the number of projects in which a prediction model was generated that outperformed the baselines is shown each failure type. The numbers shown in brackets are the average expected value

Table 2 Prediction models with best expected values

	Product failure	Abandoned implementation	Rejection reversal	Stalled development	Removed feature
$\frac{C_f}{C_a} * P_s = \alpha$	10*0.3 = 3	50*0.5 = 25	10*0.5 = 5	10*0.5 = 5	20*0.2 = 4
Apache	–	Code contributions	Words per discussion	Code contributions	–
Firefox	Posts in discussion	(*Most pessimistic*)	(*Most optimistic*)	Code contributions	(*Most optimistic*)
KDE	–	Posts by reporter	Bag of words	Bag of words	(*Most optimistic*)
Netbeans	(*Most optimistic*)	TF-IDF	Words per discussion	Per cent by assignee	Bag of words
Thunderbird	Code contributions	Bag of words	(*Most optimistic*)	Bag of words	(*Most optimistic*)
Wikimedia	(*Most optimistic*)	Bag of words	Number of participants	Bag of words	(*Most optimistic*)

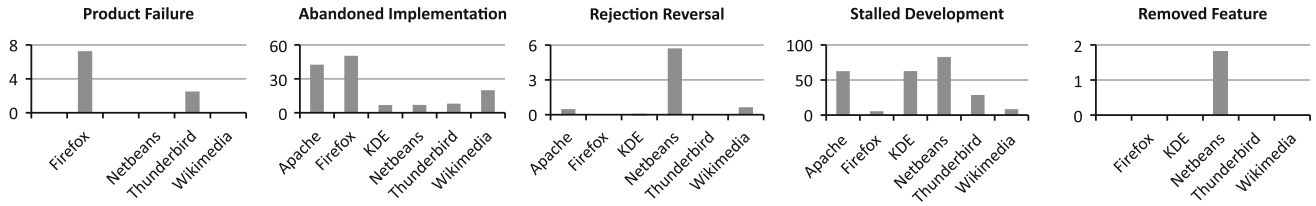

Fig. 6 Expected value per 100 feature requests from best predictive models

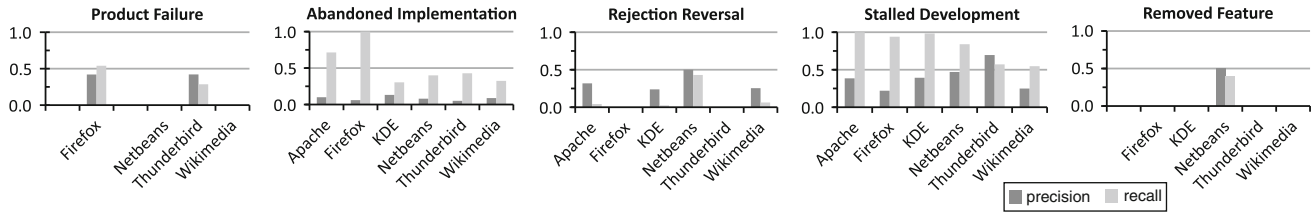

Fig. 7 Precision and recall from best predictive models

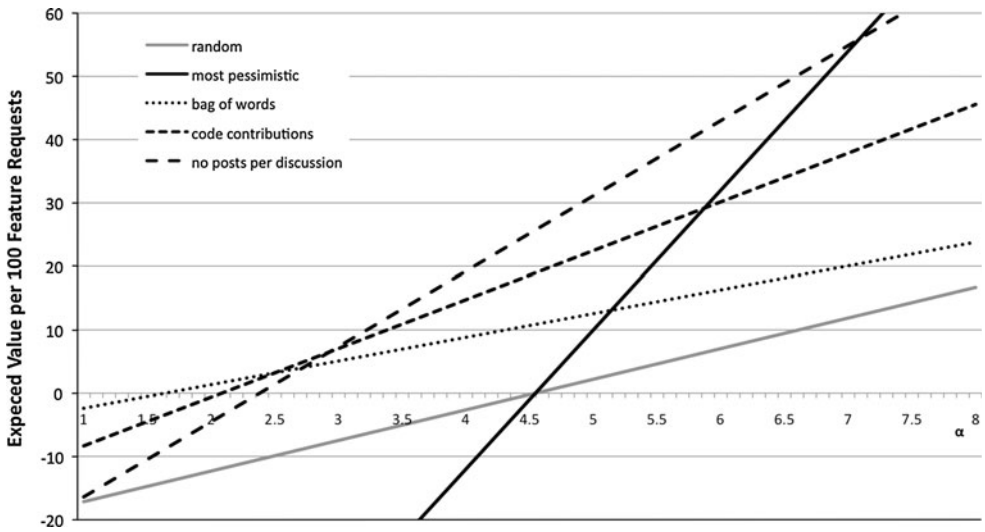

Fig. 8 Expected value as a function of α for product failure prediction models in the Firefox project

of these models. The code contributions predictive attribute, for example, generated stalled development prediction models that performed better than the baselines in 5 of the 6 projects studied, and the average expected value of these models was 59.61. Expected values should not be compared across failure types due to the differences in their units.

Interestingly, the predictive attributes that performed consistently well on all failure types for both projects are the two text-based attributes: bags-of-words and TFIDF. This suggests that analysing the actual content of the discussion, even at a very rudimentary level, could provide more reliable predictions than analysing attributes such as

Table 3 Which predictive attributes perform well for which failure types?

Predictive attribute	Product failure	Abandoned implementation	Rejection reversal	Stalled development	Removed feature	Total viable models
Number of participants	1 (0.82)	1 (7.20)	3 (0.21)	4 (53.86)	0 (0.00)	9
Posts by reporter (#)	1 (0.07)	4 (10.78)	1 (0.01)	2 (43.58)	0 (0.00)	8
Posts by reporter (%)	2 (0.87)	1 (7.20)	2 (0.02)	4 (52.92)	0 (0.00)	9
Posts by assignee (#)	2 (1.09)	1 (7.20)	0 (0.00)	4 (56.61)	0 (0.00)	7
Posts by assignee (%)	0 (0.00)	1 (9.63)	1 (0.05)	4 (56.65)	0 (0.00)	6
Posts in discussion	2 (1.18)	4 (9.33)	2 (0.60)	4 (55.49)	0 (0.00)	11
Words per discussion	2 (0.75)	4 (10.04)	1 (0.82)	5 (55.16)	0 (0.00)	11
Words per post	2 (0.87)	5 (13.38)	3 (0.28)	2 (42.41)	1 (0.01)	13
Code contributions	2 (1.35)	2 (13.28)	0 (0.00)	5 (59.61)	0 (0.00)	9
Time elapsed per post	0 (0.00)	5 (10.51)	3 (0.13)	6 (53.78)	1 (0.04)	15
Time elapsed per discussion	1 (0.73)	1 (7.20)	1 (0.35)	3 (52.08)	1 (0.04)	7
Bag of words	1 (0.86)	6 (15.55)	2 (0.73)	6 (63.36)	1 (0.26)	17
TF-IDF	1 (0.36)	4 (14.60)	3 (0.89)	6 (61.86)	1 (0.20)	14

the number of persons involved in the discussion and the discussion length. The actual words that have the highest influences on whether a feature request is classified as a failure or not are surprisingly simple words such as "would", "like", and the product name, for example, "Firefox". Unfortunately, this doesn't provide useful insights into how to write better feature requests to reduce the true risks of failures. One should not expect to obtain useful insights from predictive models using very simple natural language characteristics related to word occurrences. Richer language-based analysis based on sentence structures and the presence of specific keywords and phrases (e.g. typical ambiguous phrases or typical phrases that reveal the presence of rationale such as "so that" or "in order to") may give better and more useful failure prediction models. This is an interesting avenue for future research.

No predictive attribute always outperformed the others for a specific failure type in all the projects studied. When generating a predictive model for a new project, therefore, a range of predictive attributes could be evaluated as we have done in our experiments to find that which gives the highest expected value.

A much wider range of attributes than the ones we have used in our experiments could be tested for early failure predictions. Some of these attributes could be more complex than the ones we have used here, such as for example an analysis of the communication structure between stakeholders [31], the roles of the users involved in a discussion, or the system components affected by a feature request. We have also performed experiments where we generated and evaluated prediction models from combinations of the attributes and found that the results were not significantly improved and in some case even performed worse than when the attributes were taken in isolation. Combining these attributes more intelligently, however, such as via the use of a feature selection method [14] could yield better performance.

In the introduction, we raised the hypothesis that some of the failures are caused by defects in the feature request descriptions and discussions such as ambiguities, inconsistencies, and incompleteness. Our experiments do not test this hypothesis because information about such defects is not explicitly present in the feature request system. It would be interesting, as an extension to this work, to try generating predictive models from requirements defects. One way to make information about requirements defects available would be to extend feature request management systems to allow its users to explicitly tag feature request descriptions with defect categories [10], another would be to use the outputs from natural language requirements analysis tools [12, 17, 28]. We intend to explore both approaches in future work.

6 Related work

There is a large volume of work on predicting defects, and failures at later stages of development: techniques have been proposed to predict the location of code defects from source code metrics [23, 32], to predict build failures from the communication structure between system developers (who communicated with who) [31], and to predict the system reliability from test case results [22]. By contrast, in this paper we study the extent to which it is possible to predict failures much earlier in the development process from characteristics of online feature request discussions before a decision is made to implement or reject the feature. Further, our automated predictions extend the usual product

failures to cover process level failures as well. The precision and recall measure we obtain in our experiments (ranging approximately between 0.3 and 0.9 recall and between 0.2 and 0.7 precision) are of the same order as those obtained by the later predictions, such as in Wolf et al.'s [31] paper predicting build failures from the communication structure in bug tracking systems at build time with a recall between 0.55 and 0.75 and a precision between 0.5 and 0.76. We cannot, however, draw solid conclusions from such comparisons as they are based on studies involving different projects, failure types, and failure densities.

Failure predictions earlier in the life-cycle can be made using a causal model that aims at predicting the number of defects that will be found during testing and operational usage based on a wide range of qualitative and quantitative factors concerning a project (such as its size, the regularity of specification reviews, the level of stakeholder involvement, the scale of distributed communication, programmer ability, etc.) [8, 20]. In contrast, we aim at predicting failures in projects that may have a less disciplined approach than those for which this causal model has been designed, and we aim to be able to identify which specific feature requests are most at risk rather than predicting the number of failures. Further, the use of a classification algorithm is fully automated and does not require the human expertise needed to build the causal model.

The large volumes of data held in many feature request management systems lends itself to the use of machine learning approaches to address other requirements challenges. Clustering of similar feature requests has been proposed to help users find related feature requests [6], and a tool-supported approach for partially automating the process of triaging and prioritizing bodies of feature requests has been developed [19].

There is a large body of work on predictive techniques that, like our approach, use a cost model for making trade-offs between precision and recall when the loss associated to false positive and false negatives are asymmetric [13, 16]. These papers consider general techniques for developing and evaluating predictive models in this context. Our work is an application of such techniques for early failure predictions on feature requests. Our approach uses the cost model *a-posteriori* to find an optimal trade-off between precision and recall. In some contexts, methods that take cost into account during the construction of the prediction model have been shown to perform better than a-posteriori approaches [1]. Such a technique might be used to try to improve our results.

7 Conclusion

Early failure prediction can provide a useful, practical framework to reason about the amount of upfront analysis to perform on a feature request before deciding whether or not to implement it. Such techniques might also find their use in more traditional document-based requirements engineering processes.

We have defined different failure types that can be associated with feature requests, we have explored how accurately these failures can be automatically traced in feature request management systems, we have presented a cost-benefit equation for evaluating early failure predictions models, we have presented a tool-supported framework for the development and evaluation of such models, and we have reported the results of experiments evaluating failure prediction models using a range of predictive attributes for 7 large-scale projects. These experiments provide evidence that an early failure prediction approach can benefit projects by guiding upfront requirements analysis.

We believe that significant improvements in the performance of prediction models can still be achieved. This could be done by using a richer set of predictive attributes than the restricted set that has been successfully used in this paper. These include the presence of rationale in a feature request discussion (which could be indicated by common intentional phrases), advanced natural language processing techniques [25], the presence of manually [10] or automatically detected requirements defects [12, 17, 28], the architectural components potentially affected by a feature request, or the roles, expertise, and communication structure of the persons involved in the discussion. Techniques also exist for manipulating and cleaning a data set before passing it to a classification algorithm to generate a prediction model that could improve accuracy, for example, feature selection [14]. Directly integrating the cost model from Sect. 3 into a classification algorithm, as opposed to using it to select an alert threshold, could also yield more expected value.

It is our hope that early failure prediction techniques will allow project managers to make better informed decisions about the kind and amount of upfront requirements analysis to perform on incoming feature requests. If successful, such a technique is likely to change how people perform requirements analysis activities in a feature request management system. The early failure prediction models will therefore need to have the ability to adapt to such changes, for example through the use of online learning techniques [29].

References

1. Abrahams, AS, Becker A, Fleder D, MacMillan IC (2005) Handling generalized cost functions in the partitioning optimization problem through sequential binary programming Data Mining. In: Fifth IEEE international conference on, 2005

2. Berry D, Damian D, Finkelstein A, Gause D, Hall R, Wassyng A et al. (2005) To do or not to do: If the requirements engineering payoff is so good, why aren't more companies doing it? In: International conference on requirements engineering, 2005
3. Bird C, Pattison D, D'Souza R (2008) Latent social structure in open source projects. In: ACM SIGSOFT international symposium on foundations of software engineering, 2008
4. Boehm B, Papaccio P (2002) Understanding and controlling software costs. In: IEEE transactions on software engineering
5. Boehm B, Turner R (2003) Balancing agility and discipline: a guide for the perplexed. Addison-Wesley Professional, New York
6. Cleland-Huang J, Dumitru H, Duan C, Castro-Herrera C (2009) Automated support for managing feature requests in open forums. ACM Commun 52:68–74
7. Damian D (2004) RE challenges in multi-site software development organisations. Int Conf Requir Eng 8:149–160
8. Fenton N, Neil M, Marsh W, Hearty P, Ł Radliński, Krause P (2008) On the effectiveness of early life-cycle defect prediction with bayesian nets. Empir Softw Eng 13(5):499–537
9. Fitzgerald C (2009) Support for collaborative elaboration of requirements models. Internal UCL report
10. Fitzgerald C (2012) Collaborative reviewing and early failure prediction in feature request management systems. UCL doctoral thesis
11. Fitzgerald C, Letier E, Finkelstein A (2011) Early failure prediction in feature request management systems. International Conference on Requirements Engineering, pp 229–238
12. Gnesi S, Lami G, Trentanni G (2005) An automatic tool for the analysis of natural language requirements. Comput Syst Sci Eng
13. Granger C (1969) Investigating causal relations by econometric models and cross-spectral methods. Econometrica
14. Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. J Mach Learn Res
15. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten I (2009) The weka data mining software: An update. ACM SIGKDD Explor Newslett 11:10–18
16. Kaufhold J, Abbott J, Kaucic R (2006) Distributed cost boosting and bounds on mis-classification cost. In: IEEE computer society conference on computer vision and pattern recognition, vol 1. pp 146–153
17. Kiyavitskaya N, Zeni N, Mich L, Berry D (2008) Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. Int Conf Requir Eng 1:146–153
18. Laurent P, Cleland-Huang J (2009) Lessons learned from open source projects for facilitating online requirements processes. In: Lecture notes in computer science, requirements engineering: foundation for software quality, vol 5512. pp 240–255
19. Laurent P, Cleland-Huang J, Duan C (2007) Towards automated requirements triage. Int Conf Requir Eng, pp 131–140
20. Madachy R, Boehm B (2008) Assessing quality processes with ODC COQUALMO. In: Lecture notes in computer science, making globally distributed software development a success story
21. McConnell S (2004) Code complete, vol 2. Microsoft Press, Washington
22. Musa J (2004) Software reliability engineering: more reliable software, faster and cheaper. Tata McGraw-Hill, New York
23. Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: International conference on software engineering
24. Nuseibeh B, Easterbrook SM (2000) Requirements engineering—a roadmap. In: ICSE: Future of SE Track. pp 35–46
25. Sebastiani F (2002) Machine learning in automated text categorization. ACM Comput Surv 34:1–47
26. Shull F, Basili V, Boehm B, Brown A, Costa P, Lindvall M, Port D, Rus I, Tesoriero R, Zelkowitz M (2002) What we have learned about fighting defects. In: IEEE symposium software metrics
27. van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley, New York
28. Verma K, Kass A (2008) Requirements analysis tool: a tool for automatically analyzing software requirements documents. In: Internationalsemantic web conference
29. Widmer G, Kubat M (1996) Learning in the presence of concept drift and hidden contexts. Mach Learn 23:69–101
30. Witten I, Frank E (2005) Data mining: practical machine learning tools and techniques. Morgan Kaufmann Pub, Cambridge
31. Wolf T, Schroter A, Damian D, Nguyen T (2009) Predicting build failures using social network analysis on developer communication. In: IEEE international conference on software engineering
32. Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: International workshop on predictor models in software engineering