Early Failure Prediction in Feature Request Management Systems

Camilo Fitzgerald
Dept. of Computer Science
University College London
London, United Kingdom
c.fitzgerald@cs.ucl.ac.uk

Emmanuel Letier
Dept. of Computer Science
University College London
London, United Kingdom
e.letier@cs.ucl.ac.uk

Anthony Finkelstein
Dept. of Computer Science
University College London
London, United Kingdom
a.finkelstein@cs.ucl.ac.uk

***Abstract*—Online feature request management systems are popular tools for gathering stakeholder requirements 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.**

The paper presents a tool-implemented framework that automatically constructs failure prediction models using machine-learning classification algorithms and compares the performance of the different techniques for the Firefox and Netbeans 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 some failure types and projects. This suggests that automated failure prediction during online requirements elicitation may 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.**

I. Introduction

An increasing number of software development projects rely on online feature request management systems to elicit, analyse, and manage users' change requests [2][7]. Such systems encourage stakeholder participation in the requirements engineering process, but also raise many challenges [12]: 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 the feature request 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 requests management systems could be improved: techniques have been proposed to cluster similar threads of discussions to detect duplicates [6] and to facilitate the requirements prioritization process [13]; 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 [1].

Our objective in this paper is different: instead of detecting *defects* in feature requests, we wish to predict *failures*, i.e. 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 analysis on the feature request before making their decision. They can also use this information to monitor high-risk features more closely during their development.

There is a large volume of work on predicting failures at later stages of development: techniques have been proposed to predict run-time failures from source code metrics[16][23], to predict build failures from the communication structure between system developers (who communicated with who) [22], and to predict the system reliability from test case results [15]. Our objective in this paper is to 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.

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]. 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 of the causal model.

Our specific contributions are the following:

1) We define failure types that can be associated with feature requests. 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 [16][22] to include process failures such as abandoned development, 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, unlike what is traditionally done, the random predictor (a predictor that generates failure alerts at random based on the failure density) is not an appropriate baseline for comparing early failure prediction methods.
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 [10] to train and evaluate prediction models using a variety of classification algorithms and predictive attributes.
4) We report experiments in which early failure prediction methods for different failures are evaluated using alternate classification algorithms and predictive attributes for two large scale open source projects - Firefox and Netbeans. 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 during early stage discussions on a feature request with some of these simple attributes. The results also provide us with information about the classification algorithms and predictive attributes that work best in this context.

These results show promise for the early failure prediction approach. Together with the tool-supported framework and evaluation model they provide the basis for future work into the use of more elaborate predictive attributes and experimental datasets to further explore the approach and generate models that increase the accuracy of predictions and yield more value.

II. Failures in Feature Request Management Systems

A. 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 requests [6][12]. 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 varies 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 three large scale open source projects, Firefox[1], Netbeans[2] and Eclipse[3], that manage feature requests using the Bugzilla issue tracking

[1]`https://bugzilla.mozilla.org/`
[2]`http://netbeans.org/community/issues.html`
[3]`https://bugs.eclipse.org/bugs/`

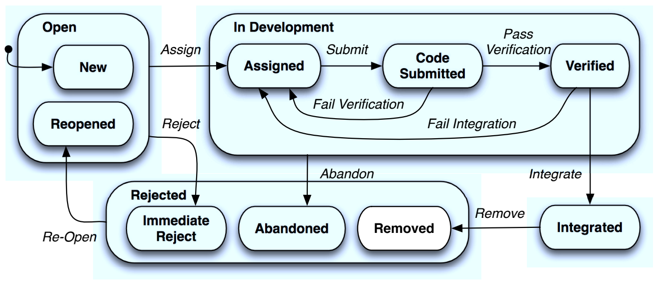

Figure 1. Life-cycle of a Feature Request

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 Figure 1. When first created, a feature request is the state New. In this state, a decision needs to be made to either reject the request or assign it for development. When 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, 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 decisions to either reject it or to 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. The state of a feature request in this model can be inferred from meta-data in the Bugzilla issue tracker or similar systems. In practice, users of feature request management systems will not always be consistent in updating the status of feature requests.

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, for example, contained 6,379 instances up to December 2010, 483 of which were active in the last three months of 2010; Eclipse contained 46,427 feature requests of which 5155 were active in the same period. Discussions within threads are often quite lengthy, as exemplified in feature request #262459[4] of the Firefox project, which contains 64 posts, approximately 4600 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.

B. Feature Request Management Failures

Defects in the description of feature requests, such as ambiguities, inconsistencies, and omissions [20] 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 project failures. It is the occurrence 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 are the following.

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'.

Abandoned Development - A feature request is abandoned if it was once assigned for implementation and the implementation effort has been cancelled before the feature has been integrated in the product. These correspond to feature requests that are in the 'Abandoned' state in the model of Figure 1. A subset of the feature requests exhibiting this failure are those 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.

Rejection Reversal - A feature request has a rejection reversal failure if it was once rejected before eventually being integrated into the product. We view the initial decision to reject a feature that is later integrated in the product as the fault that causes this failure. One may argue whether this is necessarily the case; rejecting the feature request may have been the right decision at the time it was made and the decision to reopen and assign the feature request may be due to new circumstances that did not exist at the time of the first decision. In practice, however, all rejection reversals we have seen in the projects studied correspond to cases where the initial rejection of the feature request can be argued to be premature and wrong at the time it was made. We view such rejections as faults because they delay the introduction

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

of features of value to stakeholders, can upset stakeholders, and cost them time and effort to argue for the feature request to be reopened.

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 all feature requests in the three projects. 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, this information could be used to detect 'late development' failures.

Removed Feature - A feature request is removed if it has been rejected after having been integrated into the product. This is a failure because it may be caused by the need to remove a feature that introduces undesirable behaviours for stakeholders. Such failures may be caused by insufficient upfront analysis of the feature request before its development.

Table I shows the number of occurrences of these failures recorded in Firefox, Netbeans and Eclipse. The study analysed all feature requests within the three projects from their start date until December 2010. The first three 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 number of feature requests that fall in each of the five failure types defined above. The percentage value denotes the failure density, i.e. the percentage of failure 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.

The table reveals some failure types have a relatively high density for the Firefox and Netbeans project. It also shows much lower numbers of reported failures in the Eclipse project, which after investigation can be explained by the limited use of Bugzilla for discussing and tracking feature requests in this project, with most discussions taking place outside of the feature request management system through mailing lists and IRC channels. The exception to this is the high number of occurrences of the stalled development failure, which we observed to be caused by the fact that in this project the status of feature requests are rarely consistently updated. We therefore decided to abandon using Eclipse for our experiments.

	Firefox	**Netbeans**	**Eclipse**
Duration	8.3yrs	10.3yrs	9.2yrs
Total Feature Requests	6,379	24,167	46,247
Assigned Feature Requests	382	2,152	7,564
Rejected Feature Requests	1,975	5,008	75,49
Product Failure	83 (21.7%)	49 (2.3%)	5 (0.1%)
Abandoned Implementation	23 (6.1%)	30 (1.4%)	61 (0.8%)
Rejection Reversal	50 (13.1%)	222 (4.4%)	191 (2.5%)
Stalled Development	49 (12.9%)	739 (34.3%)	4,584 (60.6%)
Removed Feature	2 (0.8%)	131 (2.3%)	17 (0.1%)

Table I
FEATURE REQUESTS AND FAILURES IN PROJECTS

III. 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 [20]. 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 countermeasures such as resolving ambiguities or inconsistencies about a desired feature, or monitoring and guiding its later progress.

An early failure prediction method 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 method 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, i.e. 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, i.e.

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

When designing a prediction method, there's an inherent conflict between these two measures: generating more alerts will tend to increase recall but decrease precision, whereas generating fewer alerts will tend to increase precision but decrease recall. An important question when designing and evaluating prediction methods 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:

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

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 [19]. The ratio between the cost of fixing a requirement defect after product release compared to fixing it during upfront requirements analysis is commonly cited to be between 10 and 100 for large projects [3][20][14], and between 5 to 10 for smaller projects with lower administrative costs [4][14].

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

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

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 methods by assessing their expected value per failure:

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

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:

$$\frac{ExpectedValue}{Failure} = recall.(\alpha - \frac{1}{precision}) \qquad (1)$$

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 is rarely available. In the absence of this data, model users can estimate these numbers based on the general findings from the empirical studies reported above, and assess the value of a prediction method 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).

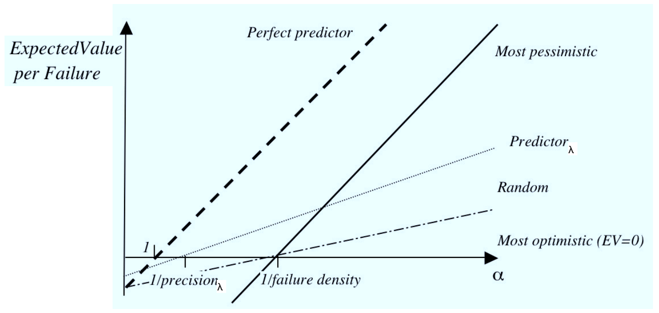

Figure 2. Expected value per failure as a function of α

The advantage of using equation 1 over using the f-score function or a function that takes some weighted average of precision and recall 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 methods with known precision and recall, as shown in Figure 2.

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 $\frac{Failure}{FR}$. 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; and 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 methods that compare themselves against a random predictor [16][6][22], the baselines in our context are the most optimistic and most pessimistic predictors.

Figure 2 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 $\frac{1}{1-recall_\lambda}.(\frac{1}{FailureDensity} - \frac{recall_\lambda}{precision\lambda})$. 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 methods will therefore be to generate predictors whose precision are higher than the failure density and whose range of α values for which they outperform the most optimistic and most pessimistic predictors is as large as possible.

IV. Predicting Failures

The class of machine learning techniques that apply to our problem, known as classification algorithms, first construct prediction models from historical data and then use these models to predict classifications for new data. In our case the historical data used to generate prediction models is past feature requests, while the new data consists 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.

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

- The failure type and the point in a feature request's life-cycle at which predictions is to be made (i.e. when it is rejected for the rejection reversal failure, and when it is assigned for the other four types of failure).
- The classification algorithm to be used; in our experiments, we have used the Naive Bayes, Decision Table, Linear Regression and M5P-Tree classification algorithms which constitute a good representation of the different types of classification techniques [21].
- The *feature request attributes* that are to be taken into account by the classification algorithm; in our

experiments we have used the following 13 attributes: the number of participants in the discussion, the number of posts by the person who first submitted the feature request, the percentage of posts made by this person, the number of posts by the person who is eventually assigned the feature feature request, the percentage of posts by this person, the total number of posts in the discussion, the number of words in each post, the number of words in the whole discussion, the number of code contributions during the discussion, the time elapsed between posts, the total time elapsed in the discussion, and finally 'bag of words' and 'term frequency in document frequency' (TFIDF) which are two standard text analysis attributes that count the number of occurrences and the frequency of each word appearing in the discussion.

- An estimated value for α used by our framework to decide the threshold at which to generate alerts so as to yield to best possible expected value, and to compare this expected value to the baselines in validation.

Using these inputs, our automated failure prediction framework performs the following steps to construct and validate a failure prediction model:

1. Extracting Data: Extract a large dataset from the distributed feature request management forum. This data in its raw form includes the textual content of discussions, their structure, their associated meta-data and the history of changes made to this meta-data.

2. Identifying Failures: Identify which feature requests correspond to failures of the requested type using the definition given in Section II-B.

3. Trimming Discussions: Trim posts within each feature request to the point where we wish to make a prediction, as we only wish to consider discussion data before point.

4. Preparing Training Set: Parse the discussions into a *training set* in a data format recognisable to classification algorithms. A training set consists of a collection of *instances*, each of which represents a feature request discussion. Instances are assigned a binary class indicating whether they have given rise to a failure. Along with this class every instance is assigned a collection of attribute-value pairs that represent the discussion's data in accordance with the discussion attributes selected by the user.

5. Training Prediction Model: The training set is given as input to a classification algorithm that generates a model for predicting the class of an instance, indicating whether or not a feature request is predicted to give rise to a failure.

6. Setting Alert Threshold: This step is only needed for prediction models that give to each feature request a numerical score, such as those generated using the Naive Bayes and the M5P-Tree algorithms, instead of an absolute prediction as to whether or not a feature request belongs to a failure class, such those generated using the Decision Table and Linear Regression algorithms. These algorithms require a threshold for generating alerts. Our framework sets this threshold automatically by calculating the precision, recall, and expected value resulting for different thresholds in the range -1 to 1, and selecting that which gives us the highest expected value. During this step. precision and recall for each threshold value are estimated using the standard method of 10-fold cross validation described in the following step. Figure 3 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.

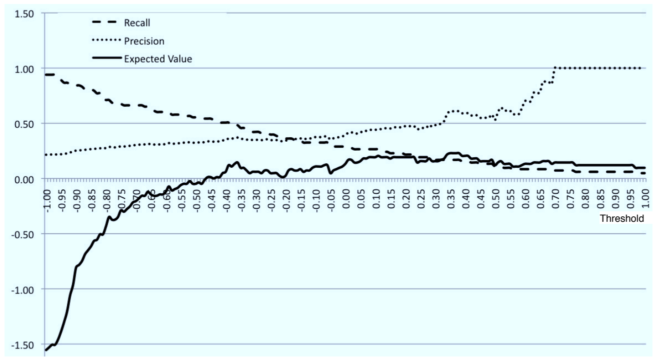

Figure 3. Recall, Precision and Expected Value as a function of Alert Threshold

7. Evaluating the Prediction Method: The prediction method is then subjected to 10-Fold cross validation - a standard method for evaluating classification algorithms [21]. The evaluation consists in performing ten experiments. In each experiment 90% of the feature requests are used to construct a prediction model in accordance with steps 5 and 6. Each prediction model is then evaluated by assessing its precision and recall using the remaining 10% of the feature requests as test cases. The precision and recall of the prediction method are estimated to be the mean of these measures for the ten experiments. The expected value of the prediction method is then computed from α and its precision and recall.

We have developed a research tool, Intueri, that implements this process. 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. Intueri processes this data into training sets, which are then used to generate prediction models and 10-fold cross validation results. This activity is outsourced to an open source tool, Weka [10], that provides an interface to many classification algorithms through both the command line and a GUI. Results are currently stored and analysed in Matlab.

V. Prediction Experiments

We have applied our early failure prediction framework to two large-scale open source projects, Firefox and Netbeans. We had also envisioned applying it to Eclipse but later rejected it for the reasons explained in Section II. 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 Section II be predicted from early discussions before the feature request is either assigned or rejected? What expected value can a project hope to obtain from such predictions?
3) What attributes of early feature requests discussions, if any, can be used as reliable predictors of later failures?

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 the Decision Table and Naive Bayes approach 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 methods, meanwhile, consistently produced similar results in terms of expected value. In all cases, however, the less computationally complex M5P-tree method generated prediction models significantly faster than the Linear Regression approach - in the order of minutes as opposed to hours. For our following experiments on the Netbeans data set we therefore only used the M5P-tree method.

To answer the second set of questions we have generated and evaluated prediction models for all failure types and predictive attributes from the Firefox and Netbeans data sets using the M5P-tree classification algorithm. Table II summarizes the results by presenting for each of the two projects the prediction method that yielded the most expected value. The model that gave the most expected value for product failure predictions in Firefox, for example, was generated using the ‘number of posts in the discussion’ attribute. The table also shows the estimated α value we have used for setting the alert threshold and computing the expected value, and the precision and recall obtained. When the best prediction method is shown as being the most pessimistic or most optimistic this means that there was no predictive model generated which performed better than these baseline predictors. In such cases, as you will recall from step 6 of our framework, the alert threshold for prediction models

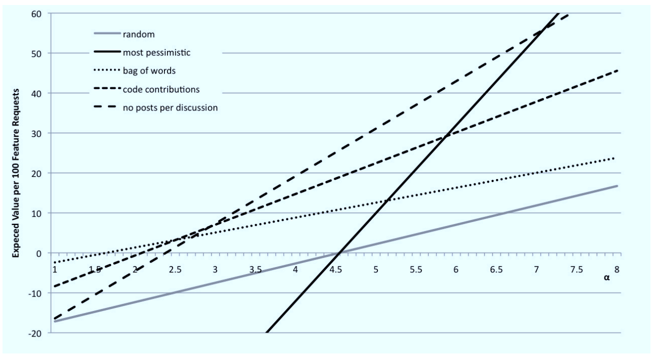

Figure 4. Expected Value as a Function of α for Product Failure Prediction Models in the Firefox Project.

is automatically set so that it behaves as the best baseline predictor.

Several observations can be made from this table. The results show that it is possible to generate early failure prediction models that provide positive expected value to a project. We have been able to generate good prediction models for all types of failures. No prediction method, however, provided the most expected value for a failure in both projects. This means that good prediction methods may be very project specific and that it may be difficult to find a prediction method for particular failure types that performs consistently across a large set of projects.

The α-values we have used for the different failure types have been estimated by ourselves and represent nothing more than informed guesses. In the absence of empirical validation 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 of the expected value of a prediction model. Figure 4, 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 5 cases in which predictive models yielded more expected value than the baselines.

Table III helps us answering the third question. We show the expected value for each of the predictive attributes in the six experiments where at least one predictive model outperforms the baselines. 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

		Product Failure	**Abandoned Implementation**	**Rejection Reversal**	**Stalled Development**	**Removed Feature**
	estimated α: $\frac{C_f}{C_a} * P_s = \alpha$	10*0.3 = 3	50*0.5 = 25	10*0.5 = 5	10*0.5 = 5	20*0.2 = 4
Firefox	Model with best expected value	Posts in Discussion	Most Pessimistic	Most Optimistic	Code Contributions	Most Optimistic
	Recall	0.54	1.00	0.00	0.94	0.00
	Precision	0.42	0.06	1.00	0.22	1.00
	Expected value per 100 FRs	7.3	50.4	0.0	5.5	0.0
Netbeans	Model with best expected value	Most Optimistic	TF-IDF	Words per Discussion	Percent by Assignee	Bag of Words
	Recall	0.00	0.40	0.43	0.84	0.40
	Precision	1.00	0.08	0.50	0.47	0.50
	Expected value per 100 FRs	0.0	7.0	5.7	82.9	1.8

Table II
PREDICTION MODELS WITH BEST EXPECTED VALUES FOR FIVE FAILURES IN THE FIREFOX AND NETBEANS PROJECTS.

Predictive Attribute	**Firefox: Product Failure**	**Firefox: Stalled Development**	**Netbeans: Abandoned Implementation**	**Netbeans: Rejection Reversal**	**Netbeans: Stalled Development**	**Netbeans: Removed Feature**
Most Pessimistic	-33.6	-34.7	-104.3	-27.1	70.7	-28.8
Random	-7.4	-4.5	0.0	-2.4	24.0	-1.7
Most Optimistic	0.0	0.0	0.0	0.0	0.0	0.0
Number of participants	5.8	0.0	0.0	0.5	70.7	0.0
Posts by reporter (#)	0.5	0.0	0.0	0.0	0.0	0.0
Posts by reporter (%)	5.1	0.0	0.0	0.0	70.7	0.0
Posts by assignee (#)	5.1	0.0	0.0	0.0	80.3	0.0
Posts by assignee (%)	0.0	0.0	0.0	0.0	82.9	0.0
Posts in discussion	7.3	0.0	2.0	3.7	70.7	0.0
Words per discussion	4.7	1.2	2.0	5.7	75.5	0.0
Words per post	5.6	0.0	0.8	1.2	-0.9	0.0
Code contributions	7.0	5.5	0.0	0.0	70.7	0.0
Time elapsed per post	5.1	0.0	0.0	2.5	70.7	0.3
Time elapsed per discussion	0.0	0.8	1.8	0.0	70.7	0.3
Bag of words	5.0	3.5	5.5	5.0	76.5	1.8
TF-IDF	2.5	4.3	7.0	5.7	73.4	1.4

Table III
EXPECTED VALUE PER 100 FEATURE REQUESTS BY PREDICTIVE ATTRIBUTE

the discussion, even at a very rudimentary level as these two attributes do, could provide more reliable predictions than analysing attributes such as the number of persons involved in the discussion and the discussion length. 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 [22], the roles of the users involved in a discussion, or the system components affected by a feature request. We have also performed experiments, not described in this paper due to space constraints, where we generated and evaluated prediction models from combined sets of 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.

VI. Conclusions

Early failure prediction can be 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 presented a cost-benefit model for evaluating early failure prediction methods, we have presented a tool-supported framework for the development and evaluation of such methods, and we have reported the results of an experiment evaluating failure prediction methods constructed using different classification algorithms and a large number of predictive attributes for two large scale open source projects. This experiment exposes the possibilities and limitations of early failure prediction using state-of-the-art classification algorithms and simple characteristics of feature request discussions as predictive attributes.

Possibilities to significantly improve the performance of early failure prediction methods might be to use richer sets of predictive attributes such as the presence of rationale in a feature request discussion (which could be indicated by common intentional phrases), the architectural components potentially affected by a feature request, or the roles, expertise, and communication structure of the persons involved in the discussion. Allowing stakeholders and developers to manually tag a feature request's description with requirements defect types could also provide for powerful predictive attributes.

References

[1] D. Berry, D. Damian, A. Finkelstein, D. Gause, R. Hall, and A. Wassyng. To do or not to do: If the requirements engineering payoff is so good, why aren't more companies doing it? 2005.

[2] C. Bird, D. Pattison, and R. D'Souza. Latent social structure in open source projects. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35, 2008.

[3] B. Boehm and P. Papaccio. Understanding and controlling software costs. *Software Engineering, IEEE Transactions on*, 14(10):1462–1477, 2002.

[4] B. Boehm and R. Turner. *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley Professional, 2003.

[5] B. Cheng and J. Atlee. Research directions in requirements engineering. pages 285–303, 2007.

[6] J. Cleland-Huang, H. Dumitru, C. Duan, and C. Castro-Herrera. Automated support for managing feature requests in open forums. *Commun. ACM*, 52(10):68–74, 2009.

[7] D. Damian. RE challenges in multi-site software development organisations. *Requirements Engineering*, 8(3):149–160, 2004.

[8] N. Fenton, M. Neil, W. Marsh, P. Hearty, Ł. Radliński, and P. Krause. On the effectiveness of early life-cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537, 2008.

[9] C. Fitzgerald. Support for collaborative elaboration of requirements models. *Internal UCL Report*, 2009.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The weka data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[11] J. Herbsleb and D. Moitra. Global software development. *Software, IEEE*, 18(2):16–20, 2001.

[12] P. Laurent and J. Cleland-Huang. Lessons learned from open source projects for facilitating online requirements processes. In *Requirements Engineering: Foundation for Software Quality*, pages 240–255. 2009.

[13] P. Laurent, J. Cleland-Huang, and C. Duan. Towards automated requirements triage. 2007.

[14] S. McConnell and I. Books24x7. *Code complete*, volume 2. Microsoft Press Washington, 2004.

[15] J. Musa. *Software reliability engineering: more reliable software, faster and cheaper*. Tata McGraw-Hill, 2004.

[16] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, page 461. ACM, 2006.

[17] B. Nuseibeh and S. M. Easterbrook. Requirements engineering - a roadmap. *ICSE - Future of SE Track*, pages 35–46, 2000.

[18] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, illustrated edition edition, 2004.

[19] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 249–258. IEEE, 2002.

[20] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, Mar. 2009.

[21] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann Pub, 2005.

[22] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 1–11. IEEE, 2009.

[23] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. 2007.