Which Work-Item Updates Need Your Response?

Debdoot Mukherjee  
IBM-Research–India  
New Delhi, India  
Email: debdomuk@in.ibm.com

Malika Garg  
Indian Institute of Technology, Delhi  
New Delhi, India  
Email: malikagarg03@gmail.com

Abstract—Work-item notifications alert the team collaborating on a work-item about any update to the work-item (e.g., addition of comments, change in status). However, as software professionals get involved with multiple tasks in project(s), they are inundated by too many notifications from the work-item tool. Users are upset that they often miss the notifications that solicit their response in the crowd of mostly useless ones. We investigate the severity of this problem by studying the work-item repositories of two large collaborative projects and conducting a user study with one of the project teams. We find that, on average, only 1 out of every 5 notifications that are received by the users require a response from them. We propose TWINY – a machine learning based approach to predict whether a notification will prompt any action from its recipient. Such a prediction can help to suitably mark up notifications and to decide whether a notification needs to be sent out immediately or be bundled in a message digest. We conduct empirical studies to evaluate the efficacy of different classification techniques in this setting. We find that incremental learning algorithms are ideally suited, and ensemble methods appear to give the best results in terms of prediction accuracy.

I. INTRODUCTION

Issue Tracking & Work-Items: Work-item management software (also known as Issue Tracking Systems (ITS)) have emerged as the de facto standard platform to support collaboration in both open source and commercial software projects. Different varieties of such tooling help to track different kinds of activities—bug tracking systems (e.g., Bugzilla) capture effort toward bug fixing in bug reports, work-item software (e.g., Rational Team Concert, Microsoft Team Foundation, JIRA) manage tasks spanning across the entire project life-cycle, ticketing systems (e.g., Tivoli Service Request Manager) maintain lists of issues raised in contact centers and so on. A work-item encapsulates all information pertaining to a single unit of work in a project. Attributes in a work-item, such as title and description, communicate the task or issue. Typically, the person who is primarily responsible for the work-item is designated as its owner. Other team members who may contribute are marked as subscribers. The team collaborates by adding comments and attachments to the work-item as they progress toward its resolution. Further, the work-item captures the status of the task, tracks the effort expended on it, and manages the review / approval workflow.

Work-Item Notifications: Any action that updates a work-item raises a notification for its subscribers1. The notification describes the work-item update—who acted, which work-item attribute was edited etc. They are sent through email and are also displayed in project dashboards. Work-item notifications help in a couple of ways:

1) The recipient can determine whether (s)he needs to respond to the work-item update. Depending upon the urgency implied in its notification, (s)he can switch focus to the work-item.

2) Improves Group Awareness [1] in distributed, collaborative projects by helping team members stay aware of each other’s activities and plans.

Notification Spam: Today, the notification mechanisms of all issue tracking tools share a common pain-point—they generate a deluge of work-item notifications to flood users’ mail-boxes on a regular basis [2], [3]. The fact that a majority of such notifications do not warrant any attention just adds to the frustration. Moreover, the really “important” notifications tend to get lost in the crowd, thus defeating the purpose of the notification feature. Our user study (Section II-C) reveals that, on an average, only 1 out of 5 notifications is deemed to be important and requires a response from the recipient. Also, we notice that those who are in a leadership role are more severely affected by notification spam than individual contributors.

Why so many notifications? Gonzalez et al. [4] discuss how today’s software professionals (analysts, developers, testers and project managers alike) profusely multi-task on a daily basis. Their research shows that people distribute their time across ten working spheres on an average; some of these spheres are of primary importance and others of peripheral importance. We observe strong evidence of such fragmented work patterns as we mine work-item repositories (Section II-B). People subscribe to many work-items and the number of notifications received by them is much greater than the number of work-item updates from them. This suggests that people are primarily concerned with only a small fraction of the subscribed work-items and their input is sparingly necessary in rest of the work-items. However, since the current notification generation systems have no built-in intelligence, the users keep receiving updates from all of their subscribed work items, all the time.

Available Alternatives: To counter the acerbity in users, popular work-item management products have introduced some rudimentary measures to support personalized tuning of work-item notification generation. Most issue tracking tools

---

1The owner and creator of a work-item are included whenever we mention subscribers henceforth.
Currently allow users to opt for daily or weekly digest emails instead of individual notifications. However, there may be notifications that solicit immediate responses; so receiving such updates at the end of the day/week does not help. Certain tools (e.g., Rational Team Concert [5]) provide greater, fine-grained control whereby users can enable only specific types of work-item updates for notification generation. For instance, one may choose to stay informed about new comments being added but not be bothered about changes in status of work-items. However, we cannot expect the update of any work-item attribute to be always considered important from a user’s perspective, e.g., not every work-item comment will solicit attention from a user; only some will do so. Also, such solutions put an additional burden on users to set the preferences and hence are seldom used. Another alternative for tracking work-item updates entails regularly visiting dashboards that show recent activity on a list of work-items selected by certain criteria (e.g., owned by me, subscribed by me, labeled by a particular category or tag). This can be tedious because the user needs to browse multiple dashboards to assess which work-items need attention—no single combination of work-item facets can select a list of work-items that remains always relevant for a user.

**Objective:** The goal of our research is to automatically determine whether a work-item notification generated for a person needs his/her attention. In this paper, we take the first step toward this goal as we predict whether a notification can prompt its recipient to act in response to it. Such a capability may be used to flag the notifications appropriately so that they stand out in a crowd. Alternately, the email generation facility may use this prediction to decide whether the update notification needs to be sent out immediately or later in a message digest. We do realize that certain notifications may be nice-to-have even if they do not result in any action; especially those that help increase group-awareness. However, these can be good candidates for being bundled in a message digest; the scope of this paper does not attempt to find such notifications.

**Approach:** We propose TWINY - *These Work Items Need You*, an approach to train a machine learning classifier to predict whether a notification may draw any response from its recipient. TWINY can learn a model of users’ responses to work-item updates by training on history logs available in work-item repositories. For every update to a work-item, TWINY creates a notification-example corresponding to each notification that is sent to the different subscribers on the work-item. The notification-example is a set of features that codify how the recipient of the notification may perceive the work-item update based on factors such as:

- the recipient’s level of involvement on the work-item
- the nature of the work-item update and the recipient’s association with it
- the recipient’s affinity with the user who made the update

Now, if the recipient updates the work-item soon after receiving the notification, then the notification-example is labeled as *Response-Reqd*, else it is labeled *No-Response-Reqd*. TWINY trains a classifier on many such notification-examples to learn a model that is able to predict the label of a new notification. Clearly, such a model needs to be updated continuously to adapt with changing project dynamics. Thus, incremental learning techniques [6], [7] that can update a model with an example and then discard the example are ideally suited in this setting. They help avoid tricky questions that otherwise arise for conventional batch classifiers viz. how many examples to use and from which window of time? Further, we investigate the use of adaptive learning [8] to deal with the non-stationary nature of the distributions of different TWINY features.

**Empirical Evaluation:** We conduct extensive empirical studies on a data-set containing over 1 million notifications recreates from more than 19k work-items from 2 large collaborative project suites—IBM’s Jazz platform and an internal IBM initiative named Consultant’s Assistant (CA). We experimentally evaluate different classification approaches—Bayesian, Support Vector Machines and Decision Tree Learners (both batch and incremental implementations)—for their efficacy in detecting notifications that prompt a response. Also, we evaluate ensemble classifiers, which combine predictions of multiple base models to enhance prediction accuracy. A bagging ensemble implementation of an Adaptive Hoeffding tree scores highest in terms of F1-measure (0.61 for Jazz and 0.44 for CA) for retrieving notifications requiring a response.

**Contributions:** Our contributions include:

- Validating the severity of notification spam in collaborative software projects through a quantitative analysis of work-item repositories and a user study (Section II)
- Developing an automatic framework to predict notifications that require a response (Section III)
- Empirically evaluating the efficacy of different classification techniques and features that may be applicable (Section IV)
TABLE I
DETAILS OF DATA-SETS USED IN OUR STUDIES

<table>
<thead>
<tr>
<th>Jazz</th>
<th>CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Work-Items</td>
<td>17600</td>
</tr>
<tr>
<td>Defects</td>
<td>8178</td>
</tr>
<tr>
<td>Tasks</td>
<td>5409</td>
</tr>
<tr>
<td>Users</td>
<td>820</td>
</tr>
<tr>
<td>Start Date</td>
<td>28 Oct 2010</td>
</tr>
<tr>
<td>End Date</td>
<td>17 July 2012</td>
</tr>
<tr>
<td>Work-item Updates</td>
<td>167,657</td>
</tr>
<tr>
<td>Notifications</td>
<td>949,082</td>
</tr>
</tbody>
</table>

II. MOTIVATIONAL STUDIES & PROBLEM SET-UP

In this section, we empirically validate the severity of the problem of notification spam—an overwhelming number of notifications being generated for users of work-item management software in collaborative projects. Also, we present results from a user study that assesses how users perceive the utility of the notifications received by them. Finally, we set up a classification problem to predict whether a notification results in a response and devise a labeling strategy to create examples necessary for training.

A. Project Profiles

Let us first review the data that we use to conduct the empirical studies presented in this section and in Section IV. For the purposes of our experiments, we crawl the history logs of work-item updates from the work-item repositories of two sets of projects. Table I lists various details on the same.

1) Jazz: Jazz, a large IBM initiative, is a collection of many products that support collaboration across the software and systems life-cycle. The work-item tool used in the Jazz projects is Rational Team Concert (RTC) and the repository is open to any registered user of www.jazz.net. It contains more than 220K work-items as of today. For the purpose of our studies, we sample a set of 17.6 K consecutive work-items raised during the period Oct 2010 to July 2012. This set contains work-items from three different Jazz products: RTC, Jazz Foundation and Jazz CLM. We find a team of 820 users to be active in these work-items—we include a user to be in the team only if (s)he participates in at least 3 work-items. Part of the team is shared across the three products. We crawl and parse the history logs for this set of work-items to resolve 167,657 updates that result in the generation of 949,082 notifications for subscribers.

2) Consultant’s Assistant (CA): CA is a joint project between IBM Research and IBM Global Business Services that has developed three productivity tool-sets for IT consultants. Just as in Jazz, the CA core team also works on all three products and uses RTC for work-item management. We recreate 167,558 notifications corresponding to 29,284 updates on 2239 work-items.

B. Case Study: Too Many Notifications–Very Few Updates?

We analyze the two projects to study the volume of notifications that the users keep receiving. In Figure 2, plots (A) and (C) show the percentage of users (Y axis) in the respective projects, who receive greater than X notifications (X axis) on more than Z days in an average working month (Z axis). For instance, we observe that 5 % of users in Jazz and 12.6 % users in CA receive greater than 20 notifications daily for more than 10 days (i.e., almost every alternate day) in an average working month. Plots (B) and (D) are drawn in a similar manner, but plot the volume of work-item updates instead. In contrast to the volume of notifications received, we find that no one in either Jazz or CA updates work-items even 3 times daily for 10 days in an average month. Figure 2 suggests that the users are receiving a disproportionate number of notifications compared to their activities on the work-items; hence it may be worthwhile to investigate whether its feasible to reduce the load of work-item notifications.

Further, we study whether the problem of notification spam is more severe for users in a leadership role than those who are individual contributors. Figure 3 shows a box plot comparing the ratio of Notifications : Updates for the 21 leads and 48 individual contributors in CA\(^2\). Clearly, the values of the ratio appear to be much higher for the leads; so the problem may be more severe for them. We verify that the difference between the ratios is statistically significant by applying the Mann-Whitney test\(^3\) (p-value = 0.047).

C. User Study: Utility of Notifications

We conduct a user study to learn how users perceive the utility of work-item notifications.

**Methodology:** We interview eight members of the CA team (2 developers, 2 architects, 2 team-leads and 2 project managers). We ask them to rate the usefulness of a list of work-item notifications, which they had received in the past. For every subject, all the notifications presented in the survey are chosen from a single day on which the subject had received more than 20 notifications. The rating is requested on a scale of 1-4:

1 - Requires an Immediate Response
2 - Requires Response but can act later
3 - Good to be Aware; no response required
4 - Not useful at all

\(^2\)We cannot replicate the experiment for Jazz because role information is not available

\(^3\)http://en.wikipedia.org/wiki/Mann-Whitney_U
We collect responses on a total of 182 notifications from our subjects.

**Observations:** Figure 4 shows that, on average, only 20% of the notifications received by a subject demand an update from him/her, i.e., are rated as 1 or 2 [Min: 15% for project managers, Max: 31% for developers]. Since only a small fraction of total notifications require a response, we feel the need for specially flagging them to ensure that they do not go unnoticed. Interestingly, we observe that project managers get a lot of notifications primarily because team members wish to keep the managers in the loop about their activities. This is evident from the fact that the project managers rate a very high fraction (71%) of their notifications as “3: Good to be Aware”, whereas notifications received by developers rarely fall under this category (only 5%).

**Anecdotes:** During the process of interviews, our subjects shared many anecdotes about how they currently deal with notification spam. Here, we list a few of them:

**Project Manager:** “Updates from my client require urgent responses..rest are mostly FYI items”

**Developer:** “Usually, I only read notifications from the work-items that I own. For others, I need to pay attention only if my name is mentioned”

**Architect:** “I am responsible for reviewing code commits made by the team, so I keep an eye on updates that have linked change-sets.”

The sheer variety in the anecdotes from our subjects seem to suggest that no silver-bullet set of rules can control notification generation for all users. Therefore, it is appealing to investigate whether machine learning approaches can learn effective models from a project’s work-item history to help in this scenario.

**D. Labeling notifications to set up classification**

We formulate a supervised classification problem wherein we learn a model from work-item notification history such that it can classify a new notification as either Response-Reqd or No-Response-Reqd. The first step toward training such a classifier is to obtain a large number of past notifications that are labeled under the same classes. Manual labeling of a sufficient number of examples turns out to be infeasible. So, we develop an automatic labeling strategy that can approximate manual labeling.

A response to a notification can mean any action—writing code, reviewing a commit or even asking someone else to act. Although an action does not necessarily have to leave any footprint on the work-item (e.g., one call his/her the team-mate to discuss), usually it does so in projects where the work-item tool is the primary medium of communication. Therefore, a user’s update on the work-item can be taken to be his/her reaction to the preceding updates by other users. We theorize this idea with the following heuristic for labeling notifications:

*Label a notification generated for a user A as*
“Response-Reqd”, if A updates the work-item either within $T_1$ days of the notification or within $T_2$ days (where $T_2 > T_1$), provided no more than $k$ users update the work-item before A does. Otherwise, it is labeled as “No-Response-Reqd”. Typical values of $T_1$, $T_2$ and $k$ can be 2, 14 and 2 respectively, which translates to the following labeling strategy: an update to a work-item is taken to be a response to any notification received from the same work-item within the last 2 days; again it may be considered to be a response to any notification received within the last two weeks provided there have been no more than 2 users updating the work-item since the receipt of the notification. Such a labeling strategy enables us to automatically label a large number of past notifications necessary for effectively training machine learning classifiers.

**Evaluation of Labeling Strategy:** We label each notification that was rated by our subjects using the labeling strategy described above. Next, we evaluate how closely the ratings from the automatic labeler agree with those given by our subjects. In order to do so, we reduce the user rating to the scale used in labeling. A rating of 1 or 2 is considered as Response-Reqd, and 3 or 4 is taken to be No-Response-Reqd. Then, we compute Cohen’s kappa to measure inter-rater agreement. Across all notifications used in the study, average kappa is observed to be 0.442. The maximum kappa for a subject is 0.9 and the minimum is 0.12. Commonly cited references in literature suggest that kappa $> 0.4$ signifies moderate agreement. The raters agree in 135 cases; in 15 of them both rate Response-Reqd. In 18 out of 26 cases of disagreement, the subject rates Response-Reqd but the auto-labeler does not. We verify that in most of these cases, the user had acted or communicated outside the work-item management system (e.g., email or phone call). Thus, we can expect our labeling strategy to perform better where the work-item tool is used with greater rigor. Also, the heuristic can be tuned by obtaining the values of the parameters, $T_1$, $T_2$ and $k$, through maximization of agreement with small manually labeled sets, wherever available.

**Why don’t we detect notifications important for group awareness?** As discussed earlier, certain notifications can help in maintaining group awareness (those with rating $= 3$ in our user study). But, detection of such notifications is left out of the scope of this paper because we are unable to devise any scheme for automatically estimating ground-truth for the same. However, we realize the possibility that issue tracking tools can be instrumented to log user actions such as: opening notification, time spent in reading notification etc. Such instrumentation can give us a richer indication of users’ perception toward notifications and help us build more comprehensive systems for categorizing them in future.

**III. TWINY - THESE WORK ITEMS NEED YOU**

In this section, we describe TWINY (These Work Items Need You)—a machine learning based technique that predicts whether a work-item notification will require any action from the user. First, we present an overview of the approach, describing how a classifier learns from examples of users’ responses to notifications. Next, we list features that are used to train such classifiers. Finally, we expound on different kinds of classification techniques that can be applied; we investigate which methods are better suited than others to address our problem.

**A. Approach Overview**

A user receives a notification for a subscribed work-item whenever someone else updates it. TWINY predicts whether the user will act in response to the notification by learning from examples of responses to “similar” notifications in the past. For any workitem update, TWINY creates a notification example corresponding to a notification that is generated for a work-item subscriber. All factors underlying the work-item update, the state of the work-item and the preferences of the subscriber that may lead the subscriber to respond or not to respond are codified as features in the notification example. If the recipient updates the work-item soon after receiving the notification, then the corresponding notification example is labeled as Response-Reqd, else it is labeled No-Response-Reqd. TWINY trains a classifier on many such examples to learn a model that is able to predict whether any new notification will result in a response.

Figure 5 presents a flow diagram to illustrate how a TWINY system integrates within a work-item management software; certain steps are described more formally in the pseudocode listed in Figure 6.

**Classifying Notifications:** The TWINY workflow kick-starts whenever a work-item is saved after some update [Step (1) in flow diagram]. A notification is generated for each subscriber of the work-item [Step (2)]. Subsequently, TWINY creates a set of features that may throw light upon whether the recipient may respond to the notification [Step (3) in flow diagram; line 7 in pseudocode]. These features describe the nature of the work-item update (e.g., which attribute changed), the work-item’s attributes (e.g., severity, priority), the history of the recipient’s actions on the work-item (e.g., what fraction of total actions are done by him/her) and so on (See Section III-B). Given such a feature-set as input, a trained TWINY classifier classifies the notification as: Response-Reqd or No-Response-Reqd [Step (4); line 8]. The email notification generation module may use this guidance to appropriately flag the Response-Reqd notifications [Step (5)]; possibly akin to what Gmail does with priority inbox. Alternately, it may decide to only send out...
1: // Triggered for each action on a workitem
2: function PREDICTNLabel
3:   Input: Action a, Model m, Workitem w
4:   Input: List(NotificationExample) stream
5:   w ← GetWorkItemForAction(a)
6:   for all User u ∈ Subscriber(w) do
7:     NotificationExample n ← GenerateFeatures(a, w, u)
8:     n_pred ← Predict(n, m)
9:   stream.add(n)
10: end for
11: List(NotificationExample) impList
12: impList ← SelectImportant(w, a.actor, stream)
13: for all NotificationExample n ∈ impList do
14:   n.label ← "Important"
15: end for
16: function SELECTIMPORTANT
17:   Input: Timestamp currentDate, WorkItem w, User u
18:   Input: List(NotificationExample) stream
19:   Timestamp earliestDate ← currentDate − T2
20:   //selectExamples: fetches notifications from workitem w
21:   list1 ← selectExamples(w, u, earliestDate, stream)
22:   list1 ← selectLastNExamples(list1)
23:   list2 ← selectExamples(w, u, currentDate − T1, stream)
24:   return list1 ∪ list2
25: end function
26: function LABELUNIMPORTANT
27:   Input: List(NotificationExample) stream
28:   for all NotificationExample n ∈ overdue do
29:     if n.label = null then
30:       n.label ← "unimportant"
31:     end if
32:   end for
33: end function
34: end function

Fig. 6. Pseudocode for selected methods in TWINY workflow

Response-Reqd notifications and save the No-Response-Reqd notifications for a message digest. Next, TWINY creates an unlabeled notification example from the feature-set and adds it to the notification stream [Step (6); line 9], so that it can be used for training after getting labeled in due course.

Labeling Examples: As discussed in Section II-D, we assume a work-item update to be a response to one of the preceding updates on the work-item. So, we label the recent notifications, generated from the work-item, for the current actor to be Response-Reqd [Step (7); lines 17–27] as per our labeling strategy. Unlabeled notification examples in the stream are marked as No-Response-Reqd whenever they cross the threshold of recency, i.e., after T2 days have passed [Step (8); lines 29–35].

Training Classifier: A separate thread [Step (9)] continually draws labeled examples from the stream in order to update the trained model or to re-learn it; depending on whether we use an incremental classifier or not (See Section III-C).

Bootstrapping TWINY: To start using TWINY in a new project, either we need to use historical data from work-item action logs to bootstrap the classifier or use a model which was trained in another project. Here is how we can train a TWINY classifier from history logs of work-item actions. We crawl and parse the action logs to obtain records such as: <<Work-Item ID, Timestamp for action, Name of Actor, Attribute Modified, Value Before Update, Value After Update >>. We process these records one at a time in order to recreate every state in the life-cycle of a work-item; we store a snapshot for each state. A work-item snapshot is represented as a set of work-item attributes (e.g., title, description, priority, list of comments, subscribers) and their values. The creation of the work-item results in its first snapshot; subsequently a snapshot is created to reflect a new state of the work-item after every save following some updates. For each set of actions resulting in a snapshot, a notification example is generated for each subscriber listed in the work-item snapshot. For each notification example, we generate a set of features from the data in the work-item snapshots and some bookkeeping information about the users (e.g., their affinity toward work-item categories, tags or other users). The example is labeled using our labeling strategy and then used for training the classifier.

B. TWINY Feature Set

TWINY generates a set of features for each notification that attempt to guess the recipient’s outlook toward the work-item update. Table II shows the list of features that we use in our evaluation. We expect these features to be correlated with the possibility of the recipient responding with a follow-up update on the work-item—the rationale behind including each feature is explained in the third column in Table II. Also, we note what information is necessary to compute each feature. Some features can obtained from the details of the action itself (e.g., the type of attribute(s) updated); certain others involve analyzing past actions in the work-item (e.g., time elapsed since the subscriber’s last action) and the rest of the features need to query the user’s action history (e.g., computation of affinity scores between users).

C. Suitability of Different Classification Techniques

Decades of research in machine learning has led to a variety of classification techniques that can effectively induce general patterns from examples. Here, we briefly describe three popular classes of classification algorithms. Section IV-B evaluates their efficacy in the TWINY setting.

1) Bayesian Methods use Bayes’ Theorem to compute posteriori probabilities for hypothesis from evidence in training data. The simplest method, Naive Bayes works with the assumption that all features in the training data are conditionally independent. Naive Bayes classifiers are used in most email spam filters.

2) Decision Tree Learners learn easy-to-interpret tree models where leaves represent class labels and branches

4In our experiments, we use this method to setup TWINY using work-item histories of Jazz and CA projects
represent conjunctions of features that lead to those class labels. e.g., C4.5, ID3.

3) **Support Vector Machines (SVMs)** construct a hyperplane or set of hyperplanes in a high dimensional space in order to separate the examples of different classes. e.g., SMO, Pegasos [9].

Even though all of the above are applicable for our cause, the following concerns arise when we try to employ their standard implementations in a traditional batch learning\(^3\) mode:

1) **How much training data to keep?** The number of notification examples produced per work-item is roughly a product of the number of its subscribers and the number of times it is updated. So, such data can grow very fast. For instance, the 17.6 K Jazz workitems in our sample generated around 0.95 million notification examples. Most implementations of decision tree learners and SVMs (e.g., J48 and SMO in Weka [10]) hold the entire training data set in memory so that they can read it multiple times. We find that they quickly run out of memory if we attempt to train them over the entire list of notification examples. Since there is no concrete guidance on how much training data is ideal, the process of keeping (or discarding) examples from our notification stream can become quite ad-hoc.

2) **Adapting classifiers to keep pace with project dynamics:** The TWINY classifier needs to be tuned on a continuous basis to keep abreast with the constant churn going on in software projects. For instance, the distribution of features like user-to-user affinity or user-

<table>
<thead>
<tr>
<th>Title</th>
<th>Definition</th>
<th>Rationale</th>
<th>Available From</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is X's Name Referenced?</td>
<td>Is true if X's name is mentioned in a comment. Some simple heuristics are applied to ascertain whether there is a question implied for X. e.g., the sentence ends in a question mark.</td>
<td>If there is a question for X, it is expected that X should respond</td>
<td>Action Details</td>
</tr>
<tr>
<td>Attributes edited in current update</td>
<td>Indicates the set of work-item attributes that are being updated. E.g., severity, priority, comments</td>
<td>Edit of certain attributes are more important than others. e.g., A comment may be considered more important than a tag.</td>
<td>Action Details</td>
</tr>
<tr>
<td>Is X Being Added As Owner?</td>
<td>Checks whether the current update adds X as owner</td>
<td>If X is being set as Owner, X should start acting on the work-item; at least update about plans of action.</td>
<td>Action Details</td>
</tr>
<tr>
<td>Owned By X?</td>
<td>Indicates whether X owns the work-item or not</td>
<td>People are more active on the items that they own. So, the probability of the getting a response from X is higher than otherwise.</td>
<td>Work-Item History</td>
</tr>
<tr>
<td>Time since X's Last Action</td>
<td>Measures time elapsed since X's last action on the work-item</td>
<td>The series of comments on a work-item can be often perceived as conversations between two or more users. So the person who puts in the last comment has a high probability of responding to the current action.</td>
<td>Work-Item History</td>
</tr>
<tr>
<td>Is X the Last Actor?</td>
<td>Indicates whether X updated the work-item just before the current action</td>
<td>If X has not acted in a long time, then probably the work-item is not of interest to X anymore</td>
<td>Work-Item History</td>
</tr>
<tr>
<td>Is X in the Last 2 Actors?</td>
<td>Indicates whether X updated the work-item within 2 actions of the current one</td>
<td>Same Rationale as Above</td>
<td>Work-Item History</td>
</tr>
<tr>
<td>Fraction of X's Actions relative to actions from other users</td>
<td>(</td>
<td>X)'s actions on this work-item</td>
<td>(</td>
</tr>
<tr>
<td>Fraction of X's Actions on this item relative to other items</td>
<td>(</td>
<td>X)'s actions on this work-item</td>
<td>(</td>
</tr>
<tr>
<td>X's Affinity Toward Current Category</td>
<td>Measure of Xs actions on work-items having the same category as the current one, relative to those in work-items having other categories</td>
<td>If work-items of this category are important to X, then it is likely that this one will also be important.</td>
<td>User Action Profile</td>
</tr>
<tr>
<td>X's Affinity Toward Current Actor</td>
<td>Number of times X acted soon after actions performed by the current user (normalized suitably)</td>
<td>If X responds quickly to some people in the team (e.g., the client), then the notification from those users will be important to X.</td>
<td>User Action Profile</td>
</tr>
</tbody>
</table>
to-category affinity may change drastically with people moving in and out of projects and switching their roles. Traditional data mining techniques are expected to work effectively with stationary data distributions only; thus such classifiers may not adapt well with the flux in software projects.

To deal with the first problem, we need incremental learning algorithms that train on one example at a time and can discard the example may be immediately after training. This ensures that we do not need to store the entire stream of notifications. Incremental implementations of decision tree learners and SVMs have been discussed in literature [6][7]; a Naive-Bayes classifier is updateable readily. The second problem of dealing with concept-drift [11] can be addressed through work on adaptive stream learning [8], [12], [13].

Yet another class of classifiers that effectively deals with noise and evolving data is known as ensemble classification [14]. Ensemble classifiers train multiple models of a base classifier and then combine their predictions to increase prediction accuracy. Bootstrap aggregating, often abbreviated as bagging, is an ensemble method that considers equal weight for each model in the ensemble vote. We evaluate incremental and ensemble classifiers in addition to conventional batch implementations as part of our empirical studies.

IV. EMPIRICAL EVALUATION

In this section, we empirically evaluate different machine learning techniques for their efficacy in predicting notifications that require responses. Further, we analyze several key considerations that come up whilst applying TWINY in practice; we organize these as a set of research questions. We report relevant empirical evidence from Jazz and CA projects as we try to answer these questions.

A. PRELIMINARIES

Metrics: Before we begin, let us review the definitions of some metrics that we report for most of our experiments. If we formulate our task as that of detecting Response-Reqd notifications from a collection of notifications, then the popular metrics of Information Retrieval can apply in a straightforward manner. Table III shows the meaning of true positives ($tp$), false positives ($fp$), false negatives ($fn$), and true negatives ($tn$) as we compare the predicted label from a classifier to the label stipulated by our labeling strategy. Given these, we can define the following:  

$\text{Precision (P)} = \frac{tp}{tp+fp}$; $\text{Recall (R)} = \frac{tp}{tp+fn}$; $\text{Specificity (S)} = \frac{tn}{tn+fn}$; $\text{Accuracy (A)} = \frac{tp+tn}{tp+fp+tn+fn}$; $\text{F - Score (F)} = \frac{2pr}{p+r}$.

Experimental Setup: We mine the work-item history logs from the RTC repositories of Jazz and CA projects (Table I). We parse these logs in order to re-create snapshots for each state (each save creates a new state) that a work-item has witnessed in its life-cycle. For every change of state, we instantiate a notification example for each of the existing subscribers of the work-item in that state. We generate a total of 949,082 notification examples for Jazz and 167,558 examples for CA. We store these notification examples, each one comprised of the TWINY features and a label, as Weka’s ARFF files. The examples are sorted in chronological order. We feed such ARFFs as input to the classifier implementations in Weka [10] and MOA⁶ to conduct experiments.

B. EFFICACY OF DIFFERENT CLASSIFICATION APPROACHES

Objective & Method: We study the efficacy of the different classification approaches (listed below) in detecting work-item notifications that may prompt responses.

1) Batch Learning: First, we employ a Naive-Bayes classifier, Pegasos—a SVM solver [9] and J48—a decision tree learner in the traditional batch setting. All three implementations are taken from the Weka toolkit [10]. Since all of these require the full training data to be kept in memory, they run out of memory when we load all examples from the Jazz set. For the purposes of experiments around batch learning, we partition the Jazz data set into 9 segments; each consisting of 100K consecutive examples. The CA set is split into 3 segments; each with 30K examples. For each segment, the first half is used for training a model and the next half for testing it.

2) Incremental Learning: We experiment with implementations of different incremental learning algorithms present in the MOA toolkit. We apply incremental versions of Naive-Bayes, Pegasos, Hoeffding Tree and Adaptive Hoeffding tree. In these evaluations, we use a test-then-train methodology to process the entire stream of notification examples that we create. For each example in the stream, first we use the available model to obtain a prediction and then update the model itself with the example.

3) Ensemble Methods: We use the Leveraging-Bag implementation for a bagging ensemble classifier in MOA. We compare two runs of the classifier for each project. We apply Naive-Bayes as the base learner in one run and apply Hoeffding Adaptive Tree in the other.

Results & Analysis: Table IV reports different metrics on the three categories of classification algorithms which we evaluate. The Naive-Bayes implementations record the highest recall and F-score in both batch and incremental classifiers; whereas the decision tree learners show a higher precision and specificity⁷. Interestingly, we observe that incremental learning gives us higher levels of accuracy than batch learning even as they possess the other desirable properties of better run-time performance, no requirement for storing training data and

⁶http://moa.cs.waikato.ac.nz/overview/

⁷Later in this section, we show the feasibility of sacrificing recall for precision and vice-versa
so on. The ensemble bagging method appears to much more effective than the other algorithms.

C. Analysis of Research Questions:

RQ1: How do we balance precision and recall? The age-old issue of balancing precision and recall comes up very prominently for the TWINY scenario. As we drastically reduce the high number of false positives presented by today’s notification systems, we naturally need to compromise on recall. The question that arises is: What are the costs of false positives vis-à-vis false negatives? The answer depends upon the manner in which we utilize the prediction in the end-user application. If we utilize the classification result to set a priority flag for the notification, then we may choose precision over recall to keep users enthused about the flagging feature. However, if we decide when to send out the notification based on the prediction, whether immediately or in a digest, then recall may become more important. Cost-sensitive classification is a flexible way to balance precision and recall. Based on the end-application, we may set a cost to indicate how false negatives should be penalized with respect to false positives. Figure 7 shows that as we increase the cost of mis-classification of fn : fp from 1 through 8, we can steadily increase recall by compromising precision, the gradient of increase becomes lower thereafter. For Pegasos, the recall reaches as high as 0.79 with precision at 0.36. However, only for Naive-Bayes, we can increase precision by using mis-classification costs of fn : fp that are less than 1. For Pegasos and J48, we can no longer find true positives with a cost less than 0.5.

RQ2: Which features help most in learning? To answer this question, we performed a feature-selection exercise using CfsSubsetEval [15] in Weka. For both CA and Jazz, the set of features listed in Table V show up as the subset of features that demonstrate maximum predictive ability with minimum redundancy in them.

RQ3: How much training data is good enough? With a fixed test set of 100K notification examples from Jazz, we evaluate the effect of increasing the size of training data on a J48 - Weka implementation. Starting with a training set of 10K examples selected from just before the window of test data, we progressively grow the size of the training window by moving backwards in time. Figure 8 shows that both precision and recall increase with increase in training data till about 50K examples, but there is no observable increase or decrease thereafter.

RQ4: Can we transfer learning from one project to another? We study how effectively can we bootstrap TWINY in one project with a classifier trained in another project. For this purpose, we test examples from CA with a model trained with Jazz data and vice-versa. Table VI shows that it is feasible to transfer learning across the projects with F-scores of 0.45 and 0.38 for Jazz → CA and CA → Jazz, respectively.

RQ5: How closely did we model user’s perception of importance? We apply a NaiveBayes classifier trained with the CA data to predict on a test set created with the 182 notifications used in our user-study with 8 CA users. Here, we compare the model’s prediction with the label assigned by the user (rating of 1, 2 → Response-Reqtd; rating of 3,4 → No-Response-Reqtd). Here are the results: $P = 0.43, R = 0.49, S = 0.83, A = 0.76, F = 0.38$. 

---

**TABLE IV**

Efficacy of Different Classification Approaches

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Jazz P</th>
<th>Jazz R</th>
<th>Jazz S</th>
<th>Jazz A</th>
<th>Jazz F</th>
<th>CA P</th>
<th>CA R</th>
<th>CA S</th>
<th>CA A</th>
<th>CA F</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaiveBayes</td>
<td>0.4</td>
<td>0.37</td>
<td>0.84</td>
<td>0.46</td>
<td>0.41</td>
<td>0.48</td>
<td>0.88</td>
<td>0.82</td>
<td>0.44</td>
<td></td>
</tr>
<tr>
<td>SVM - pegasos</td>
<td>0.29</td>
<td>0.26</td>
<td>0.84</td>
<td>0.76</td>
<td>0.16</td>
<td>0.36</td>
<td>0.65</td>
<td>0.61</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>Tree - J48</td>
<td>0.49</td>
<td>0.38</td>
<td>0.92</td>
<td>0.84</td>
<td>0.43</td>
<td>0.46</td>
<td>0.92</td>
<td>0.82</td>
<td>0.38</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE V**

Results of Feature Selection

<table>
<thead>
<tr>
<th>Is X’s Name Referenced?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction of X’s Actions relative to actions from other users</td>
</tr>
<tr>
<td>Fraction of X’s Actions on this item relative to other items</td>
</tr>
<tr>
<td>Owned By X?</td>
</tr>
<tr>
<td>Is X the Last Actor?</td>
</tr>
<tr>
<td>Is X in the Last 2 Actors?</td>
</tr>
<tr>
<td>Is X Being Added As Owner?</td>
</tr>
</tbody>
</table>

---

**TABLE VI**

Transfer Learning from One Project to Another

<table>
<thead>
<tr>
<th>Test CA with Jazz Model</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>A</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Jazz with CA Model</td>
<td>0.38</td>
<td>0.38</td>
<td>0.88</td>
<td>0.84</td>
<td>0.45</td>
</tr>
</tbody>
</table>

---

![Fig. 7. Varying costs of misclassification for fp and fn](image)

![Fig. 8. Varying size of training data in a batch setting](image)
To the best of our knowledge, ours is the first work to look into the problem of spam arising from work-item notifications. However, a large body of work exists that evaluate different machine learning techniques to address the problem of email spam filtering [16], [17], [18]. Email spam filtering demands a very different feature space than the one we have used in TWINY. Spam filters primarily use text-based features whereas TWINY’s features are mostly categorical or numeric in nature. However, spam filtering literature does indicate similar challenges to the ones that we highlight in this paper. For example, the need for adaptive learning to deal with concept drift is underscored in email spam domain as well [19], [20].

The research on detecting important or priority emails [21] comes closest to our work. Unlike spam, the notion of importance can be very subjective—users themselves don’t always agree on what could be possibly important. Gmail’s priority inbox formulates a ranking problem rather than using a classification approach; they apply simple logistic regression models so that prediction can scale over millions of users. They note that user preferences often do not correlate with their actions on emails (e.g., opening an email); thus priority inbox depends on input from individual users to tune their respective thresholds.

TWINY can be applied to generate personalized dashboards where users can view a stream of recent work-item updates that matter to them. In such a setting, one may draw an analogy with Facebook’s Edge Rank algorithm. An edge is any event that happens on Facebook (e.g., likes, comments, status updates). The algorithm ranks edges on a user’s news feed based on (1) affinity between the user and the edge creator, (2) importance of the edge (e.g., shares are more important than likes), and (3) time elapsed since the edge is created. Future work can evaluate how to effectively couple the notion of time decay with the classifier’s confidence score about importance in order to maintain the recency of personalized streams of work-item updates.

VI. CONCLUSIONS & FUTURE WORK

Based on our empirical analyses of the work-item repositories of Jazz and CA projects and the user-study conducted with a sample of the CA team, we may conclude that a large number of work-item notifications are generated in collaborative projects, which can be particularly disturbing for the users. In response, we propose TWINY – an automatic, machine learning based approach for detecting notifications which may require responses from the users. Our empirical evaluation with a data-set in excess of 1 million notifications reveals that the use of incremental learning and ensemble methods best address the problem.

Future work plans to use TWINY to deploy applications for suitably flagging notifications and controlling their distribution. We may instrument these applications to capture indicators of users’ attention spans on the notifications (e.g., time spent in reading a notification). Such indicators shall help us to create a comprehensive system for recommending “important” work-item updates. We may also investigate development of a Work-item Wall, analogous to the Facebook Wall, which can list the work-items to look into at any point in time. Subsequently, we can conduct field studies to better estimate costs of false positives and false negatives in the context of different TWINY applications.

ACKNOWLEDGMENT

The authors would like to thank the team of Consultant’s Assistant for their participation in the user study and for sharing insightful anecdotes which have helped us to imagine many of the features used for classification in TWINY.

REFERENCES