NeedFeed: Taming Change Notifications by Modeling Code Relevance

Rohan Padhye
IBM Research India
ropadhye@in.ibm.com

Senthil Mani
IBM Research India
sentmani@in.ibm.com

Vibha Singhal Sinha
IBM Research India
vibha.sinha@in.ibm.com

ABSTRACT

Most software development tools allow developers to subscribe to notifications about code checked-in by their team members in order to review changes to artifacts that they are responsible for. However, past user studies have indicated that this mechanism is counter-productive, as developers spend a significant amount of effort sifting through such feeds looking for items that are relevant to them. We present NEEDFEED, a system that models code relevance by mining a project's software repository and highlights changes that a developer may need to review. We evaluate several techniques to model code relevance, from a naive TOUCH-based approach to generic HISTORY-based classifiers using temporal code metrics at file and method-level granularities, which are then improved by building developer-specific models using TEXT-based features from commit messages. NEEDFEED reduces notification clutter by more than 90%, on average, with the best strategy giving an average precision and recall of more than 75%.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement—*Version control*

Keywords

Collaborative software development; version control; mining software repositories

1. INTRODUCTION

Software development is a collaborative activity. Multiple developers work on a project at the same time, checking-in their own code and accepting others' changes. Most collaborative development environments such as Rational Team Concert [6] or the Web-based GitHub [1] platform allow developers to sign up for a feed of code check-in notifications, in the form of emails, RSS feeds or pop-up alerts. These feeds help developers keep themselves aware of changes happening in the project's code-base. Kim [19] conducted a study to explore developers' experiences with such change feeds. She reported that developers find change notifications to be very important, especially when the changes affect their own code. Also, developers like to be notified about changes to artifacts which they are responsible for, in order to review incoming changes for possible discrepancies. However, the study revealed that developers find the existing "laundry list" of notifications to be inadequate for their information needs. The change feeds are not personalized and hence developers need to go through every item to identify changes that might be of relevance to them.

Now, what changes would a developer consider relevant? Fritz and Murphy [11] found, through a user study, that though the notion of relevance was very subjective, in general developers find information about artifacts that they have worked on in the past and likely to continue updating in the future to be more relevant than information about artifacts that they have not worked on at all. Hence, the likelihood of a developer modifying a source code artifact in the future is a good indicator of whether that developer would find changes involving that artifact to be relevant today.

In his PHD thesis [10], Fritz attempted to answer the question *"Which changes should I know about?"*. He modeled a developer's knowledge of various code files (called the DOK model) based on code authorship and recency of interaction with the source code items in an IDE. In a user study involving 3 developers, he recommended the participants to review changes to bug reports associated with source code artifacts having high DOK values, and observed that 4 out of the 6 recommendations were found useful by the developers in the study. We build upon this work in our paper, by exploring additional techniques at different granularities and by empirically evaluating these techniques over a much larger data set.

We present NEEDFEED, a system that models code relevance to personalize a developer's change notification feed by highlighting those items that a developer may need to review, by using the heuristic that developer's would find those items to be relevant which they are likely to modify in the future. We explore various techniques to model code relevance and evaluate these models on a set of 40 medium-to-large sized projects hosted on GitHub. We found that, on average, the best strategy reduces notification clutter by more than 90%, and that more than 75% of the recommended items are relevant by our definition (precision), while more than 75% of all relevant items are recommended (recall).

We first implemented a naive strategy, which, for every change-set[1], notifies all developers who have modified any of the files involved in this change-set at least once in the past. We call this the TOUCH-based strategy, which effectively subscribes users to all

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642985.

[1]We use the terms *commit* and *change-set* interchangeably to refer to an atomic set of changes authored by a single developer, which is logged in a software repository, and which can be used to uniquely identify the state of the system at some point in time.

changes involving files which they have contributed to, indefinitely. As compared to a blind strategy that notifies every developer about all changes in the project, the naive TOUCH-based strategy alone reduces the notification clutter by 82%, on average, though the average precision is less than 35%. This means that many change notifications are sent to developers who never revisit any of the changed artifacts (and thus could be considered irrelevant). This is intuitive because developers often make minor edits to files which they do not regularly work with, and which they may never revisit in the future. Notifications about changes to these files are not relevant to them and hence the TOUCH-based strategy is not very precise.

We then tried to improve on the TOUCH-based strategy by including temporal parameters about files modified in a change-set such as the developer's code ownership, relative contribution as well as recency and longevity of changes. These parameters are extracted from the project's prior change history, and hence we call this the HISTORY-based strategy. We first validate that these code metrics do, in fact, influence our notion of ideal relevance using statistical tests. We then apply machine learning techniques to train three different classifiers (Naive Bayes, Decision Tree and Rule-Based) to predict whether or not a change-set is potentially relevant for a given developer using these code metrics. With the HISTORY-based strategy, precision improves to more than 75% without severely impacting recall, which is more than 75% as well. These models also appear to generalize across projects.

We then applied these techniques at a different granularity – that of changes to individual Java methods. We analyze 10 projects which contain a high proportion of Java files. Our experiments show that we can further reduce the total volume of notifications while maintaining similar levels of precision and recall of about 75%.

Finally, we augmented HISTORY-based metrics with TEXT from commit messages associated with change-sets in order to model code relevance as a spam filtering problem. This strategy required building separate models per developer and we present results for 196 developers. The augmented model of TEXT + HISTORY performed better than the only HISTORY-based models, giving an average precision of 79.8% and average recall of 78.9%.

To summarize, this paper makes the following contributions:

1. A definition of ideal relevance of code changes based on source code artifact re-visits.
2. An evaluation of different techniques for modeling change relevance, including the TOUCH-based and HISTORY-based strategies, at both the file-level and the method-level.
3. An evaluation of developer-specific TEXT-based models which leverage comments in commit messages.

The rest of this paper is organized as follows. In Section 2, we formulate four research questions for exploring different strategies to model change relevance and also define the code metrics we use in the process. In Section 3, we empirically evaluate our different strategies on 40 open-source projects and present our findings. Section 4 is a discussion on the implications of our evaluation and other practical considerations in building a relevance-based change notification filter. We also discuss factors that might threaten the validity of our results in Section 5. Section 6 presents related work. We then outline potential future work in Section 7 and conclude in Section 8. Also, we have publicly provided the data sets used in this paper on the Web[2].

[2]http://code.comprehend.in:8080/needfeed

2. THEORY

In this section we discuss the theory behind our approach to modeling code relevance for reducing change notifications.

2.1 Ideal Relevance

We design NEEDFEED as a classifier, which, given a change-set containing a set of modified artifacts, decides whether or not it may be relevant to a given developer using one or more of the strategies discussed below. But in order to train such models and evaluate their effectiveness, we need an objective measure of *ideal relevance*, or ground truth, which is the set of changes that a developer ideally would have liked to review.

Developers may find others' changes to be relevant for a variety of reasons, but in general they would be interested in changes to artifacts which they are working on, or which they use [11]. One approach for discovering this information is to track work-items such as tasks or bugs that a developer is working on, but this is not feasible when targeting a heterogeneous landscape of projects which use different task management systems that may or may not link back to source code artifacts. Similarly, we do not capture information about which source code files a developer opens or reads, which can be used to develop *Degree-of-Interest* (DOI) models [10], as this would require an integration with every developer's IDE. The only source of data we rely on is a project's change history containing source code check-ins. Hence, a simple heuristic we use to determine ideal relevance is that if a developer authors some changes to an artifact, then they would have found every change that occurred to this artifact by their team members *since their own last modification to the artifact* as relevant. This definition excludes external changes to an artifact before a developer modifies it for the first time; the intuition being that when a developer starts working on a new artifact they would read its entire contents first, and only then make assumptions about its state which need to be kept up-to-date as their team members make changes.

> **Ideal Relevance:** A change-set Δ committed at time t_1 is considered ideally relevant to a developer d if d has modified some artifact in Δ at times t_0 and t_2, where $t_0 < t_1 < t_2$.

As this definition is based on a heuristic, it may result in the introduction of some bias in our subsequent experiments – this is discussed in Section 5.

2.2 Strategies

Our first approach to reduce notification clutter is a naive strategy, in which developers are subscribed to all changes by their team members that include modifications to artifacts that they themselves have modified at least once in the past. We call this the TOUCH-based strategy. Our hypothesis is that this naive strategy alone will reduce the clutter significantly as compared to the default blind system that broadcasts notifications regarding all changes to every member in the team. The first research question is thus:

> **RQ1:** How effective is a TOUCH-based subscription strategy in reducing change notification clutter as compared to a blind notification broadcast?

The drawback of the TOUCH-based approach is that developers are subscribed to changes involving files which they may have only made some minor edits to, but which are not relevant to them. A common scenario may be when a developer changes a method declaration in a file that they regularly work with, which causes their IDE to automatically refactor all invocations of that method, resulting in a large change-set. The files in which only the method

invocations are changed, are not necessarily relevant to this developer in the context of their work, and hence the change notification subscription is unnecessary. Another example would be if a developer, who did in fact regularly work on a file, changed their role and started working on a new module. The TOUCH-based strategy will continue to notify this developer about changes to artifacts in a module which is no longer relevant to them. Hence, we decided to explore other code metrics that change along with a project's evolution and which may be better indicators of change relevance.

The first metric that we considered was *code ownership*, which is the relative amount of an artifact's current source code that is authored by a given developer. A developer fully owns an artifact if all lines of the artifact's current code have been authored by them. Artifacts whose contents are authored by multiple developers may have multiple owners, with a varying fraction of ownership. Ownership information is easily obtainable using a feature known as *blame* or *annotate* in most SCM tools.

A related metric is *relative contribution*. If an artifact has been modified multiple times, then a developer's relative contribution is the fraction of these changes that were authored by them. There is an interesting interplay between code ownership and contribution. For example, a minor contributor may overwrite all lines of code of a certain artifact and suddenly become a major owner. Conversely a developer may make several contributions, but each time fix only a certain section of the code and hence have low ownership overall.

Apart from these two code metrics, we also consider the exact points in time a developer modifies a given artifact. The principle of temporal locality suggests that developers who have recently modified a given artifact are likely to revisit it soon and hence changes to such artifacts may be relevant to them. On the other hand, those who recently changed the artifact may have just fixed a minor bug, but the people who are really responsible for it may in fact be the developers who first created the artifact or those who were involved in its initial evolution, even if it was a long time ago. Hence, we introduce two new metrics which we call *recency* and *longevity* respectively. Hence, our next research question is:

> **RQ2:** Is there a relationship between code metrics such as *ownership*, *contribution*, *recency* and *longevity*, and our notion of ideal relevance, and if so, how effective is a HISTORY-based strategy in reducing the volume of change notifications?

Further, all these metrics can be calculated at different granularities based on what the term "source code artifact" refers to. The most straightforward granularity is the file-level. The advantage is that this technique is applicable to all types of content. But we would also like to explore if change notification clutter can reduce if code relevance is modeled at a finer granularity of changes to individual methods in source code files. This exploration is motivated by the intuition that, although multiple developers may be working on the same file, they may focus their work on different sections of that file, for example on different methods in a Java class. Hence, a change to one of the methods by one developer may not be interesting to another developer who works on a different set of methods in the same file. The third research question is thus:

> **RQ3:** Does the volume of change notifications reduce by using a finer-granularity of changes to individual methods in a source code file?

Finally, we realize that the problem of reducing change notification clutter is similar to the problem of reducing spam in other information channels such as e-mail. Hence, we also consider the option of augmenting our models with TEXT-based features using commit messages. However, unlike the HISTORY-based models which may be generalizable across developers and even across projects, a TEXT-based classifier needs to learn words that are relevant to a particular developer, and hence a different model must be trained per developer. Our final research question is:

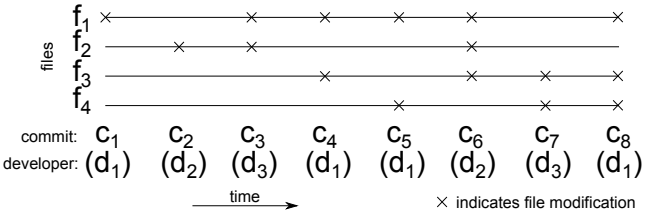

Figure 1: An example time-line of commits (change-sets), their authors and the files changed.

> **RQ4:** Does the addition of TEXT-based features from commit messages improve the usefulness of code relevance models for reducing change notifications?

These research questions will be answered with empirical evaluations in Section 3.

2.3 Code Metrics

We now formally define the different code metrics mentioned above with the help of an example, shown in Figure 1. The four horizontal lines represent time-lines of four files, named f_1 to f_4, which are modified at various points in time through commits c_1 to c_8. Each commit, or change-set, modifies the files which have a $\times$ symbol in their time-line. The developer who committed the change-set is mentioned in parenthesis; this example contains change-sets committed by three developers d_1 to d_3. For example, c_5 is a change-set committed by developer d_1 containing modifications to files f_1 and f_4. Note that although this example shows a simplified linear time-line, in general the change history of a project may be non-linear due to branching and merging.

Let D be the set of developers in a team, A the set of artifacts (e.g. files) in the project, and C the set of commits (or change-sets). We first define some primitives, which are operations usually supported by version control systems such as Git and hence directly obtainable:

- $\text{changes} : C \rightarrow 2^A$ gives the set of artifacts modified in a change-set. For example, $\text{changes}(c_6) = \{f_1, f_2, f_3\}$. The corresponding Git command for this primitive operation is: `git whatchanged COMMIT^!`
- $\text{author} : C \rightarrow D$ gives the developer who committed a change-set. For example, $\text{author}(c_6) = d_2$. This information can be fetched in Git using: `git log COMMIT^!`
- $\text{before} : C \rightarrow 2^C$ is the set of change-sets that the given commit follows. That is, it is the entire change history of the project from the start up to the given commit. For example, $\text{before}(c_6) = \{c_1, c_2, c_3, c_4, c_5\}$. The corresponding Git command for this primitive is: `git log COMMIT^`
- $\text{after} : C \rightarrow 2^C$ is the set of change-sets that follow the given commit, right upto the last commit which resulted in the current state of the project. For example, $\text{after}(c_6) = \{c_7, c_8\}$. The corresponding Git command for this primitive is: `git log COMMIT..`

- totalLines : $C \times A \rightarrow I$ is the number of source code lines that make up the given artifact at the given point in time (specified by the commit). In Git, a file's contents at an arbitrary point in time can be extracted using the command: `git show COMMIT:FILENAME`
- ownedLines : $C \times A \times D \rightarrow I$ is the number of source code lines of an artifact at the given point in time which were authored by the given developer. This information can be easily extracted in Git on a per-file-level using the command: `git blame COMMIT -- FILENAME`

These primitives can be used to define some useful auxiliary relations for a given commit c, artifact a and developer d:

- $\text{pastChanges}(c, a) = \{c' | c' \in \text{before}(c), a \in \text{changes}(c')\}$. For example, $\text{pastChanges}(c_6, f_1) = \{c_1, c_3, c_4, c_5\}$.
- $\text{pastChangesByAuthor}(c, a, d) = \{c' | c' \in \text{before}(c), a \in \text{changes}(c'), d = \text{author}(c')\}$. For example, $\text{pastChangesByAuthor}(c_6, f_1, d_3) = \{c_3\}$.
- *futureChangesByAuthor*$(c, a, d) = \{c' | c' \in \text{after}(c), a \in \text{changes}(c'), d = \text{author}(c')\}$. For example, $\text{futureChangesByAuthor}(c_6, f_1, d_1) = \{c_8\}$.
- $\text{mostRecent}(c, a, d)$ is the minimum distance between c and any commit in $\text{pastChanges}(c, a)$ which is committed by developer d. For example, $\text{mostRecent}(c_8, f_1, d_1) = 2$, because the most recent change to f_1 by d_1 was made at c_5, which is 2 commits ago in the change history of f_1 at that point. If the given developer has not modified the given artifact at all in its history, then the result is ∞. For example, $\text{mostRecent}(c_8, f_4, d_2) = \infty$.
- $\text{leastRecent}(c, a, d)$ is the maximum distance between c and any commit in $\text{pastChanges}(c, a)$ which is committed by developer d. For example, $\text{leastRecent}(c_8, f_1, d_1) = 5$, because the least recent change to f_1 by d_1 was made at commit c_1, which is 5 hops away in the change history of f_1. If the given developer has not modified the given artifact at all in its history, then the result is 0. For example, $\text{leastRecent}(c_8, f_4, d_2) = 0$.

Definition 1. Relative Contribution

$$\text{contribution}(c, a, d) = \frac{|\,\text{pastChangesByAuthor}(c, a, d)|}{|\,\text{pastChanges}(c, a)|}$$

For example, $\text{contribution}(c_6, f_1, d_1) = 3/4 = 0.75$. The range of relative contribution is 0 to 1, where 0 implies the developer has never touched the artifact, while 1 implies that all previous modifications to the artifact were made by this developer alone.

Definition 2. Recency

$$\text{recency}(c, a, d) = \frac{1}{\text{mostRecent}(c, a, d)}$$

For example, $\text{recency}(c_6, f_1, d_3) = 1/3 = 0.33$. The range of recency is 0 to 1, where 0 implies that the developer has never touched the artifact, while 1 implies that the most recent modification to this artifact was committed by the this developer. Between 0 and 1, a high value of recency indicates that the developer was involved in one of the last few changes to the artifact.

Definition 3. Longevity

$$\text{longevity}(c, a, d) = \frac{\text{leastRecent}(c, a, d)}{|\,\text{pastChanges}(c, a)|}$$

For example, $\text{longevity}(c_6, f_1, d_3) = 3/4 = 0.75$. The range of longevity is 0 to 1, where 0 implies that the developer has never touched the artifact, while 1 implies that the first ever change-set involving the artifact was made by this developer. Between 0 and 1, a high value of longevity indicates that the developer was involved in the initial contributions to this artifact.

Definition 4. Code Ownership

$$\text{ownership}(c, a, d) = \frac{\text{ownedLines}(c, a, d)}{\text{totalLines}(c, a)}$$

For example, if the file f_1 contained 21 lines at the state just before commit c_6, and 14 of these lines were authored by developer d_1, then $\text{ownership}(c_6, f_1, d_1) = 14/21 = 0.67$. The range of code ownership is 0 to 1, where 0 implies that none of the lines of code that comprise the source code artifact at this state were authored by this developer, while 1 implies that all the lines of code were authored by this developer. Note that it is possible for a developer to have an ownership of 0 while having non-zero values for contribution, recency and longevity, which would occur if all the lines of code authored by the developer were overwritten by other contributors subsequently.

The four code metrics of ownership, contribution, recency and longevity have been defined at a per-commit, per-developer, per-artifact level. However, when deciding whether or not a change-set is relevant to a given developer, we need to aggregate these metrics over all artifacts that have been modified in the change-set. To prevent missing out on important change notifications, we aggregate these metrics in such a way that will cause a notification to be deemed relevant even if one of the artifacts changed were relevant to the developer. As the numeric equivalent of the boolean or operation is the max function, the aggregation is done as follows:

$$\text{CONTRIBUTION}(c, d) = \max_{a \in \text{changes}(c)} \text{contribution}(c, a, d)$$

$$\text{RECENCY}(c, d) = \max_{a \in \text{changes}(c)} \text{recency}(c, a, d)$$

$$\text{LONGEVITY}(c, d) = \max_{a \in \text{changes}(c)} \text{longevity}(c, a, d)$$

$$\text{OWNERSHIP}(c, d) = \max_{a \in \text{changes}(c)} \text{ownership}(c, a, d)$$

Definition 5. Ideal Relevance

Finally, we define *ideal relevance* in formal terms – $\text{RELEVANT}(c, d)$ is true if and only if $\exists a \in \text{changes}(c)$ such that both the following relations hold:

1. $\text{pastChangesByAuthor}(c, a, d) \neq \emptyset$
2. $\text{futureChangesByAuthor}(c, a, d) \neq \emptyset$

In the example of Figure 1, the change-set c_6 is ideally relevant to developer d_1, but not to d_3. This is because c_6 involves changes to three files f_1, f_2 and f_3, of which f_1 and f_3 have prior contributions by d_1 who also revisits them later at c_8. The developer d_3 has also modified both f_1 and f_2 in the past, but does not revisit them. Developer d_3 does modify f_3 in the future, however that would be the first experience with that file. Hence, this change-set does not fit our definition of ideal relevance for d_3.

Note that while the running example showed a simplified linear time-line of a project's change history, none of the above definitions assume any linear timestamp-based ordering. All operations can be performed on a non-linear change history (that may arise due to branching and merging), which can be visualized as a directed acyclic graph (DAG).

Projects	Age (months)	Source Code Files		# of Commits	# of Developers	# of Notifications		
		Total	Java (%)			BLIND	IDEAL	TOUCH
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
astrid	45	3727	1259 (34%)	5112	19	56944	2686	10323 (18.13%)
atmosphere	45	331	307 (93%)	4121	78	135349	1998	14002 (10.35%)
basex	72	2123	1525 (72%)	7941	74	244926	5583	32315 (13.19%)
bigbluebutton	72	2808	787 (28%)	6248	71	143071	4472	15640 (10.93%)
carrot2	132	1932	642 (33%)	4377	7	14163	1669	3540 (24.99%)
ceylon-compiler	48	7197	4156 (58%)	7758	25	106688	10046	16958 (15.89%)
cgeo	31	1333	504 (38%)	5214	80	186935	14276	31240 (16.71%)
clojure	96	278	152 (55%)	2481	8	3573	859	1760 (49.26%)
dotCMS	22	13381	2485 (19%)	4886	34	59346	4164	12188 (20.54%)
druid	33	2703	2408 (89%)	3665	38	24552	713	5153 (20.99%)
eclim	96	1094	456 (42%)	4289	20	6481	77	195 (3.01%)
elasticsearch	48	3530	2957 (84%)	5685	36	50097	1426	5434 (10.85%)
erlide	84	2818	1220 (43%)	6335	17	41439	2929	7458 (18%)
FBReaderJ	72	1658	663 (40%)	5955	26	54550	2418	6007 (11.01%)
floodlight	25	827	692 (84%)	2112	50	37324	4546	11680 (31.29%)
gephi	59	4654	1288 (28%)	3737	53	54278	307	3414 (6.29%)
h2o	12	1433	399 (28%)	4090	34	49033	3772	9424 (19.22%)
hazelcast	60	1858	1643 (88%)	5039	50	69938	5694	15034 (21.5%)
hbase	84	2287	1885 (82%)	7552	42	143884	24066	49954 (34.72%)
hector	48	492	458 (93%)	2113	103	39241	1396	5379 (13.71%)
hibernate-orm	84	8129	6794 (84%)	4959	56	149460	4789	12528 (8.38%)
hibernate-search	72	1446	1311 (91%)	2805	17	20629	2825	4843 (23.48%)
jogl	132	2126	1174 (55%)	4883	41	85991	1408	11406 (13.26%)
k-9	60	1612	326 (20%)	4675	101	146620	9230	31999 (21.82%)
libgdx	47	5514	2041 (37%)	7221	180	289543	3229	15236 (5.26%)
mahout	72	1339	1194 (89%)	2808	23	32446	4362	9643 (29.72%)
mondrian	144	1435	902 (63%)	3356	64	86465	5221	24042 (27.81%)
mvel	84	482	445 (92%)	2821	24	16085	1750	5137 (31.94%)
netty	60	1040	983 (95%)	5311	38	89729	2409	5770 (6.43%)
nimbus	60	2741	841 (31%)	2449	16	20675	1227	2948 (14.26%)
nutz	60	1606	1232 (77%)	3363	36	36341	1770	5082 (13.98%)
orientdb	46	1699	1412 (83%)	5663	44	34648	2917	8940 (25.8%)
repose	29	2472	1297 (52%)	5105	43	58125	3392	8661 (14.9%)
rhino	180	573	352 (61%)	3010	33	39732	2926	15068 (37.92%)
rstudio	38	3110	1283 (41%)	6818	20	50194	3896	7737 (15.41%)
sensei	47	693	470 (68%)	2072	42	26443	1586	4069 (15.39%)
Spout	41	1201	1051 (88%)	5899	99	246853	12945	37890 (15.35%)
torquebox	57	4274	344 (8%)	4442	54	73107	5460	10696 (14.63%)
vraptor	58	805	682 (85%)	3343	91	70864	1794	8808 (12.43%)
zxing	72	3993	458 (11%)	2567	37	46974	961	4102 (8.73%)

Table 1: The set of 40 open-source projects analyzed.

3. DATA ANALYSIS

In this section, we describe our exploration of different techniques to model code relevance for taming change notifications. We first describe the data set of projects that were analyzed and then describe the setup and results of different experiments that were conducted for answering our four research questions.

3.1 Data Set

For our evaluation, we considered 40 medium-to-large size open-source projects hosted on GitHub. The sample set is derived from a list of top-starred Java projects on GitHub which contain between 2,000 and 8,000 commits each. A summary of the descriptive statistics of these projects is presented in Table 1. The table contains (1) the name of the project as it appears on GitHub, (2) the age of the project in months, (3) the total number of files in its latest state, (4) the number and fraction of files that contain Java source code, (5) the total number of commits from the start of the project and (6) the number of developers who have made at least one commit. The projects we selected varied in different dimensions. Projects range in development history from 1 year (*h2o*) to 12 years (*mondrian*). The number of developers range from 7 in *carrot2* to 180 developers in *libgdx*, with an average of 48 developers across projects. Column (7) contains the sum of the total volume of notifications that would have been sent to the entire team using a blind strategy that notifies every team member of every change. Note that a change notification is only sent to developers other than the committer who have made at least one commit in the past. Developers who join the project late thus receive notifications

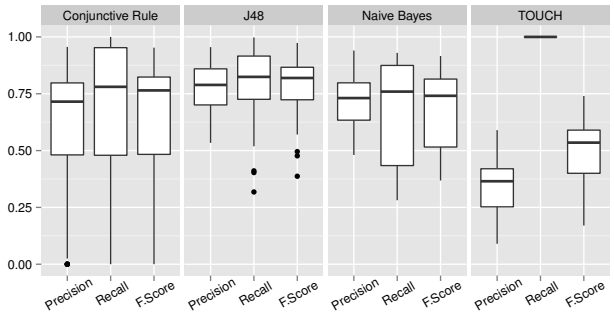

Figure 2: Precision, recall and F-Score of change-set relevance prediction at file-level.

only from that point on. Column (8) contains the ideal volume of notifications that should have been sent, by applying Definition 5 for classifying a notification as ideally relevant or not to the receiver. The values in column (9) will be addressed while answering our first research question.

When comparing different techniques for reducing change notification clutter, we use standard measures such as precision, recall and F-score, which are defined as follows:

Let N_I be the set of all notifications that would have ideally been relevant (by Definition 5) in a project (over all change-sets and all developers). Let N_S be the notifications that were actually found relevant by our model. Then,

$$\text{precision} = \frac{|N_s \cap N_I|}{|N_s|}$$

$$\text{recall} = \frac{|N_s \cap N_I|}{|N_I|}$$

$$\text{F-score} = 2 \times \frac{(\text{precision} \times \text{recall})}{(\text{precision} + \text{recall})}$$

3.2 A Naive Approach

RQ1: How effective is a TOUCH-based subscription strategy in reducing change notification clutter as compared to a blind notification broadcast?

The TOUCH-based strategy is first evaluated at a file-level of subscriptions. For each change-set committed to the repository, we identify the files modified as part of that change-set. We then identify the set of developers, who in the past (prior to this commit), had modified any of these files. A notification would be sent to all these developers (except the developer who committed this change-set). The sum of all such notifications calculated for every commit containing at least one changed file is reported in column (9) of Table 1. The numbers in parenthesis is the fraction of the upper bound (BLIND broadcast) of notifications.

Clearly, the TOUCH strategy alone results in a significant reduction of notification clutter, with the reductions ranging from 51% in *clojure* to 97% in *eclim*. However, many of these notifications may still be irrelevant to the receivers. The right-most column in Figure 2 depicts the distribution of precision, recall and F-score for the TOUCH strategy. Now, every notification that is ideally relevant *will* be sent out by the TOUCH strategy. This is because Definition 5 dictates that for a change to be ideally relevant to a developer, a necessary condition is that $\text{pastChangesByAuthor}$ is non-empty for at least one of the changed artifacts, and this is exactly the sufficient condition for sending out notifications in the TOUCH strategy. Hence, the recall for the TOUCH strategy will always be 100%. However, the average precision is about 35%, indicating that even though the TOUCH strategy reduced the overall notification clutter, it still sends out change notifications to developers who never revisit any of the changed artifacts in the future. Hence, a more intelligent system is needed in order to determine the right set of developers who are likely to find a change-set relevant to them. This brings us to the next research question.

Coefficients	Estimate	Z-Score	Pr $(>\|Z\|)$
Ownership	0.19	12.99	$< 2 \times 10^{-16}$
Contribution	0.95	42.33	$< 2 \times 10^{-16}$
Recency	1.56	124.46	$< 2 \times 10^{-16}$
Longevity	0.03	2.77	0.0055
Odds Ratio			
Ownership	Contribution	Recency	Longevity
1.20	2.58	4.78	1.03

Table 2: Results of Binary Logistic Regression on code metrics in predicting *ideal relevance*.

3.3 A Machine Learning Approach

RQ2: Is there a relationship between code metrics such as *ownership*, *contribution*, *recency* and *longevity*, and our notion of ideal relevance, and if so, how effective is a HISTORY-based strategy in reducing the volume of change notifications?

To answer this research question, we calculate the metrics *ownership*, *contribution*, *recency*, *longevity* and *ideal relevance* using Definitions 1–5 for each change-set for each developer who would have received a TOUCH-based notification. Our aim is to be able to use the code metrics to *predict* whether or not a change-set is ideally relevant.

We first determine the statistical significance of each of these metrics in predicting ideal relevance. We then train three different classifiers and evaluate their effectiveness in predicting whether a developer would find a change-set relevant or not.

3.3.1 Statistical significance of code metrics in predicting ideal relevance

For the 40 projects, we had a total of 491,703 data points for our analysis. The four code metrics are predictor variables while *ideal relevance* is the outcome variable with binary values: 0 indicating that the change-set is not relevant for the developer and 1 indicating it is relevant. Since the outcome variable is binary (categorical) and our predictor variables are continuous, we applied *binary logistic regression* to evaluate the influence of the predictor variables on the outcome.

Table 2 presents the results of binary logistic regression listing the regression coefficients, estimates, Z-score and p-value. We find three predictors (*ownership*, *contribution* and *recency*) significantly influencing the outcome variable of ideal relevance with p-values $< 2 \times 10^{-16}$, while *longevity* influences the outcome with a relatively lesser significance. The odds ratio represents the magnitude of influence. For example, for every unit change in *recency*, the odds of the notification being relevant increases by 4.78. The odds ratio suggests *recency* and *contribution* as major predictors of *ideal relevance* followed by *ownership* and *longevity*.

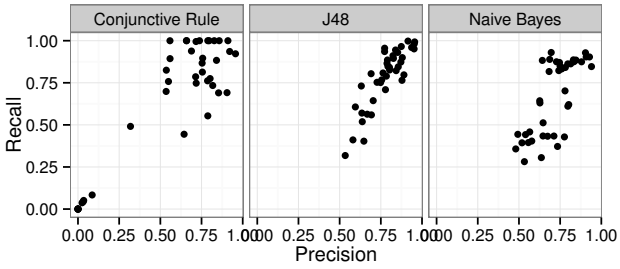

Figure 3: Precision and Recall for three classifiers using a file-level HISTORY-based strategy.

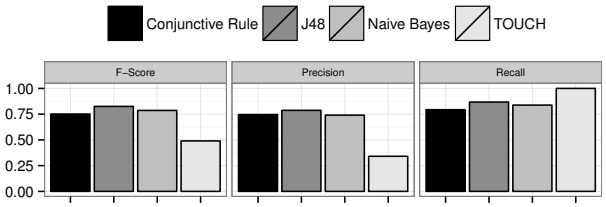

Figure 4: Performance of generalized (project-independent) file-level HISTORY-based models.

We also performed the *Wald Test* [24] to determine how well each predictor is significant. This test identifies how well the model fits by including a predictor variable as opposed to a model which excludes that predictor. We repeated the test for each predictor, and for all the predictors the *Wald Test* resulted in a significant *chi-square* with a p-value of less than 0.05.

3.3.2 Change relevance classification problem

We now evaluate different classification techniques for predicting whether a change-set would be relevant to a given developer or not based on temporal code metrics of the modified artifacts. Collectively, this represents the HISTORY-based strategy for reducing notification clutter.

Effectiveness of classification techniques: We used three classification techniques available in WEKA [13] for our study: (1) *Naive Bayes* (a probabilistic classifier), (2) *Conjunctive Rule* (a rule-based classifier) and (3) *J48* (which is an implementation of the C4.5 decision tree algorithm). We applied the standard 10-fold cross-validation technique and Figure 3 shows the scatter plot of precision and recall for different classification techniques. Each point in the plot represents the average precision and recall value aggregated over the 10-fold cross validation for each project. Intuitively, classification techniques that place majority of the data points in the top-right quadrant (where precision and recall are greater than 0.5) can be considered as performing well. Correspondingly, *J48* performs the best by predicting change-set relevance with precision and recall greater than 0.5 for 37 projects. *Conjunctive Rule* performs extremely well in few cases, where it predicts with a recall of 1 and precision greater than 0.5, but gives very low precision and recall for some other projects, and is hence not consistent. The performance of *Naive Bayes* is between the other two techniques, having at least either of precision or recall greater than 0.5, across all projects.

Improvement over the TOUCH strategy: When compared to the TOUCH strategy, the two HISTORY-based classifiers *J48* and *Naive Bayes* performed better in terms of precision and F-score (as shown in the first three columns of Figure 2). The median precision values observed for *J48* and *Naive Bayes* were greater than 70% which is a factor of two more than the precision observed for TOUCH. The entire range of precision values was higher than the corresponding precision values for TOUCH, indicating that *J48* and *Naive Bayes* outperform TOUCH with respect to precision. Although HISTORY-based classifiers report lower recall values when compared to the TOUCH-based strategy (where recall is 100% by definition), the *J48* technique predicts with a recall greater than 50% in almost all cases. The *Conjunctive Rule* classifier is not consistent and performs as bad as TOUCH in some projects. As regards to the total volume of notifications, while the TOUCH strategy resulted in a 82% reduction in clutter (compared to the BLIND broadcast strategy), the HISTORY strategy reduced clutter by more than 90% using either of *J48* or *Naive Bayes*.

Projects	# of Notifications		
	BLIND	IDEAL	TOUCH
atmosphere	61951	284	2325 (3.75%)
druid	17277	308	1609 (9.31%)
floodlight	29092	1116	4020 (13.82%)
hazelcast	52759	2127	7457 (14.13%)
hector	25466	373	2018 (7.92%)
hibernate-orm	83118	1553	5348 (6.43%)
mahout	23388	1567	4349 (18.59%)
mvel	9505	96	1015 (10.68%)
netty	74222	1059	2772 (3.73%)
vraptor	35867	536	2869 (7.98%)

Table 3: Subset of Java projects analysed for RQ3.

Generalization of the classification models: We further evaluated if a single HISTORY-based model trained using aggregate data from across projects can predict relevance for change-sets in other projects individually. We present the results in comparison with the TOUCH strategy in Figure 4. The vertical bars represent the corresponding precision, recall and F-score for each classification technique. It is evident that *J48* outperforms the TOUCH strategy as well as other HISTORY-based classification techniques, by predicting relevance correctly with approximately 75% precision, recall and F-score, on average.

3.4 Granularity of Changes

> **RQ3:** Does the volume of change notifications reduce by using a finer-granularity of changes to individual methods in a source code file?

We now explore the effectiveness of the above techniques at the method-level and compare it to the results at the file-level.

Data set and naive TOUCH strategy: We chose 10 projects from our original data set 40 projects, which had a high (more than 80%) proportion of Java files. For each change-set, we identified the set of changed methods by comparing the differences in the abstract syntax trees of Java source files before and after the change. This was implemented using the Eclipse Java Development Toolkit. The summary statistics for these projects is shown in Table 3. A comparison of Table 3 and corresponding rows of Table 1 make it evident that there is a significant reduction in the notification volume (for BLIND as well as TOUCH) at the method-level. The BLIND volume is lower than that at the file-level because not all changes

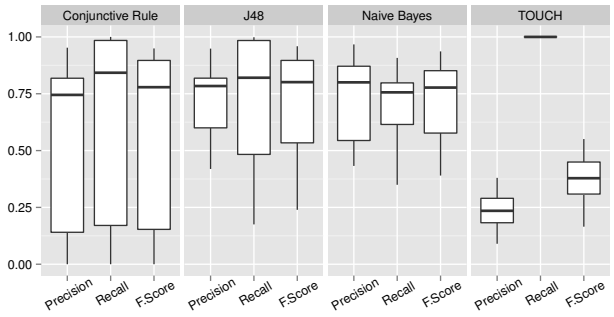

Figure 5: Precision, recall and F-Score of change-set relevance prediction at method-level.

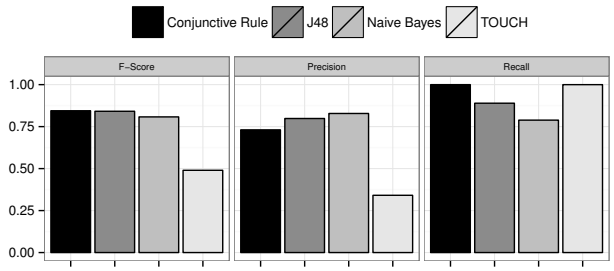

Figure 6: Performance of generalized (project-independent) method-level HISTORY-based models.

include modifications to Java methods. But the reduction in the notifications sent out by the TOUCH-based strategy is much more. The difference between method-level and file-level effectiveness of TOUCH can be realized by comparing the numbers in parenthesis, which is the percentage of the BLIND notifications which were actually sent out. The fraction is lesser in the method-level, because developers are only subscribed to changes to individual methods which they have touched, and not just all changes to the file in which they have modified some method.

Improvements with the HISTORY strategy: We also calculated code metrics such as *ownership*, *contribution*, *recency* and *longevity* for method-level changes. Note that while calculating these metrics we only considered historical changes to a given method, and hence all these values will be different for each change-set as compared to the file-level. We then applied the three classification techniques for the HISTORY-based approach, viz. *Naive Bayes*, *Conjunctive-Rule* and *J48*. Figure 5 is a box plot of the precision, recall and F-score across different classification techniques for the 10 projects. Among the classification techniques both *J48* and *Naive Bayes* performed equally well at the method-level granularity. *J48* performed well in terms of precision, while *Naive Bayes* did well from a recall perspective. In contrast, *Conjunctive Rule* spanned all ranges of precision and recall, proving that it was not consistent at method-level granularity as well.

Generalization of the classification models: For the method-level granularity, we also trained generalized project-independent models using the three classifiers to see if the code metrics can predict relevancy across projects. Figure 6 shows the results of these models. The HISTORY-based techniques seem to perform well when using a generalized method-level model as well.

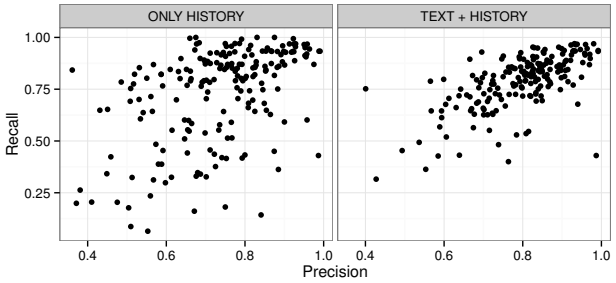

Figure 7: Precision and recall for developer-specific models for 196 developers.

3.5 Text-based Spam Filtering

RQ4: Does the addition of TEXT-based features from commit messages improve the usefulness of code relevance models for reducing change notifications?

Our final exploration was to consider the problem of reducing change notifications as a spam filtering problem, and build developer-specific models of code relevance that use, in addition to HISTORY-based metrics, TEXT-based features from commit messages that may indicate items of interest to developers.

Data set and methodology: Since the TEXT-based strategy learns words of relevance specific to a developer, this approach requires building one model per developer. For this purpose, we require enough data about a developer's prior change history, with enough examples of ideally relevant and non-relevant changes. Hence, we curtailed our data set of the 40 open-source projects and built models for only those developers who had at least 200 relevant and non-relevant changes in their history. Only 196 of the 1,822 developers in our data set met this criteria.

To build a TEXT-based code relevance model, we extracted textual comments that are associated with each change-set (also known as commit messages). These messages typically include terms that refer to the module in which the change occurs or contains words describing tasks or bugs that are addressed in the change. The commit messages were split into a bag-of-words using WEKA's `StringToVector` filter, which also removes stop-words and stems the resulting words to their root. The resulting bag-of-words formed the set of input variables for the TEXT-based code relevance models.

In order to objectively understand the effect of incorporating text in our modeling of code relevance, we trained two models per developer: one using only HISTORY-based metrics and one using both HISTORY + TEXT features. Note that the models using only HISTORY here are different from the per-project or project-independent models presented in Section 3.3. When training developer-specific models, we only calculate HISTORY metrics for that developer for each change-set in the project that the developer is contributing to. The evaluation of these models was done using the same ten-fold cross-validation technique as used previously.

Results: Figure 7 shows the results of our developer-specific models as scatter plots of precision and recall. As it is evident from the figure, the augmented models that incorporate TEXT perform slightly better (average precision and recall of 79.8% and 78.9% respectively) than the purely HISTORY based models (average precision and recall of 73.8% and 73.1% respectively), indicating that TEXT from commit messages can be useful to model code relevance.

4. DISCUSSION

We now discuss the implications of the above results for developing NEEDFEED, a relevance-based change notification tool.

Best Classifier: There is no clear winner between the various classification techniques (though *J48* and *Naive Bayes* seem to provide more consistent results) but it is clear from the evaluation that the HISTORY-based strategy is in general far better than the TOUCH strategy in reducing notification clutter. Note that, from an application point-of-view, the moderately low values of recall for the HISTORY-based strategies are not completely detrimental because a developer can still view change history of any artifact before modifying it. However, this is time consuming and may result in a loss of productivity, which is equivalent to that of sifting through a large list of irrelevant notifications. Hence, NEEDFEED would need to consider a trade-off between a large amount of notification clutter containing surely irrelevant changes and the off-chance of missing out on potentially relevant changes.

File-level vs. Method-level: Although it is clear that the volume of change notification clutter reduces significantly when using a finer granularity of individual methods, not all change-sets will contain changes to source code. Even projects that are exclusively written in one programming language contain certain configuration files or other documents that are collaboratively developed and for which a fine granularity may not be discernible.

Generalizability: From our study over the sample set of 40 projects, we found that a generalized model works as well as project-specific models. This is an interesting property which will allow new projects to leverage NEEDFEED without requiring a long change history to bootstrap the relevance model.

Text Based Filtering: While the combined TEXT + HISTORY approach seems to provide better results on average than using only HISTORY metrics, there is a constraint that developer-specific models can only be built when there is significant amount of change history for the developer which can be used for training.

Prescribed Strategy: Overall we feel the best strategy is a hybrid one which can leverage data as it becomes available. For new projects, the project-independent HISTORY-based model (which is generic and learned from other projects) could suffice, but as the project grows a project-specific model can be trained using its own change history. Similarly, as individual developers increase their participation, developer-specific models can be built for them incorporating TEXT from commit messages. In each case, if the files modified in a change-set are of a type whose source can be parsed, we can apply method-level history tracking. For other types of content, the model can fall back on file-level granularity.

5. THREATS TO VALIDITY

In this section, we list the most important threats and limitations of our empirical study.

Construct validity: We faced some minor issues while extracting the required data from Git repositories for our study. For example, developers who used different names or email addresses across their commits would have been considered as distinct people and hence distinct relevance values would have been calculated for each of them. Also, we did not track renaming of artifacts and hence such operations appear as a deletion of one artifact followed by the creation of another. These issues would affect our precision and recall but we expect such occurrences to be the exception rather than the norm.

Internal validity: Our definition of *ideal relevance* suffers from a horizon effect that would likely introduce biases in the preicision and recall values reported in our experiments. This is because for commits towards the end of the time-line there is no opportunity to identify *future* code revisits by developers in order to classify them as ideally relevant or not. Consider a notification about a change-set Δ that is sent to a developer d at time t_1. If we were to train or evaluate our model using historical data at some time t_2 (where $t_2 > t_1$) and if d has not modified any artifact in Δ between t_1 and t_2, then we will mark that change-set as ideally not relevant to d. However, it may be possible that d is, in fact, responsible for the artifacts changed in Δ, and that after a review of that change-set d makes some modifications in the future at some point t_3 (where $t_3 > t_2$). Thus, our training may suffer and our evaluation may report spurious false positives and true negatives. Note that merely restricting our training and evaluation to a subset of commit history before some mid-way point in time t_m will not resolve this issue, as this period may still include horizons for those artifacts that were last modified only before t_m. A possible solution may involve carefully ignoring changes to a select subset of artifacts in every change-set that are close to their own horizon. We leave the detailed investigation of the horizon effect as future work.

External validity: Our study was conducted on a set of 40 open-source projects hosted on GitHub. Although the model seems to generalize over this set of projects, it may or may not perform as well on different kinds of projects, such as small private projects in a tightly controlled commercial setting having a team size of just a handful of developers. Such projects can hence use the TOUCH-based strategy during their initial stages and migrate to a project-specific model as the project evolves.

6. RELATED WORK

There is a large amount of existing literature that has studied the problems that arise due to the overheads of collaboration in software development as the team size increases [3, 7], which impedes development [5, 9, 15] and increases defects [8, 18, 21]. The impact of code metrics such as ownership and contribution have also been shown to be useful in studies of software quality [2, 22].

A number of prior research work has tackled issues similar to the ones we address in this paper, specifically that of (1) information clutter in a developer's workspace, (2) code change notifications in particular and (3) other methods of keeping developers updated about their teammate's activities. We address each of these categories of related work and clarify how our work is different.

Information Clutter and Notification Spam: Mukherjee and Garg [20] conducted a study to measure information clutter arising from notifications about changes to work-items in Rational Team Concert. Their tool, TWINY, uses machine learning techniques on historical data about work items mined from the software repository to predict whether a notification will prompt a response from its recipient. Similarly, Ibrahim et al. [17] developed personalized models to automatically identify discussion threads that a developer would contribute to based on their previous contribution behavior. They built a composite model leveraging both Naive Bayesian and Decision Tree classifiers. Our work is partially inspired by these approaches but is different in scope and methodology. We focus solely on clutter arising from source code check-in notifications, and hence leverage different kinds of temporal metrics as well as different granularities of changes.

Relevancy in source code changes: Holmes and Walker [16] address the problem of modeling relevancy in code changes in their tool – YooHoo. However, this tool is mainly targeted at external changes occurring in components or libraries on which a developer's own work depends on. Due to this different scope, their tool looks for events such as a change in the API of imported libraries or the creation of a new repository branch (possibly indicating a

new release) or change in JavaDoc (possibly a change in expected behavior), and filters out minor changes to the implementation of such external components. Our study is restricted to the problem of determining relevance of changes to artifacts in a given project which are worked on by multiple developers, and hence we model code relevance differently.

Change relevance has also been mentioned as a use case in Fritz's PhD thesis [10], in which he presented a *Degree-of-Knowledge* (DOK) model for capturing a developer's familiarity with source code artifacts. Our work differs from this thesis in two major ways. Firstly, the DOK model was trained with seven developers who answered a questionnaire regarding their knowledge of source code artifacts, and the application of this model for estimating relevance of changes was evaluated by recommending a total of six bug reports to three developers. Our training and evaluation are completely automated and use a much larger data set of 40 projects, 1,822 developers and 491,703 data points. Secondly, the variables used to model code relevance are different. The DOK model uses two components: (1) a *Degree-of-Authorship* (DOA), which is based on whether a developer was the first author of a file and on the absolute number of changes delivered/accepted by them, and (2) a *Degree-of-Interest* (DOI) which is calculated by monitoring a developer's activity within an IDE. Our HISTORY metrics are similar to the DOA but are all continuous and normalized, and we do not measure IDE activity at all and hence have no DOI component, though it should be easy to incorporate this variable if the data was available. In any case, Fritz reported that the DOI variable was not significant during the training of the DOK model.

Conflict detection from concurrent modifications: A number of tools have been developed to prevent conflicts arising from multiple developers modifying source code artifacts simultaneously. These tools include CollabVS [14], Crystal [4], Palantir [23] and Continuous Merging [12]. However, such tools prevent conflicts that can possibly arise in the future by scanning yet-uncommitted changes that are being authored in parallel by two or more developers. Also, these tools are usually restricted to determining conflicts that can be automatically detected, such as those that break builds or tests. The scope of our work is different, as code check-in notifications potentially prevent inconsistencies that are authored by a single developer, but which can be detected by another developer who manually reviews the change. Also, check-in notifications are fired only after the changes are fully committed to the shared repository. Our work can thus be used in conjunction with the aforementioned tools since the addressed problems are distinct.

7. FUTURE WORK

NEEDFEED is currently implemented as a notification filter at two granularities: file-level and Java method-level. As discussed previously, although a model based on fine-granularity is better at reducing notification spam, it has to fall back to the file-level for content-types which it cannot parse. NEEDFEED thus needs to be extended with more front-ends to be effective for projects written in other languages such as C++ or JavaScript.

Currently, NEEDFEED is only capable of classifying change-sets as relevant or irrelevant to a developer in a binary fashion. Our final aim is to provide a personalized change-notification feed, which would contain, for each item, a description of why exactly the change is relevant to a given developer. Changes that are not deemed relevant may be presented in a separate, unobtrusive list, which can be examined on-demand. A mock-up of such a personalized feed is shown in Figure 8. Our tool can be integrated with other filters such as YooHoo [16] which consider different types of change relevancy, thus providing an integrated notification stream.

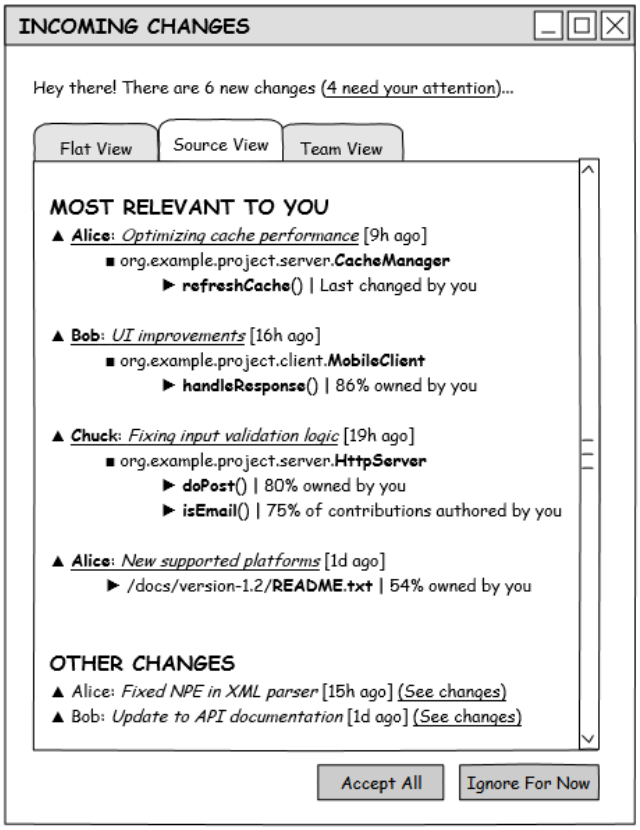

Figure 8: A mock-up of a personalized change notification stream powered by NEEDFEED.

We also envision other applications that can leverage the code relevance models of NEEDFEED. For example, the workspace of a developer in their IDE often contains a large number of files, usually everything contained by the project the developer is working on. Simply locating relevant files to open can sometimes be a non-trivial task. An alternate view of the workspace which first shows *relevant* artifacts and optionally allows navigation of the remaining files may be helpful in improving a developer's productivity. Also, as our models capture the likelihood that a developer may modify an artifact in the future, another application that is conceivable is forecasting, where project managers can make estimates about the likely workloads of developers in the near future based on their recent change history.

8. CONCLUSION

Software developers have consistently reported that the state-of-the-art in collaborative development tools are inadequate in meeting their information needs concerning changes committed by other developers, due to a lack of relevance in the notification stream.

In this paper, we have described our exploration of various techniques to model code relevance towards providing a personalized change feed for every developer. Our experiments on historical data of 40 open-source projects indicate that models that incorporate data mined from a project's software repository are useful in reducing notification clutter (by over 90% on average) with high levels of precision and recall (over 75% each on average).

We believe that the inclusion of personalized change notifications in collaborative development tools can be very effective in improving the productivity of software developers and hence has the potential to positively impact the success of any software project as a whole.

9. REFERENCES

[1] GitHub. `https://github.com`. Accessed: April 2014.

[2] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. An analysis of the effect of code ownership on software quality across Windows, Eclipse, and Firefox. Technical Report MSR-TR-2010-140, Microsoft Research, 2010.

[3] F. P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[5] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM.

[6] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '03, pages 45–49, New York, NY, USA, 2003. ACM.

[7] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, Nov. 1988.

[8] B. Curtis, E. M. Soloway, R. E. Brooks, J. B. Black, K. Ehrlich, and H. R. Ramsey. Software psychology: The need for an interdisciplinary program. *Proceedings of the IEEE*, 74(8):1092–1106, 1986.

[9] J. A. Espinosa, R. E. Kraut, F. J. Lerch, S. Slaughter, J. D. Herbsleb, and A. Mockus. Shared mental models and coordination in large-scale, distributed software development. In *ICIS*, volume 2001, pages 513–518, 2001.

[10] T. Fritz. *Developer-Centric Models: Easing Access to Relevant Information in a Software Development Environment*. PhD thesis, Ludwig-Maximilians-Universität München, 2011.

[11] T. Fritz and G. C. Murphy. Determining relevancy: how software developers determine relevant information in feeds. In *CHI*, pages 1827–1830, 2011.

[12] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[14] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 178–187, Washington, DC, USA, 2008. IEEE Computer Society.

[15] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, 29(6):481–494, 2003.

[16] R. Holmes and R. J. Walker. Customized awareness: Recommending relevant external change events. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 465–474, New York, NY, USA, 2010. ACM.

[17] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan. Should I contribute to this discussion? In *Proceedings of the 7th International Working Conference on Mining Software Repositories*, MSR '10, pages 181–190, 2010.

[18] T. Illes-Seifert and B. Paech. Exploring the relationship of history characteristics and defect count: an empirical study. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 11–15. ACM, 2008.

[19] M. Kim. An exploratory study of awareness interests about software modifications. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 80–83, New York, NY, USA, 2011. ACM.

[20] D. Mukherjee and M. Garg. Which work-item updates need your response? In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 12–21, Piscataway, NJ, USA, 2013. IEEE Press.

[21] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[22] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[23] A. Sarma, D. F. Redmiles, and A. van der Hoek. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Trans. Software Eng.*, 38(4):889–908, 2012.

[24] A. Wald. Tests of statistical hypotheses concerning several parameters when the number of observations is large. *Transactions of the American Mathematical Society*, 54:426–482, 1943.