Towards Content-driven Reputation for Collaborative Code Repositories

Andrew G. West
University of Pennsylvania
Philadelphia, PA, USA
westand@cis.upenn.edu

Insup Lee
University of Pennsylvania
Philadelphia, PA, USA
lee@cis.upenn.edu

ABSTRACT

As evidenced by SourceForge and GitHub, code repositories now integrate Web 2.0 functionality that enables global participation with minimal barriers-to-entry. To prevent detrimental contributions enabled by crowdsourcing, reputation is one proposed solution. Fortunately this is an issue that has been addressed in analogous version control systems such as the *wiki* for natural language content. The WikiTrust algorithm ("content-driven reputation"), while developed and evaluated in wiki environments operates under a possibly shared collaborative assumption: actions that "survive" subsequent edits are reflective of good authorship.

In this paper we examine WikiTrust's ability to measure author quality in collaborative code development. We first define a mapping from repositories to wiki environments and use it to evaluate a production SVN repository with 92,000 updates. Analysis is particularly attentive to reputation loss events and attempts to establish ground truth using commit comments and bug tracking. A proof-of-concept evaluation suggests the technique is promising (about two-thirds of reputation loss is justified) with false positives identifying areas for future refinement. Equally as important, these false positives exemplify differences in content evolution and the cooperative process between wikis and code repositories.

Categories and Subject Descriptors

H.5.3 [**Group and Organization Interfaces**]: *collaborative computing, computer-supported cooperative work*

Keywords

WikiTrust, wikis, code repository, SVN, reputation, trust management, content persistence, code quality.

1. INTRODUCTION

Version control systems (VCS), particularly those aimed towards source code development (*e.g.,* CVS, SVN, Git) have long supported local collaboration among known and trusted parties. More novel is opening these repositories to a global and anonymous set of participants. This trend can be seen with online repository hosts such as SourceForge and GitHub. Consider also `vehicleforge.mil` [5], a DARPA led effort to crowdsource a next-generation military vehicle. While such broad collaboration is time and cost effective it can also invite inefficient, buggy, or malicious contributions.

This not an issue unique to code repositories. Consider English Wikipedia which operates on a VCS (the *wiki* platform) where ≈7% of edits are unconstructive ("vandalism"). To this end, "content-driven reputation" was developed and encoded as the WikiTrust [2, 3] algorithm. WikiTrust's assumption is that actions which persist across subsequent revisions are indicative of reputable authors (see Sec. 2).

WikiTrust's hypothesis has been shown effective for natural language content on wikis, however, one can imagine it might hold true in many collaborative settings, including source code repositories. To investigate this we define a mapping between repository software and wiki platforms. Then, we apply WikiTrust over a case study SVN repository containing some 92,000 updates (Sec. 3).

WikiTrust plots user reputations across repository history. Our analysis scrutinizes reputation loss on these graphs, gleaming additional information from bug tracking and commit comments. Our goal is to determine whether the reputation penalty was justified, and we estimate those decrements were warranted in about two-thirds of cases (Sec. 4). False positives are given particular attention since they suggest how WikiTrust might be refined for use in code repositories. Fortunately, we are able to identify several redundant, detectable, and correctable sources of error (Sec. 5).

Should these suggestions produce a sufficiently accurate and robust reputation those values can be integrated in ways that remove administrative burden, heighten end user trust, and further promote "openness" in development. For example, additional scrutiny could be given to contributors with low reputation or the most productive users rewarded.

While the development of a refined version remains a work in progress the shortcomings of our straightforward WikiTrust application are valuable in and of themselves. Erroneous reputation events exemplify differences in the content evolution of wikis and code repositories. Collaborative semantics are not uniform across cooperative settings and research should be attune to these subtleties.

2. BACKGROUND & RELATED WORK

Discussion begins by reviewing the WikiTrust algorithm (Sec. 2.1) before examining related literature (Sec. 2.2).

2.1 WikiTrust Algorithm

Assume a VCS environment consisting of a set of documents $\{d_0, d_1 \dots d_n\}$. Each document d has a version history $ver\_hist(d_x) = [d_{x.0}, d_{x.1} \dots d_{x.m}]$ where $d_{x.0}$ is an empty document and $d_{x.m}$ is the most recent version. The transition $d_{x.y} \rightarrow d_{x.y+1}$ is termed an edit/revision and each has an author associated with it, $a_{x.y+1}$.

The WikiTrust algorithm [2, 3] begins by initializing authors with a default reputation. Whenever a new document version $d_{x.y+1}$ is created, its relationship with the previous n versions is investigated, possibly updating the reputations of the n authors $\{a_{x.y-n} \dots a_{x.y}\}$. These authors' reputations are updated according to how much of their actions "persist" to the new version per a metric of *edit survival*.

Edit survival uses an edit distance computation to quantify the similarity of an author's (prior) version to the current one. This captures not only the survival of novel authored content (*i.e., text survival*) but also reorganization and content removal. The size of a reputation delta is proportional to the degree of change and weighted by the reputation of the current version's author.

WikiTrust assumes that when an author edits a document but leaves portions intact, those portions have his/her implicit endorsement. This approach allows feedback regarding contributions to be gleamed from typical system interaction rather than explicit mechanisms. Moreover, an editor puts his/her own reputation at stake when they cast judgment on others, since he/she becomes part of the edit history.

In the interest of brevity we point readers to [2, 3] where WikiTrust is formally described. WikiTrust's successful deployment motivated its application to code-centric VCS.

2.2 Related Work

WikiTrust succeeds because it captures and quantifies the evolution of wiki content [8]. Harnessing these same properties an alternative approach [10] models a wiki as a Dynamic Bayesian Network but uses author reputation as an input.

Shifting focus towards software engineering and repositories, there have been numerous attempts to evaluate code quality in a static fashion. Metrics such as lines-of-code, nesting depth, and comment usage have all been applied, with [9] using them to evaluate open source development.

More relevant are approaches leveraging some notion of historical code evolution. The Dynamine system [7] mines revision history to find function co-occurrence *patterns* and locate existing/future violations thereof (*e.g.,* `file.close()` should generally follow a `file.open()`). A separate work [6] quantifies multiple dimensions of the collaborative history (*e.g.,* code age, number of contributors, module dependencies) to predict fault prone code. Relative to WikiTrust these approaches appear more focused on system wide evolution than attributing actions to any particular author/developer.

Additionally, Cataldo [4] has analyzed distributed development in particular, quantifying common sources of error and offering advice for corrective collaborative tools.

3. FROM REPOSITORY TO REPUTATION

To evaluate WikiTrust over code repositories we choose to "import" a repository onto a wiki platform in order to make use of an existing WikiTrust implementation. We define a general model to make this transformation (Sec. 3.1) before making some practical adaptations in our implementation of the technique over a case study SVN repository (Sec. 3.2).

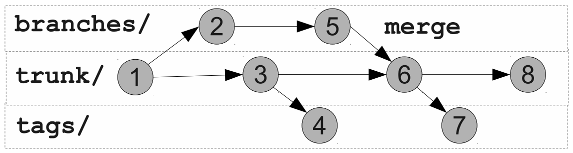

Figure 2: Repository revision model.

3.1 Repository to Wiki Model

We now describe the transformation of a code repository into a wiki representation. Generic discussion suffices given that instantiations of both repositories (*e.g.,* CVS, SVN, Git, *etc.*) and wikis (*e.g.,* Mediawiki, PmWiki, *etc.*) have an expected set of functionality. Our writing assumes some familiarity with both paradigms (see also Fig. 2).

Our aim is to "replay" the history of a repository into the wiki format. Simple repository actions (add, modify, and delete) have a straightforward mapping that requires no description. Elsewhere, minor accommodations are needed:

- *Multi-file check-in*: An atomic repository check-in can involve multiple files, while wikis support only single document edits. A single check-in can be replayed as multiple edits, and this is inconsequential so long as no reputation updates occur internal to a batch.
- *Branch-merge*: It is our design decision only to replay the primary `trunk/` line of development. Branches are intended for experimental coding and their inclusion might bring unwarranted WikiTrust punishment. As a consequence, if a branch is merged back into the `trunk/` all changes are attributed to the merging user (and this is taken as an implicit recommendation).
- *Tagging*: Tagging actions are ignored as they are only snapshots capturing no real authorship value.

3.2 Implementation and Practical Issues

Our transformation is next applied to a case study repository. We chose to use the Mediawiki [1] SVN repository which as of our late 2011 analysis had ≈92,000 check-in actions and 420,000 `trunk/` file versions. This code history is replayed into an instance of the Mediawiki [1] wiki engine (yes, the same software is both *utilized* and *analyzed*), a platform with an existing WikiTrust implementation [2].

The workflow begins by using `svnsync` to create a complete local repository copy (*not* simply a checkout). From that, `[svn log]` yields metadata that acts as a script for the replay phase where the `[svn cat filepath@revision_id]` syntax is used to generate all unique file versions in a chronological fashion. These versions are piped to the Mediawiki API and associated with the proper user. When done, the wiki DB is given to an implementation of the WikiTrust algorithm modified to output verbose reputation data.

Several practical adjustments are made to the process. As our interest lies with programming code we make a best-effort to replay the histories of only PHP code files (the core Mediawiki language). This means we exclude: (1) binary files, which are often not token-izable, (2) non-PHP text files, such as documentation, and (3) PHP internationalization files, which consist only of localized strings. We retain WikiTrust's white space delimitation and discuss alternative tokenization/normalization schemes in Sec. 5. Tab. 1 shows statistics about both the original and "filtered" repository.

PROPERTY	ORIG	MOD
authors	326	271
check-ins	91,808	53,715
file versions	585,629	117,432
... in `trunk/`	420,613	117,432
unique paths	138,741	7,521
... to PHP file	56,063	7,521
... w/2+ auth.	27,330	5,477

Table 1: Mediawiki SVN statistics per late 2011 for the entire repository (ORIG) and a reduced version used for analysis (MOD).

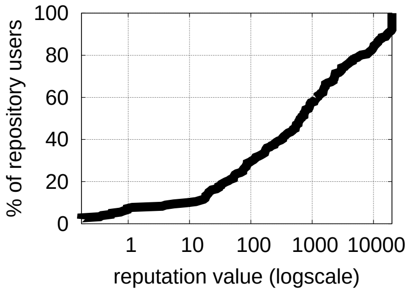

Figure 1: CDF of user reputations.

ID	ΔSIZE	REVS(#)	REP#
U_1	7.0MB	4,243(8)	1 (tie)
U_2	6.2MB	6,825(4)	14
U_3	6.2MB	7,106(3)	1 (tie)
U_4	5.1MB	9,546(1)	1 (tie)
U_5	4.2MB	7,221(2)	15
U_6	3.9MB	3,270(10)	25

Table 2: Comparing participation metrics: Users are sorted by the size of all his/her revision deltas (ΔSIZE) and ranked (#) by number of file revisions (REVS) and final reputation (REP).

4. CASE STUDY RESULTS

This section reports on the output of applying WikiTrust to the Mediawiki SVN. We begin with broad statistical observations (Sec. 4.1) before examining reputation accuracy (Sec. 4.2). Lastly, false positives are discussed (Sec. 4.3).

4.1 General Statistics

The reputations WikiTrust computes lie on $[0, 20000]$ with the upper bound being set to prevent "god-like" users whose actions carry excessive weight. Reputations grow in a logarithmic fashion and should be interpreted relatively.

A document must have 2+ authors to contribute to reputation; one cannot influence their own reputation. However, a single edit can affect the reputations of $n = 10$ prior document authors (our search depth). Over the SVN history these criteria produce ≈40k reputation updates, quantifying the behavior of 183 (of 271) authors. Of these updates 87% are reputation *increments*, a non-surprising result given the productive nature of the development community (see Fig. 3). Fig. 1 displays the reputation distribution for all authors. The graph suggests a typical exponential model of participation as few users contribute much value.

We can also compare reputations to conventional metrics as Tab. 2 shows. Those who contribute the most content do not necessarily have the best reputations; WikiTrust is not simply duplicating the results of simpler measures. Consider user U_2 who contributed the 2^{nd} most "change" but ranked 14^{th} in terms of reputation. Similarly, one user with maximal reputation ranked 78^{th} in file revision quantity.

Of course, final reputations are just a snapshot of a dynamic progression. Fig. 4 plots the reputation history for some example users. Editors E_1 and E_2 are prolific, long-term contributors and their reputations reflect this with their generally upward trend (E_1 sits on the reputation ceiling). At the other extreme are users like E_4 whose reputations tend to be low, volatile, and derived from short-term participation (and perhaps speak to why they are no longer active community members). Such observations are meaningless, however, without a greater understanding of reputation accuracy and the types of behavior being captured.

4.2 Assessing Accuracy

Setup: Intuitively, if the WikiTrust computation is accurate then reputation increments will follow from constructive contributions and vice versa. However, establishing ground truth is challenging. Reputation is accumulated when content survives, but rarely is good content celebrated. This motivates our decision to focus on reputation loss events since poor content often generates explicit evidence in subsequent commit comments and/or bug tracking reports.

Investigation began by randomly selecting 100 reputation loss updates from the top-50^{th} percentile of all loss events (by magnitude). In doing so only *significant* loss events are evaluated, skirting subtle instances and the classification difficulties they may pose. Such events report: "author $A = a_{x.y}$ lost reputation per the edit of author $B = a_{x.z>y}$." As a convenience we require that the two edits be *adjacent* in that $z = y + 1$. In this manner the author progression must be $A \Rightarrow B$, rather than with middle-men, *i.e.,* $A \Rightarrow C \Rightarrow B$, as such indirect assessments can convolute investigation.

Assuming A loses reputation on an adjacent edit by B, we seek to answer, "*would B label A's edit as unconstructive?*" We rely on B's commit and its context in making a manual classification of events as `UNCONSTRUCTIVE`, `CONSTRUCTIVE`, or `UNCLEAR`. Admittedly this is a subjective process, but fortunately evidence is often strongly indicative in nature.

Before proceeding we should note that absent an objective/reliable ground truth it is our intention to produce a proof-of-concept assessment. More rigorous analyses are needed operating on: (1) stronger ground truth, (2) multiple repositories of varied size and maturity, and (3) the full spectrum of reputation events.

Rationale & Results: Classifying reputation loss events takes considerable manual effort and proceeds according to the following criteria (see also Tab. 3):

- *Unconstructive*: Instances where the previous contribution has been at least partially reverted or considerably refined. The previous author must be implicated and the unconstructive code changes explained.
- *Unclear*: Similar to the "unconstructive" case, but reasoning is orthogonal to code quality (*e.g.,* whitespace or documentation changes). Also includes the common "not now" revert rationale used when large changes require more testing (but exhibit no blatant errors).
- *Constructive*: Instances not meeting the other two criteria per a conservative approach. Generally this means the cause of reputation loss was not the fault of the affected author (see Sec. 4.3).

Manually classifying our 100 reputation loss events, we arrive at quantities $|\texttt{CONSTRUCTIVE}| = 51$, $|\texttt{UNCLEAR}| = 19$, and $|\texttt{UNCONSTRUCTIVE}| = 30$. Discounting the ambiguous cases our small sample suggests an accuracy of $51/81 = 63\%$. While not appropriate for production use this is an encouraging result for a straightforward WikiTrust application.

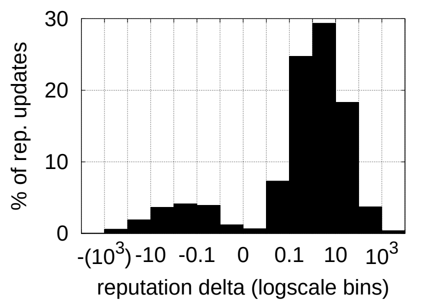

Figure 3: Histogram of reputation update magnitudes.

UNCONSTRUCTIVE
"introduced massive breakage . . . "
"revert x . . . trigger errors"
"revert . . . uglier . . . prone to error"
"revert . . . do not remove functions"

UNCLEAR
"revert x for now . . . needs testing"
"white-space [not per style guide]"

Table 3: Examples of SVN commit comments by class. Sec. 4.3 discusses cases with the "constructive" label (false positives).

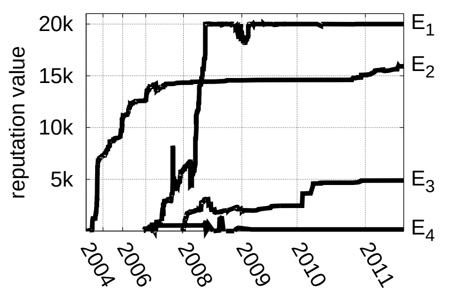

Figure 4: Example editor reputations plotted over repository history.

4.3 Scrutinizing False Positives

Crucial to improving performance is understanding false positives. We identify several redundant sources of error:

- *Reorganization*: When a file is renamed SVN reports a disjoint file deletion and addition. Thus, previous document authors see all of their content deleted (a reputation loss) and the "new" document has no provenance data. Code reorganization has similar effects.
- *The "not now" trap*: Frequently a change is reverted with a "not now" justification, *e.g.,* needing to hold for more testing. When that testing is done the changes are likely to be re-committed in much the same form, punishing the benign reverting editor.
- *Non-code issues*: Authors occasionally place `TODO` style comments in code and then lose reputation when those notations are replaced in favor of actual code. String and whitespace changes sometimes have similar effect.
- *Code upgrades*: Even good code may get refined if methods deprecate, more efficient techniques are discovered, or feature requests are fulfilled.

Algorithmic refinements towards mitigating common errors are discussed in Sec. 5, with about two-thirds of false positives fitting into the above categories.

While these errors are detrimental to performance their existence is valuable in demonstrating the non-uniformity of the collaborative process. Code and natural language evolve in different ways. This finding motivates investigation into this parity across a broader set of cooperative settings and should serve as notice to researchers that cooperative behaviors are not fixed across the "peer production" domain.

5. REFINEMENT & CONCLUSIONS

Future work needs to address weaknesses in both the algorithm and evaluation. WikiTrust excels at modeling intra-document evolution, but reorganization false positives result from inter-document transfers. A solution is to `diff` all files of an SVN check-in. If an entire file is delete-added a rename action has occurred, which wikis can accommodate. Similarly, movement of code blocks between files can be *detected* in this fashion. Modeling proves to be a bigger challenge, but at minimum, undue reputation gain/loss can be prevented.

A second refinement is document/code standardization. Should comments/strings be part of reputation? Should whitespace be normalized for tokenization? What delimiters are most appropriate (line breaks, semicolons, *etc.*)? Language specific pre-processors may be needed.

A more rigorous evaluation is crucial in moving forward. WikiTrust has many free variables. A large and objective corpus of ground truth would provide something to optimize these parameters against and be a test bed for other modifications. Such empirical grounding has proven elusive as one needs to establish not just *where* faults occurred but *when* they originated and *whom* is to blame. It may be necessary to rely on qualitative data from repository participants.

Herein we have shown that a straightforward WikiTrust application is a good foundation towards achieving our goals. A repository-to-wiki mapping was defined and utilized over a production SVN repository to which WikiTrust was applied. Informal analysis revealed mediocre results but also identified common and correctable error cases. Equally as important, these errors are indicative of differences in content evolution and the collaborative process in the differing environments. Accommodating these differences we are optimistic our suggested refinements will permit the design a content-driven reputation model optimized for use in code repositories. Such an approach would allow repositories to have security functionality while still embracing the benefits of collaborative development.

Acknowledgements: This work was supported in part by ONR MURI N00014-07-1-0907 and DARPA HR0011-11-C-0096. We thank our UPenn colleagues Jian Chang, Andreas Haeberlen, Zach Ives, Oleg Sokolsky, and Krishna Venkatasubramanian.

References

[1] Mediawiki. `http://www.mediawiki.org`.

[2] WikiTrust (online). `http://wikitrust.soe.ucsc.edu/`.

[3] B. Adler and L. de Alfaro. A content-driven reputation system for the Wikipedia. In *WWW*, 2007.

[4] M. Cataldo. Sources of error in distributed development projects: Implications for collaborative tools. In *CSCW '10*.

[5] C. Dillow. DARPA's Vehicleforge.mil aims to crowd-source next-gen combat vehicles. *Popular Science (online)*, 2011.

[6] T. L. Graves, A. F. Karr, et al. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[7] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *ESEC/FSE*, pages 296–305, 2005.

[8] R. Priedhorsky, J. Chen, et al. Creating, destroying, and restoring value in Wikipedia. In *GROUP*, 2007.

[9] I. Stamelos, L. Angelis, et al. Code quality analysis in open source software development. *Info. Sys. Journal*, 12, 2002.

[10] H. Zeng, M. Alhossaini, et al. Computing trust from revision history. In *Intl. Conf. on Privacy, Security, and Trust*, 2006.