Quantifying the Development Value of Code Contributions

Hezheng Yin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-174
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-174.html

December 14, 2018
Copyright © 2018, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This is a joint work with my colleagues Jinglei Ren, Qingda Hu, Alex Stennet, Wojciech Koszek and my advisor Armando Fox. I'd like to thank Jinglei for his invaluable contribution in overall algorithm and experiment design, and coining the term DevRank; Qingda for training the commit message classifier; Alex for providing Javascript support for DevRank; Wojciech for insightful feedback and many fruitful discussions; Armando for his support and guidance both in research and life throughout my graduate career.
Quantifying the Development Value of Code Contributions in Large Software Projects

by Hezheng Yin

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Armando Fox
Research Advisor

12-10-18
(Date)

********

Professor Koushik Sen
Second Reader

12/13/2018
(Date)
Counting the amount of source code that a developer contributes to a project does not reflect the value of the code contributions. Quantifying the value of code contributions, instead of only the amount, makes a useful tool for instructors grading students in massive online courses, managers reviewing employees’ performance, developers collaborating in open source projects, and researchers measuring development activities. In this paper, we define the concept of development value and design a framework to quantify such value of code contributions. The framework consists of structural analysis and non-structural analysis. In structural analysis, we parse the code structure and construct a new PageRank-type algorithm; for non-structural analysis, we classify the impact of code changes, and take advantage of the natural-language artifacts in repositories to train machine learning models to automate the process. Our empirical study in a software engineering course with 10 group projects, a survey of 35 open source developers with 772 responses, and massive analysis of 250k commit messages demonstrate the effectiveness of our solution.
Quantifying the Development Value of Code Contributions

Hezheng Yin
UC Berkeley
hezheng.yin@berkeley.edu

ABSTRACT
Counting the amount of source code that a developer contributes to a project does not reflect the value of the code contributions. Quantifying the value of code contributions, instead of only the amount, makes a useful tool for instructors grading students in massive online courses, managers reviewing employees’ performance, developers collaborating in open source projects, and researchers measuring development activities. In this paper, we define the concept of development value and design a framework to quantify such value of code contributions. The framework consists of structural analysis and non-structural analysis. In structural analysis, we parse the code structure and construct a new PageRank-type algorithm; for non-structural analysis, we classify the impact of code changes, and take advantage of the natural-language artifacts in repositories to train machine learning models to automate the process. Our empirical study in a software engineering course with 10 group projects, a survey of 35 open source developers with 772 responses, and massive analysis of 250k commit messages demonstrate the effectiveness of our solution.

1 INTRODUCTION
Developers contribute code to software project repositories. Those code contributions are currently characterized by simple statistical metrics, such as the number of commits (NOC) or lines of code (LOC). For example, GitHub reports NOC as a measure of developers’ contributions in a project [22]. Expertise Browser [50], a classic tool for identifying developers who have required expertise, uses the number of changed LOCs as an indicator of developers’ expertise.

Such metrics measure the amount of code contributed, rather than its value. For example, a function at the core of the application logic is more valuable than an auxiliary script. In the above examples and many other scenarios, developers who make more valuable contributions should be ranked higher or regarded as having stronger expertise.

Many measurements of value are possible; for example, traditional value-based software engineering [8, 10, 17, 49] prioritize resource allocation and scheduling to maximize business value. However, business value may not be the most relevant metric in many real world settings. One example is in software engineering education, instructors need a tool to evaluate individual students’ code contributions to group projects (besides non-code contributions). Another example is free and open-source (FOSS) software projects, where the concept of business value is not applicable but the value of code contributions may influence collaboration and credits of contributors.

Our focus is therefore to quantify the value of code contributions in internal software development activities, that is, the impact on other developers of contributed code. For example, code that addresses a time consuming development task has higher impact than code that addresses an easier task; code that saves other developers effort has higher impact than code that doesn’t. We define development value as a quantification of the development effort embodied in a code contribution and the development effort that the code contribution saves other developers.

We factor the development value into structural and non-structural components. The structural value reflects the impact of the code structure on development activities: A function that is called by many callers “reduces the workload” on those callers and thus tends to be of high value. Based on this observation, we propose structural analysis that derives development value from the code structure. In particular, we analyze the function call graph using DevRank, an algorithm we designed that is a variant of PageRank, to calculate structural development value.

On the other hand, not all development value is reflected in code structure. For example, simply abstracting a common function and enabling it to be reused may be not as valuable as coding the complex logic of the function itself. Through an extensive developer survey, we find that developers judge the value by classifying the impact of commits. We manually examine developers’ responses and develop an impact coding scheme to characterize the non-structural value. We also explore the possibility to automate the commit classification process. We take advantage of the commit messages that usually describe what impact the code makes, and apply natural language processing (NLP) and machine learning (ML) techniques.

Structural and non-structural analysis results are combined to generate a score of development value for the code contribution. We train a learning-to-rank model to find the best combination of the structural and non-structural value in evaluating code contributions, which attempts to match pairwise value comparisons sampled by human developers.

It is worth pointing out that there are valuable non-code contributions in software projects, such as triaging issues, writing documentation, and etc. As a first step towards quantifying contributions in software projects, we’ll focus on code contributions and reserve non-code contributions for future work.

Research questions. To empirically understand the defined concepts and show performance of the proposed algorithms in practice, we carry out a series of developer surveys and analyze open source repositories and databases. Our empirical study addresses two high-level research questions.

RQ1. How do developers evaluate the impact of code contributions when the influence of social factors and personal interests is removed? In an initial survey of software engineering students, we observed that peer reviews and mutual assessment of contributions are subjective and influenced by social and personal factors, leading to
18.36% bias in the code evaluations. Thus, we surveyed developers who compare their own commits in FreeBSD and Linux and found that their judgments naturally center on classification.

RQ2. Can quantitative analyses of code contributions do as well or better than developers’ evaluations? We find that our proposed methods of structural and non-structural analysis both outperform simple metrics such as LOC-counting in matching developers’ judgments of the value of code contributions, and that a combination of structural and non-structural analysis outperforms either method individually. Specifically, the combination of DevRank and the impact coding scheme outperforms the simple statistical metric by 37.4% in terms of prediction accuracy. The error rate of our automated solution is 26.9%, comparable to the human error rate 23.9% in the peer review.

Contributions. In summary, this paper makes the following contributions:

- We establish the importance of quantifying the value of code contributions, and define the concept of development value. We make a case study in a software engineering course, and show the necessity of an algorithmic code evaluation method.
- We performed a survey of 35 open source developers making 772 commit value comparisons, and make observations on how human developers evaluate code contributions. Besides, we contribute an impact coding scheme for non-structural analysis, based on manual grounded-theory analysis.
- We present a new PageRank-inspired algorithm for structural analysis, DevRank. We demonstrate the effectiveness of DevRank through the open source developers survey.
- We propose using a learning-to-rank algorithm to combine structural analysis and non-structural analysis. The combination achieves the best performance in predicting commit comparison results in the open source developers survey.
- We apply NLP and ML techniques to software project artifacts for non-structural analysis. We are among the first to perform a massive exploration of NLP/ML models for context learning (with 250k+ commit messages from Apache projects).

The rest of the paper motivates our work in Section 2, and describes the algorithms for structural and non-structural analysis in Section 3. The empirical study to evaluate them is described in Section 4. We discuss the automated commit classification in Section 5, future work in Section 6 and related work in Section 7.

2 MOTIVATION

Quantification of development value has strong potential applications in various areas in which individuals’ contributions are typically evaluated using imprecise or subjective measurements.

Education. Team-based projects are widely used in software engineering courses [54], but it can be challenging to fairly assess individual students’ contributions. Simply counting NOC or LOCs hardly reflects the actual value of students’ contributions and is easily gamed. Unfair evaluation frustrates high performers and allows low performers to “free-ride” on the efforts of colleagues. Currently, instructors typically rely on student peer review or manual check to do the evaluation. This approach is subjective and not scalable with the course size, especially for increasingly popular massive online courses (MOOCs).

Software company. Similarly, evaluating the contributions of different developers who specialize in different parts of a project can be challenging even for managers who have a deep technical understanding of the project (and not all do). As a result, developers’ performance may be judged by reporting or personal communication, which is subjective and is well-known to be prone to bias based on personality and social relationships [33]. Unfortunately, this may lead to negative outcomes. According to a recent survey of 2,000 employees [51], two widely-reported reasons employees voluntarily left their jobs were “lack of recognition or reward” (45.24%) and “boss didn’t honor commitments” (43.49%).

Distributed open-source development. Large open source software (OSS) projects run in a geo-distributed manner, which largely limits in-person social contact among developers. Instead, developers rely on computer-mediated and usually publicly-available information to facilitate development [56, 67]. For example, visibility of development history affects developers’ receptiveness to others’ code contributions [42, 59, 63]. Our work can provide developers with a more in-depth view of history contributions.

Although open-source projects are not financially motivated, many receive donations, raising the fundamental problem [53] of how to allocate them among developers. Indeed, the leading author of the popular framework Vue.js recently raised exactly this issue. Our work can pave the path to fair assignment of monetary rewards, and thus has the potential to evolve the current rewarding and financial support mechanisms for open source projects.

Even in open-source projects in which allocating reward is not a problem, the contribution profile is useful for project organization and management. Some open source projects practice role promotion [28, 61], a voting scheme [18] or some implicit form of coordination and conflict resolution [37]. Being able to quantify the value of different contributors’ code would provide a solid foundation for those processes. Such information can even be used for job advertisements [24].

Research. Beyond the above concrete applications, our work makes a new quantitative tool to observe development activities. Prior research, which is limited by lack of such a tool, can be extended to answer many interesting questions: What is the relationship between the value of developers’ code contributions and the system or community structure [4, 15, 37]? How developers’ code contributions influence conflict resolution, leadership and organization [18, 57, 62]? To what extent can development value quantification influence or help project management and maintenance [5, 7, 47]?

In this work, we aim to advance the understanding of the value of code contributions in development activities. To that end, we parse the development value from two perspectives, structural (§3.1) and non-structural (§3.2). They are complementary factors that constitute the concept of development value.

3 COMPUTING DEVELOPMENT VALUE

We postulate that a code contribution carries two kinds of value. The structural value reflects its role in the structure of the program, and can be derived by considering both the effort that went into creating the code and the effort the code saves other developers.

1https://changelog.com/rfc/12
Section 3.1 presents our algorithm (based on Google PageRank) for computing structural value. A contribution’s nonstructural value reflects the contribution’s impact to the project in a way that code structure alone cannot: for example, a critical bug fix, while simple, might be as important as a complex new feature. Section 3.2 describes how we manually code developers’ judgments about the relative importance of different commits as a way of capturing nonstructural value; in Section 5.2, we discuss how the manual coding used in this paper could be automated.

The overall value of a contribution is obtained by combining its structural and nonstructural value. Section 3.3 describes how we use a Learning-to-Rank (L2R) model to do so. In Section 4, we show how well this computation works on a set of open-source projects.

### 3.1 Structural Value: DevRank

In most imperative programming languages, a function (procedure, method) is a basic unit of program structure. Functions both provide a way to divide a complex task into subtasks and enable reuse of code that is called from multiple sites. Therefore, the development value of a function is based not only on the effort spent creating the function, but also the development effort saved when other parts of the code call the function. Our graph-based algorithm for ranking development value, DevRank, is an extension of the original PageRank algorithm.

#### 3.1.1 Background

PageRank [11] is the basis for Google Web Search, and finds applications in various domains [23]. The algorithm runs over a directed graph of web pages. It hypothesizes a web surfer with assumed visiting behavior, and iteratively calculates the probability that the surfer visits each page. The meaning of the calculated probabilities depend on the behavior of the surfer. In the original PageRank, the surfer does two random actions: (1) upon arriving at a page, the surfer randomly selects a link on that page to follow; and visits the linked page; (2) at random times, instead of following a link on the current page the surfer stops teleports to a random page and continues. Based on the behavior, the resulting probability reflects how likely a page is visited according to the link structure of pages. Intuitively, what is reflected is the popularity or importance of a page on the web.

More precisely, let $P$ denote a stochastic matrix, where $P_{i,j}$ is the probability of visiting page $i$ from page $j$. The surfer follows a link with probability $\alpha$ and teleports to another page with probability $(1 - \alpha)$. The stochastic column vector $p$ stores probabilities of the surfer teleporting to all pages, respectively. We also need $e$, an auxiliary row vector of all ones. Then PageRank is a Markov chain with the stationary distribution $v$, a column vector that satisfies

$$ (\alpha P + (1 - \alpha)e)v = v. $$

To calculate $v$, we perform the iteration

$$ v^{(k+1)} = \alpha P v^{(k)} + (1 - \alpha)e, $$

where $k \in \mathbb{N}$ and $v^{(0)} = p$.

#### 3.1.2 DevRank

To compute each function’s development value, we analyze the code repository’s function call-commit graph that contains function calls and history commits. Of course, during program execution, control flow never randomly jumps to an irrelevant function as in PageRank. However, we find that the PageRank-over-call-graph model is a surprisingly convenient tool to characterize program development. To do this, we make two important changes to PageRank. First, we interpret random teleportation as navigating the development activity of the code, rather than runtime activity. Second, we consider not only the instantaneous state of the function call graph, but its development history as revealed by a set of revisions in a revision-control system over time.

In DevRank, PageRank’s hypothetical “surfer” becomes a DevRank development sniffer, whose task is to detect development effort. To construct the behavior of the sniffer, we assume that the number of LOCs of the function indicates the development effort spent on the function in general (this definition can be extended as discussed later). Based on this assumption, the sniffer performs one of two random actions. (1) Upon arriving at a function, it visits one of the called functions with probabilities proportional to the sizes of those functions. That means the more development effort associated with a called function, the more likely it is visited. As we regard calling a function as a way to save development effort on the caller, this behavior reflects how much development effort is saved by coding a call to each function. (2) At random times, the sniffer teleports to any function with a probability proportional to the size of the function. Such teleportation can be explained as the sniffer’s search for development effort. Overall, we can see that the resulting probability of the sniffer showing up on each function reflects the development effort spent on the function and that function’s potential to save other developers’ effort. Therefore, it reflects the development value of a function.

The call-commit graph is formally defined as $G = (V, S, E, W)$. Each vertex in $V$ represents a function’s code, and a directed edge in $E$ from $v_1$ to $v_2$ means that $v_1$ calls $v_2$ (dynamic binding is approximated by establishing an edge from the abstract function to every possible binding); $S$ is vertex the set of commits; $W$ are directed edges across functions and commits, representing changes that commits made to functions. An edge $(s, v)$ in $W$ means that commit $s$ modifies function $v$. We associate a weight $w$ to each edge in $W$ to record the number of modified code lines. Both code additions and reductions are counted in the weight, as they are all valuable development efforts.

The mathematical nature of DevRank is identical to PageRank. We only need to reset $P$ and $p$. We use $|f|$ for the size of a function. Suppose function $j$ calls a set of functions, $F_j$. According to the sniffer’s first behavior, we set

$$ P_{i,j} = \frac{|f_i|}{\sum_{f_j \in F_j} |f_j|} \quad \text{if } f_i \in F_j, $$

$$ 0 \quad \text{otherwise}. $$

Furthermore, the universal set of functions in the call graph is $U$. According to the second behavior, we have

$$ p_i = \frac{|f_i|}{\sum_{f \in U} |f|}. $$

The development effort associated with a function is not necessarily entirely revealed by examining the call graph at an instant of time. Since we have the full history of software revision in a repository, we can more precisely quantify the development effort associated with code. The development effort for a function is not
only the latest code in the function, but also a history of updates that result in the code. In Git, a commit is a group of changes made atomically to a subset of files in the repository that takes the repository between two observable states. Without loss of generality we will use the term commit in this way; other source/configuration management systems have a similar mechanism even if they use a different term. We define the total number of code lines of all changes to a function denotes the development effort spent on the function. For example, a developer replaces $n_1$ code lines of a function with $n_2$ new code lines. Then we count $(n_1 + n_2)$ as the development effort contributed to the function.

Therefore, we run the above definition of DevRank on a view of the call-commit graph, $G' = (V, E)$, but define the function size $|f|$ as below:

$$|f| = \sum_{(s, f) \in W} w(s, f).$$

Finally, the call-commit graph offers a way to distribute development value of functions to commits, and further to developers. DevRank computes a number for each function, which represents the development value. Our default policy is to allocate the value of a function to all commits that change the function, proportionally to their weights. After every commit has collected the value from all functions it modifies, we assign the value of commits to their corresponding authors. In this way, developers receive credits for their contributions to the development value.

### 3.2 Non-Structural Value: Impact Coding

Not all development value is embodied in the code structure. A code contribution also has a non-structural impact on the whole project, e.g., fixing a bug, making an improvement, creating a new feature, or maintaining a document. Our surveys of open source developers confirm the importance of such impact in assessing the overall value of contributions. Therefore, we introduce another method, impact coding, to capture such non-structural value.

Different types of impacts that commits bring to a project can be manually labelled by developers in a similar way to the use of JIRA, where each issue is associated with a type and a priority.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text adjustment</td>
<td>Cleanup</td>
<td>Trivial textual or stylistic changes</td>
</tr>
<tr>
<td>Code style adjustment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document update</td>
<td>Documentation</td>
<td>Descriptions of the code including comments</td>
</tr>
<tr>
<td>Document addition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>Maintenance</td>
<td>Structural changes for better software engineering</td>
</tr>
<tr>
<td>Code structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logic bug</td>
<td>Fix</td>
<td>Correction of mistakes</td>
</tr>
<tr>
<td>Design bug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better logic</td>
<td>Improvement</td>
<td>Changes for better quality or performance</td>
</tr>
<tr>
<td>Better design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New use</td>
<td>Feature</td>
<td>New internal or external functionality</td>
</tr>
<tr>
<td>New design</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The impact coding scheme, developed from developer surveys by the grounded theory methodology, for non-structural analysis.

In this work, we define a coding scheme for commit classification in terms of non-structural value. Previous work [39] defines related categories of development activities and is reflected in standards [27], but they are not proven to be suitable for representing non-structural value. Instead of providing any predefined categories or even assuming developers ever do classification at all, we let developers freely express their reasons for value comparisons, and analyze their responses offline. Our current survey is sent out to top FreeBSD and Linux developers. In total, we manually review 848 commit descriptions written by 26 developers. We follow the grounded theory approach to derive the scheme. The process is in several stages: we first gather concepts from developer’s answers and then group them into categories. In that sense, concepts can be regarded as subcategories. We actually went through several iterations to make concepts and categories coherent, after merging and splitting some of them. Table 1 lists our result.

In the survey, we see that developers use vague terms in their natural-language descriptions. For example, the word “cleanup” may refer to a superficial text tweak as in Figure 1(a), but may also refer to a code structure change as in Figure 1(b), which brings a different and larger impact than the text tweak. Besides, some terms in developers’ responses are inconsistent. For example, the commit message of Figure 1(c) sounds like a bug fix, but it is in fact a performance improvement, according to the developer’s explanation of its value. We looked into this commit and it solves a performance degregation due to packet loss. Although in a broad sense it can be regarded as a “performance bug”, the commit does not impact the correctness of the program. To avoid any confusion, we classify it as an improvement.

Therefore, it is a necessity and a critical job for us to unify the definitions of the terms in our impact coding scheme.

- **Cleanup** is source code hygiene that is not expected to impact the compilation or interpretation output. We observe two typical activities in this category, text adjustment and code style adjustment. The former is to refine the text in the source code, such as fixing a typo or using a more readable variable name. The latter is formatting the code to follow the proper code style. Figure 1(a) shows an example of small stylistic cleanup, which only increases a single-line indentation.

- **Documentation** refers to a change that impacts documents as well as comments. We distinguish two types of documentation activities. One is to draft a new content, and the other is to revise the existing content.

- **Maintenance** is a general term, but here it only means enhancement on the software engineering practice that does not impact the program behavior. Figure 1(b) shows an example of maintenance that removes an unnecessary parameter. A more extensive refactoring can also fall into this category. Another example is putting a constant modifier before a data structure to avoid any careless modification. We refer to the type of the former example as a code structure maintenance, and the type of the latter example a data structure maintenance.

- **A fix deals with a bug** that incurs a wrong logic or design of the program. Here the logic refers to a relative small local flow. A logic bug can be, for example, a null pointer deference or a

Hezheng Yin

---

https://git-scm.com
Quantifying the Development Value of Code Contributions

memory leak. Meanwhile, a fix on the program design impacts multiple collaborating components of the program (e.g., a concurrent bug).

- An improvement makes the program of higher quality in terms of security, reliability, performant, or other metrics. The concepts of logic and design are the same to those under the fix category. Figure 1(c) shows an example of a logic improvement, which relocates one line but does not involve high-level program behavior redesign.

- A feature impacts the program by adding internal or external functionality to it. A feature typically changes the design instead of tweaking local logic, so there is not a "logic" feature concept (the survey result confirms this). But a feature can be internal to the program without visibility to end users (e.g., creating a utility function). The usage concept of this category covers the cases where the feature is not realized by a new design but by a new way to using existing building blocks. Figure 1(d) is such a case. That commit adds a new device node, in a similar way to the existing code (note the surrounding code).

3.3 Combining DevRank and Impact Coding Using L2R

As structural and non-structural analyses capture two fundamental aspects of development value, we combine the two to calculate overall development value. Suppose a commit has structural value $d$ and non-structural value $t$. Our goal is to find a function that combines them: $v = \phi(d, t)$. In our solution, $d$ is the DevRank score, and $t$ is a one-hot vector encoding the commit category, given by the impact coding scheme or the context learning algorithm.

If we had reliable ground truth—that is, a large set of commits with the known overall development value of each commit—we could pose the task as an optimization problem: from the data set, determine the weight vector $w$ in:

$$\phi(d, t) = w^T \begin{bmatrix} d \\ t \end{bmatrix},$$

so that the average error between the true value and $\phi(d, t)$ of every commit is minimized.

Unfortunately, developers find it very hard to directly score code values, e.g., giving one commit 0.17 and another 0.06, so we lack reliable ground truth for the values. Instead, we ask developers to compare many pairs of commits and identify which in each pair is more valuable. Based on this "pairwise ground truth," we use a learning to rank (L2R) algorithm to determine $\phi$. L2R is a supervised learning model originally proposed to rank documents for information retrieval. Our task can be formulated as a pairwise case of the L2R framework. There are several established training algorithms for training the model, including Ranking SVM [25], IR SVM [12] and RankBoost [19]. To control for the model expressiveness and focus on the performance of different feature combinations, we choose Ranking SVM (RankSVM) as the learning model in this paper.

https://github.com/torvalds/linux/commit/1dbb670

https://github.com/torvalds/linux/commit/11afbde

Figure 1: Examples commits and developer descriptions of some impact types.

For pairwise L2R, given a pair of commits, the target is to figure out which commit has more development value, i.e., a binary classification. A data set of $m$ instances is provided as $\{(x_i^{(1)}, x_i^{(2)}, y_i), i = 1, 2, \ldots, m\}$, where $y_i$ is the class label of instance $i$. In this case, it is a binary classification task with $y_i = 1$ (commit $x_i^{(1)}$ is more valuable) or $y_i = 0$ (commit $x_i^{(2)}$ is more valuable). The label $y_i$ is determined by the human annotation.
1, 2, ..., m, where each instance consists of two commits to compare, \(c_i^{(1)}, c_i^{(2)}\), and a label, \(y_i \in \{+1, -1\}\), denoting which commit is more valuable. We collect labels \(y_i\), \(i = 1, 2, ..., m\) by surveying experienced developers in successful open source projects. Then we train a Ranking SVM to match human developers’ judgments.

To be precise, the training workflow is as follows. Given \(k\) features that characterize a commit, the two commits of each pair \(c_i^{(1)}, c_i^{(2)}\) are characterized by two feature vectors, \(x_i^{(1)} = [x_{i1}^{(1)}, x_{i2}^{(1)}, ..., x_{ik}^{(1)}]\) and \(x_i^{(2)} = [x_{i1}^{(2)}, x_{i2}^{(2)}, ..., x_{ik}^{(2)}]\), respectively. Then the difference between the two feature vectors, \(x_i^{(1)} - x_i^{(2)}\), is fed into the learning model, together with the corresponding label \(y_i\).

The trained SVM can not only be used to rank the commits, but also results in the weight vector \(w\) to be used in \(\varphi\). To get \(w\), we use the DevRank score \(d\) and the one-hot encoded commit category \(t\) as the input features to RankSVM. After training, we take the weight vector of the SVM as \(w\) in \(\varphi(d, t)\), allowing us to combine the structural and nonstructural value scores for each commit to determine its overall development value score. Moreover, to connect commits to developers, we can modify the final step of DevRank: instead of “crediting” the commit’s author with the structural value of the commit, we credit the author with the overall value \(\varphi(d, t)\) of the commit. After the adjustment, outputs of DevRank are proportionally unified so that all scores still sum up to 1.

4 EMPIRICAL FINDINGS

The way a human developer assesses the value of code should be the basis of an algorithmic solution. RQ1 asks how developers evaluate the impact of code contributions when the influence of social factors and personal interests is removed. We hypothesize that they do so by considering what they believe to be the impact of a commit on the overall codebase or on other developers:

HYPOTHESIS 1. Human developers judge the value of commits mainly by classifying impacts of the commits.

As we show, however, developers’ subjective evaluations of such impact are subject to measurement bias, so we ask developers to compare and rank randomly-selected pairs of their own commits to mitigate this effect and provide a data set for evaluating H1.

We then turn to RQ2—whether quantitative analyses of code contributions can do as well or better than developers’ evaluations. As a baseline, we compare our algorithms to LOC counting, as the number of changed lines of code, including both additions and deletions, is a widely used measurement of code contributions [22, 31, 50]. We evaluate our structural analysis, non-structural analysis, and combination of the two, as follows:

HYPOTHESIS 2. DevRank, our algorithm for structural analysis, outperforms LOC counting in predicting developers’ comparisons of commits.

HYPOTHESIS 3. Impact coding, our algorithm for non-structural analysis, outperforms the LOC counting method in predicting developers’ comparisons of commits.

HYPOTHESIS 4. The combination of DevRank and the impact coding scheme outperforms either analysis method alone in predicting developers’ comparisons of commits, and shows an error rate comparable to that of human assessment.

4.1 Developers Judge Their Own Commits by Impact Classification

Developers may assess code value through their understanding of the code, and their experience with and impressions of other developers. Typically, human-generated results are regarded as the ground truth in an empirical study, as when human labels on images provide ground truth in the ImageNet data set for visual object recognition [16]. However, judging the development value of code contributions is not as simple as identifying a cat in an image. Moreover, humans may be biased by social factors and personal interests. At the same time, any algorithmic solution should be consistent with how a “reasonable” developer would assess the value of code contributions.

To better understand the limitations and the validity of human assessment, we surveyed 10 teams of students (58 students total) in an undergraduate software engineering course of a major research university. All team projects involved working on "real" applications with partner organizations. The 8-week-long project is divided into four agile development iterations, each 2 weeks long. After each iteration, individual students complete the following survey evaluating their teammates’ contributions during that iteration:

"Suppose your team forms a company. Assign team members (including yourself) shares in the company, normalized to 100% total, based on their code contributions for this iteration, where the contribution captures your view of both the quantity and impact of code contributed.”

Figure 2 shows statistics of each student’s self-assigned share and shares received from teammates. We calculate the mean and standard deviation of the received shares. All shares that a student
assigns to the whole team are normalized to 100%. We see that team-
mates give very different amounts of shares to the same student, 
showing the subjectivity in human value assessment. The deviation 
of code shares received by a student ranges from 0.1% to 36.95%.

We also examined the average correlation between students’ rat-
ings. For each possible pair of students in a team, we computed the 
Pearson’s r coefficient between their ratings. By averaging across 
teams and across teams, we found the overall average of Pearson’s 
r to be 0.54/0.52 for general/code contribution respectively, only 
indicating moderate level of agreement among students.

Moreover, students’ self-assigned shares are 18.36% more than 
their peer-assigned shares, suggesting that their self-assessment is 
subjectively more optimistic than their peer assessment. TA com-

munications with students (which we cannot reveal due to privacy 
considerations) also suggested gaming/collusion among team mem-
ers as well as personal biases.

Methodologically, therefore, in our larger survey we ask develop-
ers to compare pairs of their own commits, to mitigate this bias. We 
tested Hypothesis 1 by examining the methodologies developers 
used for comparing commits. We developed a survey system and 
invited FreeBSD and Linux developers to compare random pairs of 
their own commits and explain the reasons for their choices. The 
commits are randomly selected from among the 100K FreeBSD 
commits since commit e0562690 and the 160K Linux commits 

To test Hypothesis 2, we compared DevRank and LOC’s perfor-
mance in predicting developers’ comparisons of commits. The com-
mit that has higher DevRank or LOC value is predicted as more 
valuable. As Table 3 shows, contrary to the traditional intuition that 
LOC correlates well with the amount of contribution, LOC-counting 
only achieved an accuracy of 53.2%, slightly higher than by purely 
characteristic of Hypothesis 2.

To test Hypothesis 3, we manually labeled all commits in the sur-
vey with categories according to the impact coding scheme. Then 
we one-hot encoded the commit categories into feature vectors and 
fit a ranking SVM to our data. We used 10-fold cross validation 
and the average accuracy of the ranking SVM was 69.1%, signifi-
cantly higher than the 53.2% of LOC counting method, indicating 
support for H3.

4.2 DevRank and Impact Coding Outperform 
LOC Counting for Predicting Developers’ 
Commit Comparisons

To test Hypothesis 2, we compared DevRank and LOC’s perfor-
mance in predicting developers’ comparisons of commits. The com-
mit that has higher DevRank or LOC value is predicted as more 
valuable. As Table 3 shows, contrary to the traditional intuition that 
LOC correlates well with the amount of contribution, LOC-counting 
only achieved an accuracy of 53.2%, slightly higher than by purely 
characteristic of Hypothesis 2.

To test Hypothesis 3, we manually labeled all commits in the sur-
vey with categories according to the impact coding scheme. Then 
we one-hot encoded the commit categories into feature vectors and 
fit a ranking SVM to our data. We used 10-fold cross validation 
and the average accuracy of the ranking SVM was 69.1%, signifi-
cantly higher than the 53.2% of LOC counting method, indicating 
support for H3.

Table 2: Statistics of the classification methodology adopted by developers in commit comparisons.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Sample explanations</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit</td>
<td>[&quot;add a new feature to support flame-graph more efficiently&quot;] is more valuable than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;improve user experience&quot;]</td>
<td>83.3%</td>
</tr>
<tr>
<td></td>
<td>[&quot;performance boost&quot;] is more valuable than [&quot;fix a bug which is not super important&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;cleans up code&quot;] is more valuable than [&quot;is trivial textual fix&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;fixes a bug&quot;] is more valuable than [&quot;is a fairly minor performance &amp; code clarity optimisation&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;a fix for memory leak&quot;] is more valuable than [&quot;a fix for documentation&quot;]</td>
<td></td>
</tr>
<tr>
<td>Implicit</td>
<td>[&quot;improve user experience&quot;] is more valuable than [&quot;just a trivial change to export necessary functions for future use&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;protect structure contents&quot;] is more valuable than [&quot;optimizes non critical path&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;just shrinks kernel size&quot;] is more valuable than [&quot;just a cosmetic spelling fix&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;spec compliance&quot;] is more valuable than [&quot;tool for static code checking&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;Make sure that RESET name for dts is accurate&quot;] is more valuable than [&quot;cleanup&quot;]</td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>[&quot;commit body&quot;] is more valuable than [&quot;no commit body&quot;]</td>
<td>16.7%</td>
</tr>
<tr>
<td></td>
<td>[&quot;this null dereference doesn’t affect normal users&quot;] is more valuable than [&quot;tiny impact, doesn’t affect normal users at all&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;networking stuff is important&quot;] is more valuable than [&quot;code cleanup&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;is required for newer &gt;= 16 MIPS CPUs to work&quot;] is more valuable than [&quot;is trivial contact information&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[&quot;show aggregated stat and task info when CPU is idle&quot;] is more valuable than [&quot;show raw value of fields in a trace record&quot;]</td>
<td></td>
</tr>
</tbody>
</table>

We manually examined developers’ explanations for commit 
comparisons, and labeled whether each explanation implies a form 
of classification, either explicit or implicit. The labels, example 
explations and their corresponding proportions in all explana-
tions are shown in Table 2. We find support for H1 as the “through 
classification of commits” is the most frequently used method and 
accounts for 83.3% of all explanations. In the table, the “explicit” 
label means that the explanation contains clear key words for clas-
sification, such as “add a new feature...”, and “performance boost”. The “implicit” label means that, although the explanation does not 
directly point to classification, we can easily derive a category from 
it. For example, “protect structure contents” is a maintenance of 
the data structure; “tool for static code checking” should involve a 
new feature. In contrast, explanations without doing classification 
take only 16.7%. They mention various very specific reasons. For 
example, “networking stuff is important” indicates the component 
of the commit; “show raw value of fields in a trace record” is a detail 
in functionality. We believe those specific reasons are hard to gen-
eralize (or at least our current scale of developer surveys, though it 
is already extensive, still do not support such generalization).
Figure 3: Developers’ relative changes between scores calculated by LOC counting and DevRank. y = 1 means both scores for a developer are identical. Legend keys, i.e., 0.4, 0.8, are the values of α in DevRank.

Table 3: Performance of individual analyses and combinations in predicting developers’ commit comparisons.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>-</td>
<td>53.2%</td>
</tr>
<tr>
<td>DevRank</td>
<td>-</td>
<td>59.0%</td>
</tr>
<tr>
<td>Coding</td>
<td>Ranking SVM</td>
<td>69.1%</td>
</tr>
<tr>
<td>Coding + LOC</td>
<td>Ranking SVM</td>
<td>70.2%</td>
</tr>
<tr>
<td>DevRank + Coding</td>
<td>Ranking SVM</td>
<td>73.1%</td>
</tr>
</tbody>
</table>

4.3 The DevRank + Impact Coding Combination Achieves the Best Performance

We tested Hypothesis 4 by comparing the performance of using the combination of DevRank and impact coding with either analysis method alone. We also explore how other combinations of our analysis methods and LOC counting may perform. A ranking SVM is trained in a similar way described in §4.2, except that the feature vectors include both DevRank scores and commit categories this time. 10-fold cross validation was employed and the average accuracy was 73.1%. This is our highest accuracy among all analysis methods and feature combinations.

We assess human error rate by asking two senior CS graduate students without Linux kernel development experience to compare 100 pairs of commits randomly sampled from our Linux survey data set. Their answers were graded against the ground truth provided by the authors of those commits and they achieved an accuracy of 69.0% and 84.0% respectively. The gap between their performance reflects both human developer’s ability difference and subjectivity issue. Note that our best model’s performance (73.1%) is in between the two human accuracies, which supports Hypothesis 4 that the combination of DevRank and impact coding shows an error rate comparable to that of human assessment.

In our current empirical study, we regard developers’ commit comparisons as the ground truth, for all structural analysis, non-structural analysis and their combination. The assumption here is that developers take into account both structural value and non-structural value when they compare commits. Our results in Table 3 supports such an assumption in that both structural analysis and non-structural analysis add to the accuracy of predicting developers’ comparisons. However, this commit-oriented approach is more suitable for surveying non-structural value than structural value, because developers only see individual commits and thus tend to lose sight of the structure of related code. We found that quantitatively only 10.6% of all explanations mentioned concepts related to code structure and structural value. This may explain why our results show that non-structural analysis increases the accuracy more than structural analysis does. We believe the effectiveness of structural analysis is underestimated in our study.

To better observe the effects of DevRank, we perform another experiment on three Apache projects to calculate developers’ scores of development value. We select Flink, Kafka and SystemML for this experiment as they are representative of different ranges of the number of project developers: Flink has 283 developers; Kafka has 191 developers; SystemML has 40 developers. Figure 3 depicts how DevRank extensively changes individual developers’ scores, compared to LOC counting. The change rate is from \(-90.6%/-73.3%/-59.7\%\) to \(24.3%/16.7%/6.7%\) for Flink/Kafka/SystemML. DevRank largely changes developers’ scores, compared to LOC counting, due to its inherent capability of structural analysis.

Another phenomenon in Figure 3 is that, as α increases, developers with less contributions tend to see more reduction of scores. That means DevRank shares tend to more intensively gather around
major contributors as \( \alpha \) becomes larger. \( \alpha \) can largely influence the share distribution of DevRank.

5 DISCUSSION

We discuss two factors of the threads to validity in our research, and the potential of applying context learning to our work in practice.

5.1 Threats to Validity

**Non-code contributions.** This paper focuses on the code contribution, though writing code is only one part of the work that leads to a successful project. Other non-code contributions include user interview, requirements definition, planning, coordination, management, and so on. One one hand, those types of contributions are conceptually different from the code contribution, so our results should not be generalized to them. On the other hand, it is possible that the code contribution is an indicator of more general contributions in practice. We experience many cases where the developers who contribute most code naturally take more roles in other aspects of project development. To preliminarily test this, we put another question in the course survey that asks students about their general contributions instead of the code contribution. It turns out that the general-contribution shares students received are close to their code-contribution shares. Both types of shares are highly correlated, with a correlation coefficient of 0.93. One reason for the correlation is that developers of those small projects are not very specialized. Therefore, at least for a small project, the developers who make most code contributions tend to make most general contributions as well.

**Gaming and attacks.** Our empirical study is ex post analysis, so neither gaming behavior of developers nor attacks against our solution are assumed. But in order to generalize our work to actual use, we have to consider them. Developers may deliberately increase the number of LOCs or commits if they know in advance that contributions are judged by such metrics. Meanwhile, there are constraints against adverse behavior in reality. Code commits are typically reviewed by peer developers before being merged to the repository. It is relatively easy for a developer to split one commit to multiple ones or write lengthy code without being noticed in code review. But it costs more thoughts to inflate DevRank, which involves logic or structure changes. Also, simply making one function call into multiple nested ones rarely adds to the DevRank score (a property of PageRank-style algorithm). Overall, our solution is robust to developers’ potential gaming behavior or even attacks.

5.2 Automated Commit Classification

In this study we relied on manual classification of commits into different types, but it is burdensome for developers to label every commit. Therefore, we are investigating automating this task. We expect to at least provide an assistance tool to prevent/monitor human mistakes or gaming behaviors. As the first step in this direction, we take advantage of NLP and ML techniques to classify commits according to commit messages.

A favorable fact about software development is that many communications among developers are computer-mediated [56, 67]. As a result, each code commit is associated with a context developers implicitly build in development activities. Particularly, the context is recorded as a natural-language description of the commit in bug/issue tracking systems or pull requests [14, 64]. They provide an adequate corpus for constructing a machine learning model to infer the impact of a commit. We refer to our approach as context learning.

Since the developer survey and our manually labelled commits are of a limited scale for machine learning, we leverage the JIRA issue database\(^6\) used by many Apache Software Foundation projects. In the database, developers label issues with predefined types (in contrast to, e.g., GitHub Issues, which does not enforce such a constraint), and many projects also follow the convention that the JIRA issue ID (e.g., "SPARK-16742") appears in commit messages for commits addressing that issue, making it possible to link commits to their corresponding issue types (feature, improvement, bug fix, maintenance). We have experimented with training NLP and ML models on 267,446 issues and their associated commit messages from 139 Apache projects (projects with top most issues are selected), but have not yet reached sufficiently high accuracies to rely on this classification in place of manual coding.

We explore three main NLP + ML models, bag-of-words (BoW) [32], a convolutional neural network (CNN) [34], and a recurrent neural network (RNN). For each model, we experiment with two types of inputs, the commit message title and the complete message. We adopt ConceptNet Numberbatch (CN) word embeddings [60]. For all issue types, we show F1 scores in Table 4. We can see that CNN and RNN have comparable performance, but significantly outperform the bag-of-words model. The best average F1 score that our CNN model achieves among all classes is 0.60, using full commit messages; similarly, the best average F1 score of our RNN model is 0.59, using commit titles. In contrast, the best F1 score of the bag-of-words model is only 0.55, using commit titles.

The models show different levels of performance among classes of commits. The best F1 score for bug is 0.87 with precision 0.88 and recall 0.87 (by the RNN model using commit messages), while the best F1 score for maintainance is only 0.46 with precision 0.58 and recall 0.38 (by the CNN model using commit messages). Such different learning results should be attributed to the number of commits of a class in the data set. Bug fixes are dominant and numerous so that training for them is effective. Overall, identification of bug fixes and improvements can be regarded as usable for DevRank, but that of minor classes is too low to be used in practice.

<table>
<thead>
<tr>
<th>Type</th>
<th>Maint. Feature</th>
<th>Improv.</th>
<th>Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td># commits</td>
<td>329</td>
<td>1517</td>
<td>14136</td>
</tr>
<tr>
<td>BoW-title</td>
<td>0.28</td>
<td>0.34</td>
<td>0.63</td>
</tr>
<tr>
<td>BoW-message</td>
<td>0.2</td>
<td>0.33</td>
<td>0.62</td>
</tr>
<tr>
<td>CNN-title</td>
<td>0.37</td>
<td>0.39</td>
<td>0.65</td>
</tr>
<tr>
<td>CNN-message</td>
<td>0.46</td>
<td>0.39</td>
<td>0.64</td>
</tr>
<tr>
<td>RNN-title</td>
<td>0.4</td>
<td>0.35</td>
<td>0.67</td>
</tr>
<tr>
<td>RNN-message</td>
<td>0.33</td>
<td>0.36</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Table 4: Performance (F1 score) of three NLP + ML models for context learning.

\(^6\)https://issues.apache.org/
6 FUTURE WORK

This work can be extended in several ways. We have the following plans to improve the accuracy of development value quantification and further advance our knowledge on this topic.

First, collect and impact-code more commit comparison data as the ground truth for analysis and training. This can be done by a larger-scale survey of developers and crowdsourcing for manual labeling. Such a data set will provide a basis for the following plans.

Second, experiment with more advanced machine learning models. In Section 5.2, we already show that the current context learning model performs best for commit types that have a large number of data points. Meanwhile, we currently use a linear SVM for L2R, but this choice is made mainly due to the limited amount of training data. Both context learning and L2R see room for optimization as long as there is enough data input.

Third, support more programming languages. Our current implementation cannot apply to dynamically-typed languages because our program analysis component is based on SrcML [41] to parse code structures, which supports C/C++ and Java, but not (for example) Ruby. To support a dynamic language like Ruby, we will have to implement static typing [1, 2].

Fourth, explore program analysis and comprehension techniques to do impact assessment. Currently, we rely on either costly human reasoning or context learning over natural-language commit messages. It may help if our solution can directly understand the change that a commit makes. Such understanding may lead to better commit classification.

7 RELATED WORK

PageRank has been applied to many areas [23]. In the software engineering area, one use case is to run PageRank on the function call graph to estimate bug severity and maintenance effort [5]. Besides, graph topology or characteristics (e.g., degree centrality) are studied to capture software properties [6, 55, 65] and predict defects [26, 69]. This technique is also applied to portray developers [20, 30, 45]. Besides, social factors are also exploited for various purposes of analysis [9, 40, 44, 52]. In contrast, our DevRank is a variant of PageRank adapted to reflect the development value.

Effort-aware models [43, 48] show the effects of considering the development effort in software engineering, and different effort estimation schemes [31, 58, 66] have been proposed. They share similarity with our development value assessment, but a key difference is that we additionally count the effort to be saved, instead of merely the effort that is spent. Also, we combine development effort with non-structural impact analysis in development value assessment. Finally, DevRank can be extended to use a more advanced effort estimation scheme (e.g., complexity, churn) to calculate the “size” of a function node (Section 3.1), while the overall framework and core methodologies of our work are orthogonal and remain applicable.

There are different standards to classify development activities. For example, ISO/IEC 14764 [27, 39] specifies four types of maintenance: corrective, preventive, adoptive and perfective. We are the first to examine what should be the classification standard for assessing development value, and propose the impact coding scheme. Furthermore, Li et al. [38] conducted a developer survey to grade the influence of different types of software changes (e.g., checking “very influential” for “fix pervasive bugs”). In contrast, we use L2R to learn the weights of each category from a large data set of concrete commit evaluations. (Since we use different category definitions, the resulting weights are hardly compared to their grades.)

Machine learning has found its way to software engineering work. For example, it is used for bug prediction [21, 29], bug triage [13, 68] and bug assignment [3]. Particularly, Menzies et al. [46] and Lamkanfi et al. [35, 36] used text classification to predict the severity of a bug from the text of its bug report. Their models under evaluation include Bayes, SVM and rule learning. Our context learning follows a similar approach, but it is among the first massive evaluations of CNN and RNN models for commit classification. We regard it as future work to design more efficient NLP and ML models for our task.

8 CONCLUSION

There are commercial, pedagogical, and stewardship reasons to evaluate the impact of individual code contributions to a large codebase. This task is difficult for developers to do manually, not only because of the subjectivity inherent in the task but also because few developers have a wide enough view of the entire project to do it effectively and in a manner well-calibrated to their fellow developers. To make the process both objective and amenable to automation, we postulated that a given code contribution has both structural and nonstructural value, and presented a combination of a PageRank-inspired algorithm and a machine learning model trained from developer’s holistic rankings of their own contribution that captures and combines these two kinds of value. We also identified a promising future direction in automatically classifying commits, to further automate the process of computing nonstructural value. We hope our approach will enable and support an even stronger ecosystem of contribution-based projects with a “very long tail” of contributors as well as giving developers and project leaders better insights on the relative strengths of their own contributors and code.

Acknowledgment

This is a joint work with my colleagues Jinglei Ren, Qingda Hu, Alex Stennet, Wojciech Koszek and my advisor Armando Fox. I’d like to thank Jinglei for his invaluable contribution in overall algorithm and experiment design, and coining the term DevRank; Qingda for training the commit message classifier; Alex for providing Javascript support for DevRank; Wojciech for insightful feedback and many fruitful discussions; Armando for his support and guidance both in research and life throughout my graduate career.

REFERENCES


