

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

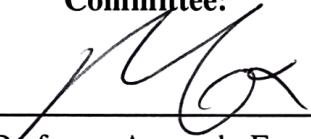
by Hezheng Yin

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)

Quantifying the Development Value of Code Contributions

Hezheng Yin

UC Berkeley
hezheng.yin@berkeley.edu

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 runs 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.js¹ 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 [57]. 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.

¹<https://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 \mathbf{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 α and teleports to another page with probability $(1 - \alpha)$. The stochastic column vector \mathbf{p} stores probabilities of the surfer teleporting to all pages, respectively. We also need \mathbf{e} , an auxiliary row vector of all ones. Then PageRank is a Markov chain with the stationary distribution \mathbf{v} , a column vector that satisfies

$$(\alpha\mathbf{P} + (1 - \alpha)\mathbf{pe})\mathbf{v} = \mathbf{v}.$$

To calculate \mathbf{v} , we perform the iteration

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

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

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 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 \mathbf{P} and \mathbf{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

$$P_{i,j} = \begin{cases} \frac{|f_i|}{\sum_{f \in F_j} |f|} & \text{if } f_i \in F_j, \\ 0 & \text{otherwise.} \end{cases}$$

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 Git², 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.

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

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 degradation 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

²<https://git-scm.com>

memory leak³. Meanwhile, a fix on the program design impacts multiple collaborating components of the program (e.g., a concurrent bug⁴).

- 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 function⁵). 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 use 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: $v = \varphi(d, t)$. 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:

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

so that the average error between the true value and $\varphi(d, t)$ 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 φ . 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.

³<https://github.com/torvalds/linux/commit/0147ebc>

⁴<https://github.com/torvalds/linux/commit/1dbb670>

⁵<https://github.com/torvalds/linux/commit/11afbde>

```
/drivers/pci/host/pcie-rockchip.c
@@ -973,4 +973,4 @@
+   if (region_no == 0) {
-       if (AXI_REGION_0_SIZE < (2ULL << num_pass_bits))
+       return -EINVAL;
-       return -EINVAL;
+   }
}
```

Commit message: “PCI: rockchip: Indent “if” statement body”
Developer description: “code cleanup”

(a) Cleanup (code style)

```
/fs/btrfs/extent_io.h
@@ -301,7 +301,7 @@
+ static inline int set_extent_defrag(...,
- u64 end, struct extent_state **cached_state, gfp_t mask)
+ u64 end, struct extent_state **cached_state)
{
+   return set_extent_bit(tree, start, end,
+       EXTENT_DEALLOC | EXTENT_UPTODATE | EXTENT_DEFRAG,
-   NULL, cached_state, mask);
+   NULL, cached_state, GFP_NOFS);
}
...
```

Commit message: “btrfs: sink gfp parameter to set_extent_defrag”
Developer description: “cleanup, preparatory work”

(b) Maintenance (code structure)

```
/drivers/net/virtio_net.c
@@ -892,18 +892,17 @@
- ctx = (void *) (unsigned long) len;
+ ...
+ sg_init_one(rq->sg, buf, len);
+ ctx = (void *) (unsigned long) len;
+ err = virtqueue_add_inbuf_ctx(rq->vq, rq->sg, 1, buf, ctx,
+   gfp);
```

Commit message: “virtio_net: fix truesize for mergeable buffers”
Developer description: “performance improvement”

(c) Improvement (logic)

```
/arch/arm/boot/dts/sun5i.dtsi
@@ -585,7 +585,19 @@
+   lradc: lradc@01c22800 {
+       ...
+       reg = <0x01c22800 0x100>;
+       interrupts = <31>;
+       ...
+   };
+
+   codec: codec@01c22c00 {
+       ...
+       reg = <0x01c22c00 0x40>;
+       interrupts = <30>;
+       ...
+   };
```

Commit message: “ARM: sun5i: Add the Audio codec DT node”
Developer description: “enables audio support on a board”

(d) Feature (usage)

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 $\{(c_i^{(1)}, c_i^{(2)}), y_i\}, i =$

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 $\{y_i\}, i = 1, 2, \dots, m$ 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, $x_i^{(1)} = [x_{i,1}^{(1)}, x_{i,2}^{(1)}, \dots, x_{i,k}^{(1)}]_{\% \text{ share}}$ and $x_i^{(2)} = [x_{i,1}^{(2)}, x_{i,2}^{(2)}, \dots, x_{i,k}^{(2)}]$, 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 φ . 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.*

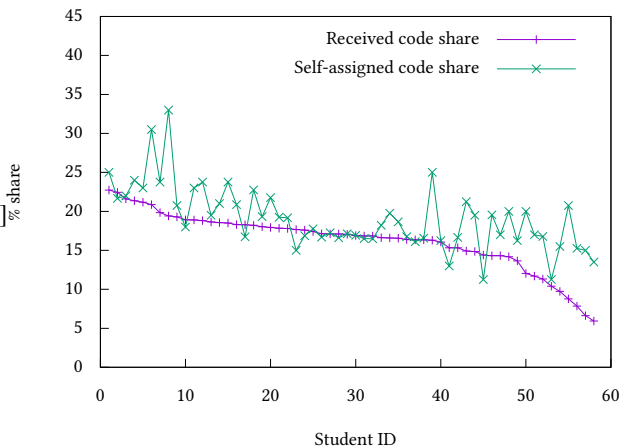


Figure 2: Mean shares for each student. Students are sorted by the mean code share.

To address the research questions, we assemble two data sets: (1) course surveys of students required to assess teammates’ contributions in a software engineering course; (2) value comparisons of commits and explanations collected from open source developers;

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

Classification	Sample explanations	Percentage
Explicit	“[add a new feature to support flame-graph more efficiently] is more valuable than [improve user experience]” “[performance boost] is more valuable than [fix a bug which is not super important]” “[cleans up code] is more valuable than [is trivial textual fix]” “[fixes a bug] is more valuable than [is a fairly minor performance & code clarity optimisation]” “[a fix for memory leak] is more valuable than [a fix for documentation]”	83.3%
Implicit	“[improve user experience] is more valuable than [just a trivial change to export necessary functions for future use]” “[protect structure contents] is more valuable than [optimizes non critical path]” “[just shrinks kernel size] is more valuable than [just a cosmetic spelling fix]” “[spec compliance] is more valuable than [tool for static code checking]” “[Make sure that RESET name for dts is accurate] is more valuable than [cleanup]”	
None	“[commit body] is more valuable than [no commit body]” “[this null dereference doesn’t affect normal users] is more valuable than [tiny impact, doesn’t affect normal users at all]” “[networking stuff is important] is more valuable than [code cleanup]” “[is required for newer >= r6 MIPS CPUs to work] is more valuable than [is trivial contact information]” “[show aggregated stat and task info when CPU is idle] is more valuable than [show raw value of fields in a trace record]”	16.7%

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

assigns to the whole team are normalized to 100%. We see that teammates 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 0.10% to 36.95%.

We also examined the average correlation between students’ ratings. 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 pairs, 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 communications with students (which we cannot reveal due to privacy considerations) also suggested gaming/collusion among team members as well as personal biases.

Methodologically, therefore, in our larger survey we ask developers 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 prior to v4.4. We count commits in the master branch as well as other branches, but exclude merge commits, which serve to fuse changes from different branches but do not generally contribute code changes themselves. We invited the 100 developers who made the largest numbers of commits selected from the above scope to complete our survey; of these, 35 developers contributed 772 comparisons to our survey. We asked developers to phrase their reasoning for ranking a given pair of commits as follows:

“Commit A does/is/has —, and Commit B does/is/has —. Thus, Commit A is more valuable than Commit B.”

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

explanations and their corresponding proportions in all explanations 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 classification, 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 generalize (or at least our current scale of developer surveys, though it is already extensive, still do not support such generalization).

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

To test Hypothesis 2, we compared DevRank and LOC’s performance in predicting developers’ comparisons of commits. The commit 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 chance at 50%. By incorporating the structural information in the codebase, DevRank achieved an accuracy of 59.0%, showing support for Hypothesis 2.

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

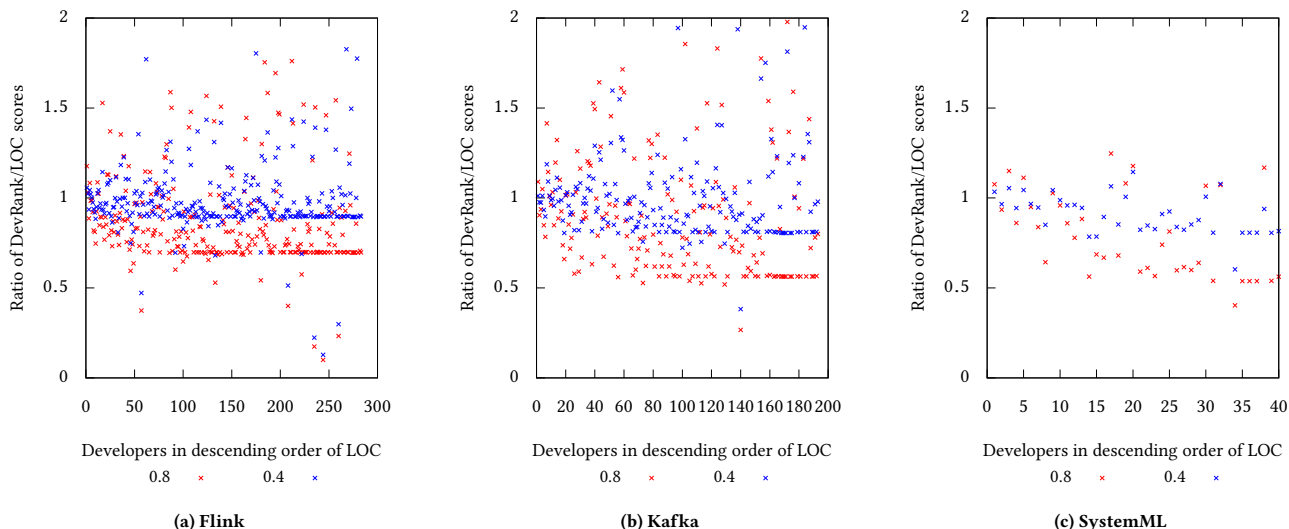


Figure 3: Developers’ relative changes between scores calculated by LOC counting and DevRank. $y = 1$ means both scores for a developer are identical. Legend keys, i.e., 0.4, 0.8, are the values of α in DevRank.

Feature	Model	Accuracy
LOC	-	53.2%
DevRank	-	59.0%
Coding	Ranking SVM	69.1%
Coding + LOC	Ranking SVM	70.2%
DevRank + Coding	Ranking SVM	73.1%

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

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 -90.0% / -73.3% / -59.7% to $24.3\times/16.7\times/2.65\times$ 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 α becomes larger. α can largely influence the share distribution of DevRank.

5 DISCUSSION

We discuss two factors of the threats 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. On 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 barely 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 inference 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 database⁶ 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 maintenance 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.

	Maint.	Feature	Improv.	Bug
# commits	329	1517	14136	28818
BoW-title	0.28	0.34	0.63	0.85
BoW-message	0.2	0.33	0.62	0.85
CNN-title	0.37	0.39	0.65	0.86
CNN-message	0.46	0.39	0.64	0.87
RNN-title	0.4	0.35	0.67	0.86
RNN-message	0.33	0.36	0.68	0.87

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 follow 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 directly 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

- [1] Jong-hoon An, Avik Chaudhuri, and Jeffrey S. Foster. 2009. Static Typing for Ruby on Rails. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE ’09)*. IEEE Computer Society, 590–594. <https://doi.org/10.1109/ASE.2009.80>
- [2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’11)*. 459–472. <https://doi.org/10.1145/1926385.1926437>
- [3] John Anvik. 2006. Automating Bug Report Assignment. In *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06)*. ACM, 937–940. <https://doi.org/10.1145/1134285.1134457>
- [4] Carliss Baldwin, Alan MacCormack, and John Rusnak. 2014. Hidden structure: Using network methods to map system architecture. *Research Policy* 43, 8 (2014),

Quantifying the Development Value of Code Contributions

- 1381 – 1397. <https://doi.org/10.1016/j.respol.2014.05.004>
- [5] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. Graph-based Analysis and Prediction for Software Evolution. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE, 419–429. <http://dl.acm.org/citation.cfm?id=2337223.2337273>
- [6] Marco Biazzi, Martin Monperrus, and Benoit Baudry. 2014. On Analyzing the Topology of Commit Histories in Decentralized Version Control Systems. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. 261–270. <https://doi.org/10.1109/ICSME.2014.48>
- [7] Stamatia Bibi, Apostolos Ampatzoglou, and Ioannis Stamelos. 2016. A Bayesian Belief Network for Modeling Open Source Software Maintenance Productivity. In *Proceedings of The 12th International Conference on Open Source Systems (OSS '16)*. 32–44.
- [8] Stefan Biffl, Aybuke Aurnum, Barry Boehm, Hakan Erdogmus, and Paul Grünbacher (Eds.). 2009. *Value-Based Software Engineering*. Springer.
- [9] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting It All Together: Using Socio-technical Networks to Predict Failures. In *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE '09)*. IEEE, 109–119. <https://doi.org/10.1109/ISSRE.2009.17>
- [10] B. Boehm and Li Guo Huang. 2003. Value-based software engineering: a case study. *IEEE Software* 36, 3 (Mar 2003), 33–41. <https://doi.org/10.1109/MC.2003.1185215>
- [11] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7 (WWW-7)*. 107–117. <http://dl.acm.org/citation.cfm?id=297805.297827>
- [12] Yunbo Cao, Jun Xu, Tie-Yan Liu, Hang Li, Yalou Huang, and Hsiao-Wuen Hon. 2006. Adapting Ranking SVM to Document Retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '06)*. ACM, New York, NY, USA, 186–193. <https://doi.org/10.1145/1148170.1148205>
- [13] Davor Cubranic and Gail C. Murphy. 2004. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE '04)*.
- [14] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work (CSCW '12)*. 1277–1286. <https://doi.org/10.1145/2145204.2145396>
- [15] Cleidson de Souza, Jon Froehlich, and Paul Dourish. 2005. Seeking the Source: Software Source Code As a Social and Technical Artifact. In *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work (GROUP '05)*. 197–206. <https://doi.org/10.1145/1099203.1099239>
- [16] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. <https://doi.org/10.1109/CVPR.2009.5206848>
- [17] M. Denne and J. Cleland-Huang. 2004. The incremental funding method: data-driven software development. *IEEE Software* 21, 3 (May 2004), 39–47. <https://doi.org/10.1109/MS.2004.1293071>
- [18] Roy T. Fielding. 1999. Shared Leadership in the Apache Project. *Commun. ACM* 42, 4 (April 1999), 42–43. <https://doi.org/10.1145/299157.299167>
- [19] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. 2003. An Efficient Boosting Algorithm for Combining Preferences. *J. Mach. Learn. Res.* 4 (Dec. 2003), 933–969. <http://dl.acm.org/citation.cfm?id=945365.964285>
- [20] Thomas Fritz, Gail C. Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. 2014. Degree-of-knowledge: Modeling a Developer’s Knowledge of Code. *ACM Trans. Softw. Eng. Methodol.* 23, 2, Article 14 (April 2014), 42 pages. <https://doi.org/10.1145/2512207>
- [21] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing Fine-grained Source Code Changes and Code Churn for Bug Prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, 83–92. <https://doi.org/10.1145/1985441.1985456>
- [22] GitHub. 2017. Viewing contribution activity in a repository. <https://help.github.com/articles/viewing-contribution-activity-in-a-repository/>. (2017).
- [23] David F. Gleich. 2015. PageRank Beyond the Web. *SIAM Rev.* 57, 3 (Aug. 2015), 321–363.
- [24] Claudia Hauff and Georgios Gousios. 2015. Matching GitHub Developer Profiles to Job Advertisements. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, 362–366. <http://dl.acm.org/citation.cfm?id=2820518.2820563>
- [25] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. 2000. Large margin rank boundaries for ordinal regression. 88 (01 2000).
- [26] Wei Hu and Kenny Wong. 2013. Using Citation Influence to Predict Software Defects. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. 419–428. <http://dl.acm.org/citation.cfm?id=2487085.2487162>
- [27] ISO/IEC. 2006. ISO/IEC 14764:2006 Software Engineering – Software Life Cycle Processes – Maintenance. <https://www.iso.org/standard/39064.html>. (Sept. 2006).
- [28] Chris Jensen and Walt Scacchi. 2007. Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, 364–374. <https://doi.org/10.1109/ICSE.2007.74>
- [29] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. 2014. Dictionary Learning Based Software Defect Prediction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, 414–423. <https://doi.org/10.1145/2568225.2568320>
- [30] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From Developer Networks to Verified Communities: A Fine-grained Approach. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 563–573. <http://dl.acm.org/citation.cfm?id=2818754.2818824>
- [31] M. Jorgensen, B. Boehm, and S. Rifkin. 2009. Software Development Effort Estimation: Formal Models or Expert Judgment? *IEEE Software* 26, 2 (March 2009), 14–19. <https://doi.org/10.1109/MS.2009.47>
- [32] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *CoRR* abs/1607.01759 (2016). <http://arxiv.org/abs/1607.01759>
- [33] Dishan Kamdar and Linn Van Dyne. 2007. The joint effects of personality and workplace social exchange relationships in predicting task performance and citizenship performance. *Journal of Applied Psychology* 92, 5 (2007), 1286–1298.
- [34] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP '14)*. Association for Computational Linguistics (ACL), Doha, Qatar, 1746–1751.
- [35] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. 2010. Predicting the severity of a reported bug. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR '10)*. 1–10. <https://doi.org/10.1109/MSR.2010.5463284>
- [36] Ahmed Lamkanfi, Serge Demeyer, Quinten David Soetens, and Tim Verdonck. 2011. Comparing Mining Algorithms for Predicting the Severity of a Reported Bug. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11)*. 249–258. <https://doi.org/10.1109/CSMR.2011.31>
- [37] Philip Levis. 2012. Experiences from a Decade of TinyOS Development. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*. 207–220. <http://dl.acm.org/citation.cfm?id=2387880.2387901>
- [38] Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F. Bissyandé, David Lo, and Yves Le Traon. 2016. Watch out for This Commit! A Study of Influential Software Changes. *CoRR* abs/1606.03266 (2016). <http://arxiv.org/abs/1606.03266>
- [39] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. 1978. Characteristics of Application Software Maintenance. *Commun. ACM* 21, 6 (June 1978), 466–471. <https://doi.org/10.1145/359511.359522>
- [40] Luis Lopez-Fernandez, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2004. Applying social network analysis to the information in CVS repositories. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR '04)*. 101–105.
- [41] Jonathan I. Maletic and Michael L. Collard. 2015. Exploration, Analysis, and Manipulation of Source Code Using srcML. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, 951–952. <http://dl.acm.org/citation.cfm?id=2819009.2819225>
- [42] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. 2013. Impression Formation in Online Peer Production: Activity Traces and Personal Profiles in Github. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work (CSCW '13)*. ACM, 117–128. <https://doi.org/10.1145/2441776.2441792>
- [43] Thilo Mende and Rainer Koschke. 2010. Effort-Aware Defect Prediction Models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*. IEEE Computer Society, Washington, DC, USA, 107–116. <https://doi.org/10.1109/CSMR.2010.18>
- [44] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. 2008. Predicting Failures with Developer Networks and Social Network Analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. 13–23. <https://doi.org/10.1145/1453101.1453106>
- [45] Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat. 2013. Mining Software Repositories for Accurate Authorship. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. 250–259. <https://doi.org/10.1109/ICSM.2013.36>
- [46] T. Menzies and A. Marcus. 2008. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance (ICSM '08)*. 346–355. <https://doi.org/10.1109/ICSM.2008.4658083>
- [47] Martin Michlmayr. 2004. Managing Volunteer Activity in Free Software Projects. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC '04)*. 39–39. <http://dl.acm.org/citation.cfm?id=1247415.1247454>
- [48] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying high impact fix-inducing changes. *Empirical Software Engineering* 21, 2 (01 Apr 2016), 605–641. <https://doi.org/10.1007/s10664-015-9370-z>

- [49] Ivan Mistrik, Rami Bahsoon, Rick Kazman, and Yuanyuan Zhang (Eds.). 2014. *Economics-Driven Software Architecture*. Morgan Kaufmann.
- [50] Audris Mockus and James D. Herbsleb. 2002. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, 503–512. <https://doi.org/10.1145/581339.581401>
- [51] Paychex. 2016. Employee Retention: What Makes Employees Stay or Leave. <https://www.paychex.com/articles/human-resources/employee-retention-what-makes-employees-stay-leave>. (Aug. 2016).
- [52] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, 2–12. <https://doi.org/10.1145/1453101.1453105>
- [53] Eric S. Raymond. 2001. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media.
- [54] Debbie Richards. 2009. Designing project-based courses with a focus on group formation and assessment. *ACM Transactions on Computing Education (TOCE)* 9, 1 (2009), 2.
- [55] Martin P. Robillard and Gail C. Murphy. 2002. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, 406–416. <https://doi.org/10.1145/581339.581390>
- [56] W. Scacchi. 2002. Understanding the requirements for developing open source software systems. *IEE Proceedings - Software* 149, 1 (Feb 2002), 24–39. <https://doi.org/10.1049/ip-sen:20020202>
- [57] Walt Scacchi. 2007. Free/Open Source Software Development: Recent Research Results and Emerging Opportunities. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers (ESEC-FSE companion '07)*. 459–468. <https://doi.org/10.1145/1295014.1295019>
- [58] Emad Shihab, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2013. Is Lines of Code a Good Measure of Effort in Effort-aware Models? *Inf. Softw. Technol.* 55, 11 (Nov. 2013), 1981–1993. <https://doi.org/10.1016/j.infsof.2013.06.002>
- [59] Leif Singer, Fernando Figueira Filho, Brendan Cleary, Christoph Treude, Margaret-Anne Storey, and Kurt Schneider. 2013. Mutual Assessment in the Social Programmer Ecosystem: An Empirical Investigation of Developer Profile Aggregators. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work (CSCW '13)*. ACM, 103–116. <https://doi.org/10.1145/2441776.2441791>
- [60] Robert Speer and Joanna Lowry-Duda. 2017. ConceptNet at SemEval-2017 Task 2: Extending Word Embeddings with Multilingual Relational Knowledge. In *Proceedings of the 11th International Workshop on Semantic Evaluations (SemEval-2017)*. Association for Computational Linguistics (ACL), Vancouver, Canada, 85–89.
- [61] Megan Squire. 2013. Project Roles in the Apache Software Foundation: A Dataset. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, 301–304. <http://dl.acm.org/citation.cfm?id=2487085.2487142>
- [62] Damian A. Tamburri, Patricia Lago, and Hans van Vliet. 2013. Organizational Social Structures for Software Engineering. *ACM Comput. Surv.* 46, 1, Article 3 (July 2013), 35 pages. <https://doi.org/10.1145/2522968.2522971>
- [63] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, 356–366. <https://doi.org/10.1145/2568225.2568315>
- [64] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's Talk About It: Evaluating Contributions Through Discussion in GitHub. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. 144–154. <https://doi.org/10.1145/2635868.2635882>
- [65] L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye. 2009. Linux kernels as complex networks: A novel method to study evolution. In *2009 IEEE International Conference on Software Maintenance (ICSM '09)*. 41–50. <https://doi.org/10.1109/ICSM.2009.5306348>
- [66] H. Wu, L. Shi, C. Chen, Q. Wang, and B. Boehm. 2016. Maintenance Effort Estimation for Open Source Software: A Systematic Literature Review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 32–43. <https://doi.org/10.1109/ICSME.2016.87>
- [67] Yutaka Yamauchi, Makoto Yokozawa, Takeshi Shinohara, and Toru Ishida. 2000. Collaboration with Lean Media: How Open-source Software Succeeds. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW '00)*. 329–338. <https://doi.org/10.1145/358916.359004>
- [68] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer. 2013. Categorizing bugs with social networks: A case study on four open source software communities. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. 1032–1041. <https://doi.org/10.1109/ICSE.2013.6606653>
- [69] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. 531–540. <https://doi.org/10.1145/1368088.1368161>