Towards Quantifying the Development Value of Code Contributions

Jinglei Ren* Microsoft Research, China
Hezheng Yin* UC Berkeley, USA
Qingda Hu Tsinghua University, China
Armando Fox UC Berkeley, USA
Wojciech Koszek FreeBSD Project, USA

Persper Foundation, USA
{jinglei,hezheng,qingda,fox,wkoszek}@persper.org

ABSTRACT

Quantifying the value of developers' code contributions to a software project requires more than simply counting lines of code or commits. We define the *development value* of code as a combination of its structural value (the effect of code reuse) and its non-structural value (the impact on development). We propose techniques to automatically calculate both components of development value and combine them using Learning to Rank. Our preliminary empirical study shows that our analysis yields richer results than human assessment or simple counting methods, and demonstrates the potential of our approach.

CCS CONCEPTS

• **General and reference → Metrics**; • **Software and its engineering → Software post-development issues**;

KEYWORDS

value of code, development value, static program analysis, software repository mining, call graph, learning to rank

ACM Reference Format:
Jinglei Ren, Hezheng Yin, Qingda Hu, Armando Fox, and Wojciech Koszek. 2018. Towards Quantifying the Development Value of Code Contributions. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3264842

1 INTRODUCTION

Developers contribute code to software project repositories. Those code contributions are typically characterized by simple metrics such as the number of commits (NOC) or lines of code (LOC). For example, GitHub uses NOC to rank developers of a project [13]. Expertise Browser [33], a classic tool for identifying developer expertise, uses the number of changed LOCs as an indicator. Such metrics measure the *amount* of code contributions, rather than their *value*. For example, a function at the core of the application logic is probably more valuable than an auxiliary script.

There are many use cases where we need to compare and recognize the value of different developers' contributions. While traditional value-based software engineering [3, 5, 32] focuses on creating *economic value* as a way to prioritize resource allocation and scheduling, other measurements of value may be more relevant in some of the use cases. One example is that instructors need a tool to evaluate individual students' code contributions to group projects (besides non-code contributions). Such measurement of code contributions has nothing to do with economic returns. As a second example, an engineering manager may need a quantitative measurement of team members' performance. Additionally, for free and open source software (FOSS) projects, developers' contributions heavily influence collaboration, coordination, and leadership [26, 38]. Finally, software engineering researchers observe development activities *per se,* but not necessarily their economic returns. As the above Expertise Browser case shows, a new quantitative tool for the code contributions would help better understand software development processes.

In this paper, we outline our work to quantify *the value of code contributions in software development*, that is, the *effect on development activities* of contributed code. In general, code that addresses a time consuming development task has higher value than code that addresses an easier task; code that saves a huge amount of other developers' effort has higher value than code that saves little. Therefore, 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 effect of the code structure on development activities: A function that is called by many callers reduces the development effort of those callers and thus tends to be of high value. Based on this observation, we design *DevRank*, a variant of PageRank to derive development value from the function call graph. On the other hand, not all development value is reflected in code structure. Through interviewing three seasoned open source developers, we find that developers judge the value by classifying the impact of commits. Leveraging natural language processing (NLP) and machine learning (ML) techniques, we explore the possibility to automate the commit classification by commit messages that usually describe what impact the code makes. Finally, we train a learning-to-rank (L2R) model to find the best combination of the structural and non-structural value to generate an overall score of development value.

*Both authors contributed equally to this work.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264842

2 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 (§2.1). The *non-structural value* reflects the impact to the project in a way that code structure alone does not (§2.2). This section describes how we compute both kinds of value, and how we combine them to get an overall value of a code contribution (§2.3).

2.1 Structural Value: DevRank

In most imperative programming languages, a function (procedure, method) is a basic unit of program structure. The development value of a function is based not only on the effort spent creating the function, but also the effort saved when other code calls the function. The structural component of the development value (structural value) is captured by our graph-based algorithm *DevRank*, which is an extension of the original PageRank algorithm.

PageRank [7] is the basis for Google Web Search, and finds applications in various domains [15]. 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, with probability α, the surfer randomly selects a link on that page and visits the linked page; (2) with probability $1-\alpha$, the surfer teleports to a random page and continues. The damping factor α is a fixed probability chosen in advance. 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.

To compute each function's structural value, we analyze the code repository's *static function-call graph*. Although program *execution* never randomly jumps to an irrelevant function as in PageRank, we find that PageRank is a surprisingly convenient model to characterize code *development*. We interpret random teleportation as navigating the *development activities* of the code, rather than execution behavior. In addition, we consider development history as revealed by commits of code in a revision-control system over time.

In DevRank, the hypothetical "surfer" becomes a *development sniffer*, whose task is to detect development effort. We assume that the development effort spent on a function is revealed by the total LOCs of all changes that result in the function across the development history. We believe it can more precisely quantify the development effort than only counting the LOCs at the latest snapshot. Based on this assumption, the behavior of the sniffer is constructed in the following way. (1) Upon arriving at a function, with probability α, the sniffer visits one of the called functions with probabilities proportional to the development efforts of those functions. 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) With probability $1-\alpha$, the sniffer teleports to a random function with a probability proportional to the development effort 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 the function saves other developers. Therefore, it reflects the development value of a function.

After computing DevRank scores for functions, we can distribute development value of functions to commits, and further to developers. This is done by allocating the value of a function to all commits that change the function, proportionally to the size of their changes (i.e., the number of changed LOCs) and then assigning the value of commits to their corresponding authors. In this way, developers receive credits for their contributions.

2.2 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 proposal for measuring non-structural value is inspired by how human developers assess a code contribution's non-structural impact on the project. We interviewed three open source developers: an author of a popular Twitter client and two FreeBSD developers each with over ten years of experiences. Specifically, we asked them to give a free-form answer to the following question: what procedure would you use to compare the value of commits in a project? Despite the vast answer space, all three interviewees mentioned that they would start with commit classification by what kind of impact a commit has on the project. One of the FreeBSD developers even gave a comprehensive hierarchy of commit-value: "fix for build errors > fix for severe non-build errors > important new features > fix for severe speculative errors > fix for minor errors > regular new features > cosmetic errors > source code hygiene".

Therefore, the impact type of a commit is an important feature for determining its non-structural value. We introduce *impact coding* to capture such non-structural value. Impact coding classifies a commit according to a predefined set of impact categories. Previous work defines related categories of development activities [17, 23], but focuses on only certain aspects of software development (e.g., maintenance). To construct a set of impact categories that comprehensively represent non-structural value, we plan to conduct a large-scale survey and let developers freely express their reasons for commit value comparisons, and analyze their responses following the grounded theory approach [14]. This approach will inductively generate impact categories during the labeling of data.

The impact-based commit classification has the potential to be automated because many communications among developers are computer-mediated [35, 41]. Each commit is associated with a *context* implicitly built in development activities, which includes natural-language descriptions of the commit in a bug/issue tracking system, a pull request or a commit message [10, 39]. Those natural-language descriptions provide an adequate corpus for constructing a machine learning model to infer the impact category of a commit. We refer to this approach as *context learning* and give a concrete example in §3.3.

2.3 Combining DevRank and Impact Coding

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 $\mathbf{t}$. Our goal is to find a function φ that combines them: $v = \varphi(d, \mathbf{t})$. In our solution, d is the DevRank score, and $\mathbf{t}$ is a one-hot vector encoding of the commit category.

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

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

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

Unfortunately, developers find it very hard to directly score code values in a free-form manner, e.g., giving one commit 0.17 and another 0.06, so we lack the reliable ground truth in that form. Instead, we can ask developers to compare *commit pairs* of *the same* author and identify which of each pair is more valuable. It is a much easier question to answer and eliminates the influence of personal interests and interpersonal relationships. Based on this "pairwise ground truth," we can use a learning to rank (L2R) algorithm [8, 11, 16] to determine φ. We use d and $\mathbf{t}$ as the input features to Ranking SVM [16]. After training, we take the weight vector of the SVM as $\mathbf{w}$ in $\varphi(d, \mathbf{t})$. This method allows us to combine the structural and non-structural value scores for each commit to determine its overall development value score.

3 PRELIMINARY EXPERIMENTS

Our current experiments make three points: a case study in the education setting shows the limitation of human assessment and motivates our measurement; the different results of DevRank and LOC-counting show the effects of capturing structural value; the performance of mainstream ML models in classifying non-structural impacts reveals both opportunities and challenges.

To collect empirical evidence, we assemble two data sets: (1) course surveys of students assessing teammates' contributions in a software engineering course; (2) over 250k issues and associated commit messages collected from Apache Software Foundation projects, for training the context learning models.

3.1 Human Assessment Is Not Reliable

Developers may assess code value through their understanding of the code and their impressions of other developers. However, those factors are biased by social factors and personal interests. To better understand the validity of human assessment, we surveyed 10 teams of students (58 in total) from an undergraduate software engineering course at UC Berkeley. During a 8-week project timespan, individual students were asked to evaluate their teammates' code contributions every 2 weeks, by assigning team members (including themselves) shares normalized to total 100%.

We keep statistics of each student's self-assigned share and shares received from teammates. First, we see that most students receive very different amounts of shares from their teammates, showing the subjectivity in human value assessment. For every pair of students in a team, we computed the Pearson's r coefficient between the shares they assign to other students. On average, the Pearson's r is only 0.52, indicating moderate level of agreement among students. Second, students' self-assigned shares are 18.36% more than their peer-assigned shares on average, suggesting that their self-assessment is subjectively more optimistic than their peer assessment. This reflects the influence of personal interests.

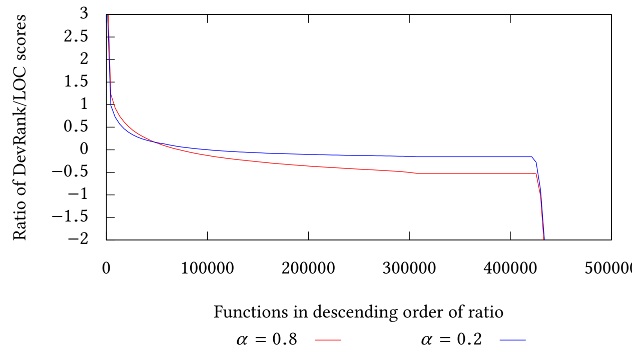

Figure 1: Ratios between a function's DevRank score and its LOC. $y = 1$ means both scores for a function are identical. Legend keys, i.e., 0.2, 0.8, are the values of α in DevRank.

3.2 LOCs Incompletely Capture Value

To observe the effects of DevRank, we evaluated the code contributions in the Linux kernel code at the function level using DevRank and LOC respectively and compared their results.

We first extracted LOC information for each function by parsing the source code of release v4.14 using a static analysis tool, srcML [25]. We collected 458,054 functions from 45,959 source files. Then we computed DevRank values for all functions by analyzing 5,000 commits on the master branch before release v4.14. We experimented with multiple α values and empirically set α to 0.5 for most following analysis.

If we rank functions by their LOCs and DevRank values respectively, the generated rankings by LOC and DevRank are very different: the Kendall's τ between these two rankings is 0.64, only indicating moderate level of agreement. We further explore DevRank's effects on different functions by computing the ratio between each function's DevRank value and normalized number of LOCs. Figure 1 shows this ratio for all functions in descending order. A positive log-scale ratio suggests that the function's DevRank value is larger than its normalized number of LOCs. We observe from Figure 1 that DevRank amplifies contributions of a small portion of functions, which account for 20.9% of the total number of functions.

One limitation of DevRank algorithm is that it may over-estimate the contributions of some simple utility functions, including getters and setters, because of their high in-degrees in the call graph. For example, `check_memory_region` is ranked among the most valuable functions under the `mm` directory. To avoid this issue, we filtered out these simple utility functions by setting a threshold of the number of LOCs (20 by default) before ranking.

Table 1 shows the 5 most valuable functions under the `mm` (memory management) directory by DevRank and LOC counting after filtering out the simple utility functions. We look into the top functions `slob_free` and `shrink_page_list` on the two rankings, respectively, and showcase how DevRank better models the development value of code. `shrink_page_list` reclaims page frames, while `slob_free` reclaims SLOB blocks. `shrink_page_list` has

Table 1: Most valuable functions under `mm` directory

	#	Function	File
DevRank	1	slob_free	mm/slob.c
	2	mempool_alloc	mm/mempool.c
	3	dma_pool_free	mm/dmapool.c
	4	kasan_slab_free	mm/kasan/kasan.c
	5	mempool_free	mm/mempool.c
LOC	1	shrink_page_list	mm/vmscan.c
	2	shmem_getpage_gfp	mm/shmem.c
	3	__vma_adjust	mm/mmap.c
	4	balance_dirty_pages	mm/page-writeback.c
	5	__alloc_pages_slowpath	mm/page_alloc.c

a larger LOC number than `slob_free`, but both have to check and deal with many cases in their algorithms. If we only compare LOCs, `shrink_page_list` seems likely to embody more development value. However, `slob_free` is called about 3,500 times[1] more than `shrink_page_list`. Every time `slob_free` is called, the slob allocator saves developers effort to deal with memory allocation. That is also a form of development value we should consider. Moreover, as the memory allocator is so frequently used, many efforts have been spent on improving its efficiency in Linux [6, 21, 27]. We believe that taking into account the call structure as in DevRank gives a fairer evaluation of `slob_free`'s development value.

3.3 Automatic Impact Classification

We leverage the JIRA issue database[2] used by many Apache Software Foundation projects. In the database, developers label issues with predefined types (feature, improvement, bug fix, maintenance). We collect 267,446 issues and their associated commit messages from 139 Apache projects that have top most issues.

We explore three main NLP + ML models: bag-of-words (BoW) [20], a convolutional neural network (CNN) [22], and a recurrent neural network (RNN). For each model, we experiment with two types of inputs: the commit message title and the full complete message. We adopt ConceptNet Numberbatch (CN) word embeddings [37]. For all issue types, we show F1 scores in Table 2. We can see that CNN and RNN have comparable performance but outperform the bag-of-words model. The best accuracy that our models achieve among all classes is 78.0%, using RNN on full commit messages. Moreover, using commit messages constantly outperforms using commit titles in all models. That should be because messages contain more information than titles.

The results show that some categories of commits are easier to infer than others. For example, the best F1 score for classifying Bug commits is 0.873 by the RNN model using commit messages, while the best F1 score for classifying maintenance commits is only 0.459 by the CNN model using commit messages. A possible reason is that those categories have different numbers of commits in the data set: Bug fixes are dominant, so training for them is more effective.

As a first step, our analysis shows the possibility of automatically classifying commit impact using solely the commit message and a neural network model. Overall, identification of Bug fixes and Improvements can be regarded as usable for DevRank, but that of less-represented categories is still too low. As part of our future

[1]Most of the calls are through `kfree`.
[2]https://issues.apache.org/

Table 2: Performance (F1 score) of three NLP + ML models for context learning.

	Maint.	Feature	Improv.	Bug	*Accuracy*
# commits	410	1482	13648	29261	
BoW-title	0.287	0.339	0.631	0.851	75.1%
BoW-message	0.204	0.334	0.622	0.853	75.5%
CNN-title	0.365	0.391	0.652	0.865	77.0%
CNN-message	**0.459**	**0.391**	0.641	0.869	77.5%
RNN-title	0.401	0.344	0.665	0.863	77.0%
RNN-message	0.326	0.360	**0.677**	**0.873**	**78.0%**
Average	0.330	0.359	0.648	0.862	

work, we hope to improve the accuracy of automatic classification with a more comprehensive data set and optimized models.

4 ONGOING WORK

We are planning a larger survey of developers to collect pairwise commit-comparison results and reasons from developers as the ground truth (§2.3) for analysis and training. Such a data set will help us construct the impact categories for non-structural analysis and also allow us to experiment with more advanced machine learning models: both context learning and L2R should improve with more training data. We will open the data set for public use, to hopefully stimulate collaborative efforts in this research direction.

Our current implementation only supports C/C++ and Java. We are adding support for dynamic languages such as Python and JavaScript, which requires implementing static typing [1, 2].

5 RELATED WORK

PageRank-like algorithms have been used to portray developers [12, 18, 30] or their social relationships [4, 24, 29, 34]. Our DevRank is a variant of PageRank adapted to reflect the development value. Effort-aware models [28, 31] consider the development effort in software engineering, and different effort estimation schemes have been proposed [9, 19, 36, 40]. A key difference of our work is to additionally consider the effort that is saved, instead of merely the effort that is spent, as we have seen in §3.2. Meanwhile, DevRank can be extended to use a more advanced effort estimation scheme (e.g., polynomial, churn). The framework and methodologies of our work are orthogonal and remain applicable.

6 CONCLUSION

There are commercial, pedagogical, and stewardship reasons to evaluate the value of individual code contributions to a large code base. 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 non-structural value, and proposed a combination of a PageRank-inspired algorithm and an impact coding scheme through manual labels or a machine learning model trained from developer's artifacts. We hope this on-going research work will finally enable and support an even stronger ecosystem of contribution-based projects with a "long tail" of contributors as well as give better insights on the relative strengths of contributors and code.

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] Stefan Biffl, Aybuke Aurum, Barry Boehm, Hakan Erdogmus, and Paul Grünbacher (Eds.). 2009. *Value-Based Software Engineering*. Springer.

[4] 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

[5] 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

[6] Jeff Bonwick. 1994. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (USTC'94)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1267257.1267263

[7] 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

[8] 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

[9] Bradford Clark, Sunita Devnani-Chulani, and Barry Boehm. 1998. Calibrating the COCOMO II Post-architecture Model. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*. IEEE Computer Society, 477–480. http://dl.acm.org/citation.cfm?id=302163.302218

[10] 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

[11] 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

[12] 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

[13] GitHub. 2018. Viewing contribution activity in a repository. https://help.github.com/articles/viewing-contribution-activity-in-a-repository/.

[14] Barney G Glaser, Anselm L Strauss, and Elizabeth Strutzel. 1968. The discovery of grounded theory; strategies for qualitative research. *Nursing research* 17, 4 (1968), 364.

[15] David F. Gleich. 2015. PageRank Beyond the Web. *SIAM Rev.* 57, 3 (Aug. 2015), 321âĂŞ363.

[16] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. 2000. Large margin rank boundaries for ordinal regression. 88 (01 2000).

[17] ISO/IEC. 2006. ISO/IEC 14764:2006 Software Engineering – Software Life Cycle Processes – Maintenance. https://www.iso.org/standard/39064.html.

[18] 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

[19] 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

[20] 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

[21] Brian W. Kernighan. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference.

[22] 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.

[23] 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

[24] 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.

[25] 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

[26] 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

[27] Matt Mackall. 2005. slob: introduce the SLOB allocator. https://lwn.net/Articles/157944/

[28] 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

[29] 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

[30] 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

[31] 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

[32] Ivan Mistrik, Rami Bahsoon, Rick Kazman, and Yuanyuan Zhang (Eds.). 2014. *Economics-Driven Software Architecture*. Morgan Kaufmann.

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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, 85âĂŞ–89.

[38] 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

[39] 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

[40] 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

[41] 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