Frontiers in Computing and Intelligent Systems
ISSN: 2832-6024 | Vol. 14, No. 2, 2025

Code Commit Health (CCH): An Effective and Accessible Metric for Quantifying Development Code Contributions

Shuang Chen [1,*], Nannan Diao [2]

[1] School of Shanghai Advanced Institute of Finance, Shanghai Jiao Tong University, Shanghai 200030 China
[2] School of Educational Information Technology, Central China Normal University, Wuhan Hubei, 430079 China

[*] **Corresponding author**: Shuang Chen (Email: casparchen@alumni.sjtu.edu.cn)

Abstract: The utilisation of Lines of Code (LOC) or Number of Commits (NOC) is an inadequate representation of the actual contributions of programmers. Furthermore, evaluation systems based on complex and difficult-to-obtain information, such as code structures, are inherently limited to small-scale applications and cannot be applied on a large scale. In order to enhance the precision of assessment while preserving simplicity, this paper puts forth a novel evaluation metric: Code Commit Health (CCH) is a new metric that combines two existing metrics: the number of lines of code and the number of commits. It is designed to provide a more comprehensive evaluation of programmers' code contributions by employing a reasonable formula. In the actual study, the validity of CCH was verified through an in-depth analysis of more than 300,000 commit records from ten renowned open-source projects and an empirical study of the commit data of 110 developers in an enterprise. The findings of the study demonstrate that CCH is an effective means of reflecting the actual contributions of programmers and has good applicability. It is our intention to provide a practical method for quantifying programmers' contributions for the open source community, enterprises and academia, and to promote the science and rationalisation of code contribution evaluation.

Keywords: Code Commit Health; Quantifying Development; Code of Value; Code Contribution Assessment.

1. Introduction

At present, the majority of companies and organisations continue to employ two single metrics, namely lines of code (LOC) or the number of code commits (NOC), to assess the contributions of programmers. For example, GitHub, the largest open-source community in the world, employs the number of commits as a means of calculating programmer contribution rankings[1]. In contrast, OpenHarmony applies the number of lines of code as a metric for this purpose[2]. However, these two single metrics are inadequate for accurately reflecting the actual contributions of programmers. Furthermore, the use of a single metric can easily result in the phenomenon of "brushing metrics". For instance, an assessment based solely on the number of lines of code may result in the proliferation of superfluous code, whereby programmers may utilize a greater number of lines of code to implement a given function that could be achieved with a smaller amount of code. The mere enumeration of commits is insufficient for identifying those that are devoid of meaningful content. For instance, a single commit comprising a single line of configuration, accompanied by a multitude of other commits, may be regarded as a trivial contribution. This phenomenon is particularly prevalent within the GitHub community, which applies a commit count to determine the relative value of individual contributions. Furthermore, scholars have proposed sophisticated measurement algorithms based on the structure of the code[3], which are undoubtedly more effective than single metrics. However, the threshold for obtaining these complex metrics is high, and different programming languages have different structures. There are currently dozens of popular programming languages[4], which has resulted in the majority of companies and organisations retaining the most rudimentary single metrics.

In order to quantify and evaluate programmers' contributions in a more reasonable manner, this paper employs two metrics, NOC and LOC, and develops a set of formulas to calculate Code Commit Health (CCH). CCH circumvents the various shortcomings associated with a single metrics and offers a more comprehensive assessment of programmers' contributions. The evaluation of a programmer's contribution can be conducted in a more effective manner, and the data necessary for CCH is readily accessible, thereby enhancing the applicability of CCH. For instance, CCH is employed for the assessment of contributors in open-source communities, the evaluation of employee performance within enterprises, and the scoring of student contributions in programming teaching activities.

2. Research Method

The methodology of this study is comprised of the following principal stages. The initial stage of the process entails the extraction of commit records from the code repository. The following step is the cleaning of the data using the isolated forest algorithm, with the objective of removing abnormal commits. Subsequently, the programmers are grouped together, and the total number of commits and the average number of lines committed are calculated for each programmer. Ultimately, the aggregated data is subjected to modelling and calculation in order to ascertain the final code commit health.

2.1. Cleaning Abnormal Commit Data

A review of data from open source projects and internal enterprise commit records revealed a significant imbalance in the distribution of lines of code per commit, accompanied by a considerable degree of variance. Some commits alter a

single line of code, whereas others modify thousands or even tens of thousands of lines of code. This phenomenon is particularly prevalent in certain front-end projects. For instance, programmers may be required to copy or integrate third-party JavaScript files, resulting in the submission of a substantial number of third-party code contributions that are not typically considered part of the original contribution. Given the considerable discrepancy in the number of lines of code submitted on each occasion and the uneven distribution, it is not feasible to employ straightforward statistical techniques to assess the outliers. Instead, machine learning methodologies must be utilised to validate the outliers. In this paper, we adopt the isolated deep forest algorithm[5] as the calibration method for outliers. The Isolated Deep Forest algorithm randomly cuts the data space by constructing multiple isolated trees to achieve the detection of outliers. This randomness makes the isolated forest insensitive to the distribution of the data, thereby ensuring strong robustness and greater applicability to the research scenario described in this paper. Four prominent open-source projects were selected for experimental investigation: Vue3[6], Dubbo[7], Kafka[8] and Transformers[9]. These projects span four distinct domains: front-end, back-end, middleware and machine learning. Figure 1 illustrates the distribution of single-commit code lines for the four projects. It is evident that the distribution of single-commit code lines is highly uneven, and the Isolated Deep Forest algorithm is capable of distinguishing outliers with greater efficacy.

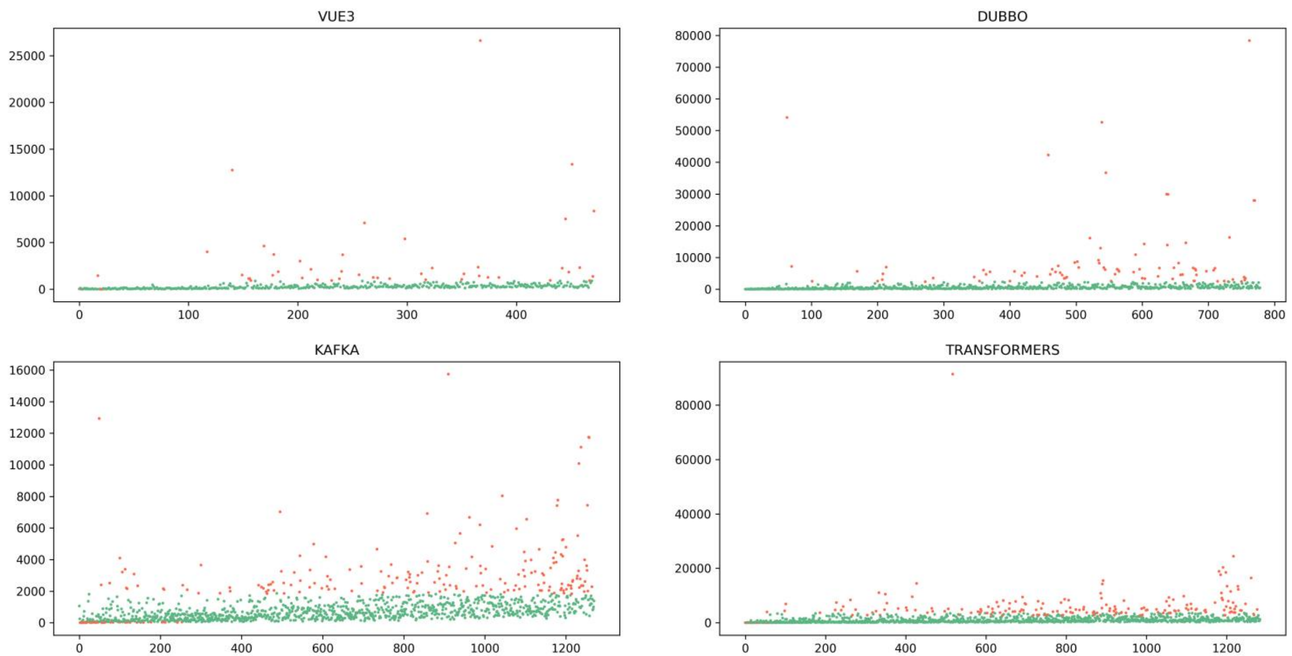

Figure 1. Isolation forest anomaly detection

2.2. Establishing the Relative Importance of LOC and NOC

Once the two data sets of LOC and NOC have been obtained, it is necessary to confirm the relative importance of these two metrics in order to inform the subsequent formula design. A questionnaire survey was conducted on 36 programmers, with the design of the questionnaire based on the consistency matrix method in AHP hierarchical analysis[10]. This entailed a comparison of the two metrics on a two-by-two basis, with the use of a relative scale to enhance the precision of the evaluation. In this section, the relative importance of the NOC and LOC metrics is determined by comparing them two by two. In particular, the question posed was as follows: which of the two metrics do you consider to be more important in measuring programmer contribution in comparison to NOC and LOC? A total of five options were presented. The responses indicated that LOC is significantly important, LOC is slightly important, LOC is equally important, NOC is slightly important, and NOC is significantly important. These responses were represented by scores of 2, 1, 0, -1, and -2, respectively. Upon analysis of the collected results, it was found that the majority of respondents agreed that NOC was relatively more important than LOC, with a mean value of 1.03. The detailed distribution is shown in Figure 2.

Figure 2. Relative importance scores of LOC and NOC

A higher number of commits typically indicates that the developer is actively engaged in the project and contributes more to its development. This is a common practice in globally renowned open-source communities such as GitHub, Gitee, and OpenStack, where NOC is primarily utilized to document the developer's activity[1,11,12]. Furthermore, there is a correlation between NOC and code quality. It can be posited that an increase in NOC may be accompanied by an increase in the quality of the code. However, it must be acknowledged that NOC, in isolation, is not a comprehensive indicator of code quality. Rather, it should be considered alongside other, more complex metrics[13].

3. Benchmarking the LOC

While the number of commits is a more important metrics of developer contribution than LOC, it is not the sole factor that should be considered when assessing contribution. It is erroneous to assume that a high frequency of commits necessarily implies an actual high level of contribution, particularly in the presence of commit count swiping behaviours. It is therefore essential to assess the reasonableness of the number of lines of code submitted. In order to guarantee the fairness and accuracy of the assessment, it is necessary to clean the data in order to remove any anomalous commit records, such as those with a very low or very high number of lines. These outliers can be caused by incorrect commits, duplicate commits, or artificially swiped commits. The cleaned commits provide a more accurate reflection of the actual workload of the developer.

In order to ascertain a reasonable number of lines of code, the mean value of the cleaned commit records can be employed as a benchmark. By comparing each developer's commit lines to this mean, it is possible to determine which commits are more reasonable and score higher; conversely, commits that deviate more from the mean are considered unreasonable and score lower. This approach effectively balances the frequency of commits and the amount of code, and reduces the incidence of developers increasing their scores through unnatural means.

3.1. Formula Design

3.1.1. Calculation of Contribution Coefficient

In light of the findings from the preceding steps, it becomes evident that the NOC holds greater significance than the LOC. In the formula design, it is essential to consider the significance of the NOC, at the same time, to assess the reasonableness of the LOC. This implies that the closer the value is to the mean value of the cleaned commit records, the more favourable it is. In order to achieve a more uniform distribution of contribution values, a logarithmic function is employed to address the phenomenon of logarithmic growth, which exhibits a gradual deceleration in the rate of expansion as the number of submissions increases. The rate of the score will be slower in order to balance the score gap between those who have a very high number of submissions and those who have fewer submissions. Furthermore, this approach will avoid a very small number of high-frequency commits from scoring too highly. Concurrently, to circumvent the potential issue of zero, logarithmic processing is conducted by uniformly appending a value of one to the processed quantity. Subsequent to this addition, programmers with zero submissions are assigned a score of zero, rather than an undefined value, while programmers with lesser submissions are allotted a lower, yet non-zero, score. Moreover, programmers with a greater number of submissions witness a progressively slower growth rate in their scores, which serves to maintain equilibrium across the distribution. The design formula is as follows:

$$A_i = \frac{\ln_{(c_i+1)}}{\ln_{(|\bar{x}_i - \bar{x}|+1)}} \tag{1}$$

A_i denotes the contribution coefficient of the i-th individual, C_i denotes the number of commits made by the i-th individual, $\bar{x}_i$ denotes the average number of lines commit by the i-th individual, $\bar{x}$ denotes overall average number of lines committed after data cleaning.

3.1.2. Standardisation Contribution Coefficient

In order to facilitate comparison of contribution coefficients, it is necessary to standardise them. Standardisation can eliminate the influence of scale in the raw data, improving the performance and accuracy of subsequent data processing [15,16]. In the raw scores, a discrepancy in the performance of individuals with markedly disparate scores may impede subsequent comparative analysis. Standardisation entails the conversion of all scores to standard scores on a uniform scale, thereby facilitating direct comparison.

The standardisation of data entails a shift in the centre of the distribution, with the mean value set to zero and the variance set to one. This facilitates the subsequent mathematical operations, particularly when utilising sigmoid functions, as it allows for a more straightforward mapping of the standardised data to the desired range.

The standardised contribution coefficient can be calculated by means of a mean and standard deviation calculation for the contribution coefficient.

$$z_i = \frac{A_i - \mu}{\sigma} \tag{2}$$

z_i denotes the standardised contribution coefficient of the i-th individual, A_i denotes the contribution coefficient of the i-th individual, μ denotes the overall average number of contribution coefficient, σ denotes the standard deviation of overall contribution coefficient.

3.1.3. Normalisation

In order to enhance the intelligibility and accessibility of the score results, the normalised contribution coefficients were normalised using the sigmoid function. The sigmoid function is a widely employed normalisation method that is capable of mapping an arbitrary range of values to a range from 0 to 1, thereby facilitating the comparison of scores [17] Subsequently, the result is multiplied by 100 and displayed as a percentage. This treatment not only renders the score more readily interpretable, but also more accurately reflects the developer's contribution in an intuitive manner. The normalised formula is as follows:

$$S_i = \frac{1}{1 + e^{-z_i}} \times 100 \tag{3}$$

S_i denotes the contribution score of the i-th individual, e: The natural constant is approximately equal to 2.71828.

3.1.4. Combining Formulas

In the preceding sections, the significance of the number of commits and the reasonableness of the number of lines of code committed to the contribution assessment were discussed, and corresponding formulas were proposed to quantify the impact of these metrics. In order to obtain a comprehensive measure of the developer's contribution, the three aforementioned equations are merged to create a single Contribution Score Calculation formula. This final formula is derived from the combination of Equation 1 (Calculation Contribution Coefficient), Equation 2 (Standardisation Contribution Coefficients), and Equation 3 (Normalisation Contribution Coefficients).

$$S_i = \frac{1}{1 + e^{-\left(\frac{\frac{\ln(c_i+1)}{\ln(|\bar{x}_i - \bar{x}|+1)} - \mu}{\sigma}\right)}} \times 100 \quad (4)$$

The final Contribution Score formula incorporates each of these metrics in order to guarantee a more comprehensive and equitable assessment of developer contributions. The formula is based on the following implications. (1) The NOC serves as a significant metrics of contribution assessment. The greater the number of commits, the greater the investment of time and effort by the developer in the project. Consequently, the contribution should be commensurately higher. The formula therefore includes consideration of the NOC, and through the introduction of the logarithmic function, the effect of the NOC on the contribution level is gradually smoothed in the case of high-frequency commits, thus avoiding the problem of a few high-frequency commits scoring too highly. (2) It is also important to consider the reasonableness of the LOC. The presence of an excess or deficiency of lines in a single commit may be indicative of underlying issues pertaining to the quality of the code or the irrationality of the developer's commit behaviour. Consequently, a factor has been incorporated into the formula to assess the reasonableness of the average number of submitted lines, which is adjusted in accordance with the proximity of the average LOC of developers to the average LOC of the overall cleaned project. In this context, a closer alignment with the average results in a higher score, whereas a deviation from the average yields a lower score.

The final Contribution Score formula considers not only the number of commits made by a developer, but also the reasonableness of the number of lines of code committed. By combining these metrics, a scoring system is obtained that comprehensively reflects the actual contribution of the developer. This formula encourages developers to make more high-quality commits within a reasonable range, thus increasing their contribution to the project.

4. Application Examples

In accordance with the CCH score calculation formula proposed in Section 2.4, this section will verify its application through two actual cases. The first case study will examine the application of the CCH formula in the context of a prominent open-source software project, illustrating how it can be used to assess the contribution of open-source contributors. Subsequently, we will examine the application of the CCH formula within an enterprise context, illustrating the tangible outcomes of the formula's implementation in programmer performance assessment and contribution measurement through an actual enterprise case study, thereby substantiating its applicability and efficacy in diverse settings.

4.1. Application in Open Source Projects

To illustrate, we examined the case of Dubbo, a renowned micro-services open-source software developed under the Apache Software Foundation. Our analysis encompassed the entire history of the Dubbo project, from its inception on June 20, 2012, to July 27, 2024. During this period, a total of 535 programmers contributed to the project's development, collectively making 5,803 code commits. In the initial processing of the commit records, 779 records were de-duplicated based on the number of lines of code committed. Subsequently, the data underwent cleaning using the Isolated Deep Forest algorithm, resulting in the retention of 695 records. The data distribution is illustrated in Figure 3.

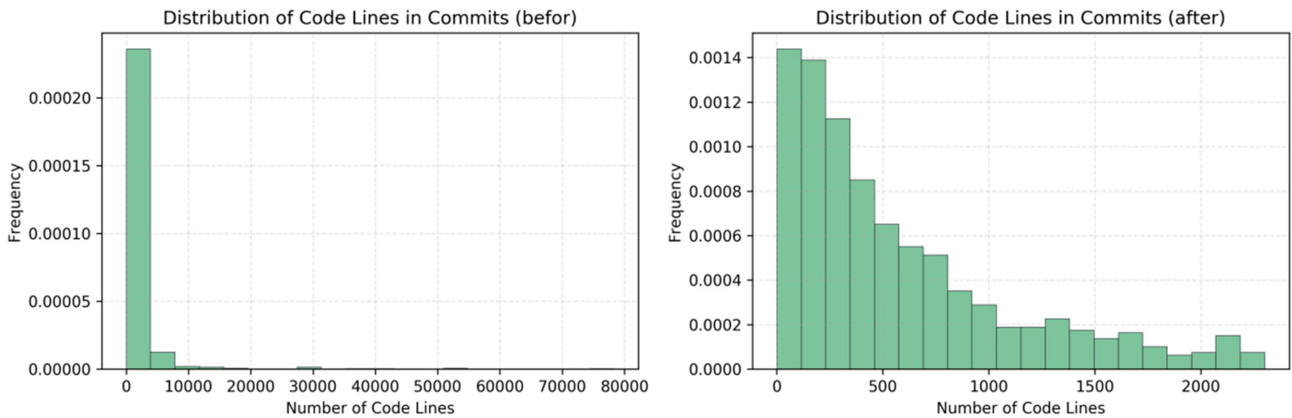

Figure 3. Distribution of line of code before and after cleaning using the isolated deep forest algorithm

Figure 3 illustrates the distribution of lines from a single code commit prior to cleaning. It is evident that the data is highly uneven, exhibiting significant outliers. The right part of the figure depicts the data distribution following cleaning with the isolated deep forest algorithm. It is notable that the data distribution is more uniform and fluctuates more steadily after cleaning.

A comprehensive summary of the descriptive data statistics is presented in Table 1.

Table 1. Data descriptive statistics for the number of lines of code commits

	Min	Mean	Media	Max	Standard Deviation
Origin	1	1599	474	78332	5083
Cleaned	1	578	397	2301	538

The commitment records from Dubbo were incorporated into the CCH contribution score formula to determine the top 10 rankings, as illustrated in Table 2.

Table 2. The top 10 contributors to the dubbo project, as calculated by CCH

Rank CCH	Rank GitHub	Author	Count	Mean	Sum	*Ai*	CCH
1	1	chickenlj	1058	560	592766	2.319	99.998
2	6	CrazyHZM	147	507	74542	1.163	99.344
3	2	AlbumenJ	923	204	188051	1.151	99.301
4	5	cvictory	149	435	64879	1.007	98.518
5	3	dependabot[bot]	593	1	880	1.004	98.497
6	4	beiwei30	264	163	42938	0.925	97.735
7	8	mercyblitz	108	763	82386	0.897	97.396
8	13	kylixs	70	451	31577	0.877	97.105
9	7	liangfei0201	128	312	39919	0.869	96.988
10	9	oldratlee	104	352	36577	0.857	96.794

As can be observed from the table, there are some discrepancies between the rankings produced by CCH and those derived from GitHub. The first position remains unchanged, with chickenlj retaining the distinction of having the most commits and an average number of commit lines that is very close to the overall average of 578. Additionally, chickenlj is the primary contributor to the dubbo project. The table indicates a rise of at least two places by the CCH calculation in the column marked in red, and a fall of at least two places in the column marked in green. It is noteworthy that CrazyHZM has experienced a significant shift in rank, advancing from 6th to 2nd place. This is primarily attributable to the fact that, despite ranking 6th in terms of the number of commits, his average number of lines of code commits is only slightly below the overall average of 578. Consequently, he has been assigned a higher ranking. The Dubbo project, like the majority of open source projects, adheres to the open source contribution convention and has a defined specification for the number of commit messages [18]. A further analysis of the detailed commit logs of AlbumenJ and CrazyHZM revealed that commits beginning with the words 'Merge', 'Bump' or 'Prepare' are for the maintenance of version numbers. It can be observed that these behaviours contribute significantly less to the project than the actual code writing. A total of 228 out of 923 commits for AlbumenJ are for versioning, representing 24.70% of the total number of commits. In comparison, CrazyHZM has only 5 versioning commits, accounting for only 3.40% of the total number of commits. This also corroborates the assertion that CrazyHZM has a higher average contribution per commit. Furthermore, the dependabot[bot] account has fallen from third to fifth place. Despite having a considerable number of commits (593), which is way ahead of the original fourth-place account (264), its average number of lines of code committed is only one, which is considerably lower than the average. It is evident that this account is responsible for versioning the various third-party packages. Despite committing code on numerous occasions, the majority of these commits entail merely changing the version number, and thus, the value of the contribution is not particularly noteworthy.

The Dubbo project demonstrates that the CCH contribution score formula is a valid and fair method of evaluation in practical applications. Furthermore, the same validation process has been applied to ten projects in different languages and domains, with similarly positive results. This approach can effectively reduce the ranking to a level comparable to that of the dependabot [bot] account, providing a reasonable representation of actual contribution. Refer to Table 3 for a detailed overview of these and other open-source projects.

Table 3. List of open-source projects performing CCH experiments

Project	Language	Tag	Stars	Contributors	Commits
Vue3	TypeScript	front-end	46.2K	477	5896
echarts	TypeScript	front-end	59.9K	231	9375
dubbo	Java	back-end	40.2K	560	5803
kafka	Scala	middleware	28.1K	1185	13487
spark	Scala	big data	39.1K	2110	41859
paimon	Java	big data	2.2K	173	2962
transformers	Python	machine learning	130K	2664	16587
tensorflow	C++	machine learning	185K	3581	168228
redis	C	database	65.9K	717	12213
tidb	Go	database	36.7K	871	24354

4.2. Application in Enterprise

CCH can be applied not only to the contribution ranking calculation of open source project personnel, but also as a reference basis for the assessment of code commits of programmers within the enterprise. In order to verify the applicability of CCH in the enterprise environment, the CCH algorithm was applied to Company A. The code commits records of Company A in the second quarter of 2024 were analyzed by CCH calculation.

4.2.1. Data Preparation and Tagging

All code commit records of Company A for the second quarter of 2024 were extracted, comprising 20950 code commits contributed by 110 developers. In order to evaluate

the effectiveness of the CCH algorithm, it is first necessary to prepare the tagged dataset. A total of 20 basic leaders were asked to mark the team members they led based on the actual performance of the team members' code in the second quarter of 2024. This performance was evaluated based on a number of criteria, including the amount of tasks completed, task difficulty, and quality of the software interactions. The leaders were asked to perform a binary classification marking, with a value of 1 indicating high contribution and a value of 0 indicating low contribution. The actual marking result is that 76 individuals were identified as having made a high contribution, representing 69.09% of the total, while 34 individuals were identified as having made a low contribution, representing 30.91% of the total. This demonstrates that the distribution of contribution value is unbalanced, with a greater number of individuals identified as having made a high contribution than a low contribution. This aligns with the anticipated usage scenario of ROC and AUC [19,20].

4.2.2. Validity Analysis

The tagged real contribution data were employed to calculate and analyse the ROC curves and AUC values for the three evaluation metrics, namely CCH, NOC and LOC. To visualise the classification performance of each metric, the corresponding ROC curves were plotted, as illustrated in Figure 4. The curves facilitate a more precise comparison of the accuracy and effectiveness of the different metrics in evaluating code contribution.

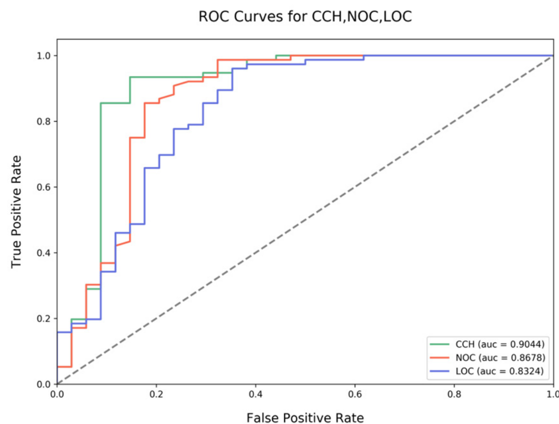

Figure 4. ROC curves for CCH, NOC and LOC

The results of this experiment demonstrated that all three metrics exhibited a satisfactory level of classification performance. The area under the curve (AUC) value for LOC is 0.8324, the AUC value for NOC is 0.8678, and the AUC value for CCH is 0.9044. All AUC values exceed 0.8, indicating that even these single metrics are effective in reflecting the majority of programmers' contributions within an organisation. However, the CCH metric performs particularly well, with an AUC value exceeding 0.9, indicating that CCH is capable of more accurately differentiating between high and low programmer code contribution.

By calculating the true positive rate (TPR) and the false positive rate (FPR), we were able to conduct a more detailed analysis of the accuracy of each metric in distinguishing between high and low contributing developers. The TPR indicates the proportion of all actual high contributing developers that were correctly identified, while the FPR indicates the proportion of actual low contributing developers that were misclassified as high contributors. The TPR and FPR values for each metric are presented in Table 4.

Table 4. Confusion matrix for the three metrics

	TPR	FPR	AUC
CCH	0.6842	0.0882	0.9044
NOC	0.6579	0.1471	0.8678
LOC	0.6447	0.1765	0.8324

The experimental results demonstrate that the True Positive Ratio (TPR) of all three metrics is less than 0.7, indicating that when identifying high contributors, these metrics exhibit discrepancies with the actual contributions, and may miss identifying those developers with higher actual contributions. In the context of enterprise operations, while the number of commits and lines of code contributed by these high-performing developers may not be substantial, the complexity of the tasks they undertake is often elevated, as are the intrinsic values of the submitted code. The inability to assess these factors exclusively through commit records represents a major limitation of CCH. Nevertheless, CCH continues to demonstrate superior performance in comparison to NOC and LOC. It is noteworthy that CCH exhibits a false positive rate (FPR) of 0.0882, which indicates that it is highly accurate in identifying low contributors.

In conclusion, the CCH algorithm performs well in enterprise internal applications, not only in identifying high and low contributors, but also in terms of overall AUC performance in comparison to NOC and LOC. CCH enhances the precision and impartiality of the assessment process, offering a more robust foundation for enterprise-level developer contribution evaluation.

5. Conclusion

This paper proposes a new method for assessing developer contributions, designated CCH (Code Commit Healthiness). The objective is to evaluate developers' actual contributions to a project in a more simplistic and reasonable manner, taking into account the reasonableness of the number of commits and the average number of lines of commits, and combining it with data cleaning techniques. The efficacy and viability of the CCH algorithm are validated through case studies on open-source projects and internal applications of Company A.

In the present study, we conducted an analysis of 10 prominent open-source projects, encompassing a total of over 300,000 commit records. The results demonstrate that the CCH algorithm not only transcends the constraints of assessment techniques that rely exclusively on the number of commits or lines of code, but also effectively identifies developers who have made significant contributions to the project. In comparison to the original ranking produced by GitHub, the CCH algorithm provides a more reasonable and fair ranking, particularly effective in the context of bot accounts and high-frequency commits with low actual contributions.

In an internal application at Company A, the CCH algorithm was found to demonstrate a clear advantage in reflecting the actual work contribution of programmers, with a significantly higher accuracy rate compared to traditional LOC and NOC evaluation methods, following the analysis of 110 developers and 20,950 commit records. These findings

demonstrate that the CCH algorithm is not only applicable in open source projects, but also provides a robust reference for enterprises in developer performance evaluation.

The implementation of the CCH algorithm in open-source projects and enterprise environments serves to corroborate its intrinsic soundness. Furthermore, the source of CCH's metrics is relatively straightforward to obtain and is not constrained by the specific development language. As more data and cases are accumulated, it is anticipated that the CCH algorithm will undergo further optimisation and promotion, thereby providing a more scientific and reasonable solution for the evaluation of developer contributions.

References

[1] Viewing a project's contributors, GitHub, 2024, [Online]. Available: https://docs.github.com/en/repositories/viewing-activity-and-data-for-your-repository/viewing-a-projects-contributors. 2024.

[2] OpenHarmony main warehouse code contribution metrics rules, OpenHarmony, 2024, [Online]. Available: https://metrics. openharmony. Cn.

[3] Jinglei Ren, Hezheng Yin, Qingda Hu, Armando Fox, and Wojciech Koszek. 2018. Towards quantifying the development value of code contributions. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 775–779. https://doi.org/ 10. 1145/3236024.3264842.

[4] Tiobe. 2024. TIOBE Index for August 2024. https://www. tiobe.com/tiobe-index/.

[5] Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. Eighth IEEE International Conference on Data Mining, Pisa, Italy, 2008, pp. 413-422, doi: 10.1109/ ICDM. 2008. 17.

[6] GitHub. 2024. Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web. https:// github. com/vuejs/core/graphs/contributors.

[7] GitHub. 2024. Mirror of Apache Kafka. https://github. com/ apache/kafka.

[8] GitHub. 2024. The java implementation of Apache Dubbo. An RPC and microservice framework. https://github. com/ apache/ dubbo.

[9] GitHub. 2024. Transformers: State-of-the-art Machine Learning for Pytorch, TensorFlow, and JAX. https://github. com/ huggingface/transformers.

[10] Saaty T L .Decision making with the analytic hierarchy process. Int J Serv Sci[J].International Journal of Services Sciences, 2008, 1(1):83-98.

[11] Gitee. 2024. User Influence. https://portrait. gitee. com/ help/ articles/ 4383.

[12] Stackalytics. 2024. OpenStack community contribution in Caracal release. https://www.stackalytics.io.

[13] Bird, C., Nagappan, N., Gall, H., Murphy, B., & Devanbu, P. 2009. Does distributed development affect software quality? An empirical case study of Windows Vista.

[14] Tukey, J.W. 1962. The Future of Data Analysis. Annals of Mathematical Statistics, 33, 1-67.

[15] Ruppert, D. 2004. The elements of statistical learning: data mining, inference, and prediction.

[16] Ioffe, S., & Szegedy, C. 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift. JMLR.org.

[17] Han, J., & Moraga, C. 1995. The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning. International Work-Conference on Artificial and Natural Neural Networks.

[18] GitHub. 2024. Contribution to dubbo. https://github. com/ apache/dubbo/blob/3.2/CONTRIBUTING.md#code-conventions.

[19] Fawcett, T. 2006. An introduction to ROC analysis. Pattern Recognit. Lett., 27, 861-874.

[20] Bradley, A.P. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit., 30, 1145-1159.