Noname manuscript No.
(will be inserted by the editor)

Corrective Commit Probability: A Measure of the Effort Invested in Bug Fixing

Idan Amit[1,2] · **Dror G. Feitelson**[1]

the date of receipt and acceptance should be inserted later

Abstract The effort invested in software development should ideally be devoted to the implementation of new features. But some of the effort is invariably also invested in corrective maintenance, that is in fixing bugs. Not much is known about what fraction of software development work is devoted to bug fixing, and what factors affect this fraction. We suggest the Corrective Commit Probability (CCP), which measures the probability that a commit reflects corrective maintenance, as an estimate of the relative effort invested in fixing bugs. We identify corrective commits by applying a linguistic model to the commit messages, achieving an accuracy of 93%, higher than any previously reported model. We compute the CCP of all large active GitHub projects (7,557 projects with 200+ commits in 2019). This leads to the creation of an investment scale, suggesting that the bottom 10% of projects spend less than 6% of their total effort on bug fixing, while the top 10% of projects spend at least 39% of their effort on bug fixing — more than 6 times more. Being a process metric, CCP is conditionally independent of source code metrics, enabling their evaluation and investigation. Analysis of project attributes shows that lower CCP (that is, lower relative investment in bug fixing) is associated with smaller files, lower coupling, use of languages like JavaScript and C# as opposed to PHP and C++, fewer code smells, lower project age, better perceived quality, fewer developers, lower developer churn, better onboarding, and better productivity.

Keywords Corrective maintenance · Corrective commits · Effort estimate · Process metric

I. Amit **(corresponding author)**
idan.amit@mail.huji.ac.il

D. G. Feitelson
feit@cs.huji.ac.il

[1] Department of Computer Science
The Hebrew University of Jerusalem, Jerusalem 91904, Israel

[2] Acumen Labs
Tel Aviv, Israel

1 Introduction

Software quality is an important area in software engineering [13, 33, 113, 128, 129]. But despite decades of work on this issue, there is no agreed definition of "software quality". For some, this term refers to the quality of the software product as perceived by its users [51,121]. Others use the term in reference to the code itself, as perceived by developers, for example as reflected by code smells [44,74,131,133,142]. Non-functional properties of code (e.g. reliability, modifiability) were also used as quality measures [26]. Others include correctness as the foremost property [41]. Our approach is also that correctness is an important element of quality, and our focus is on the costs of removing defects.

The goal of software development is to create software that provides services and features to its users. But part of the development effort is actually invested in fixing bugs that occurred in the software development process [86]. In fact, corrective maintenance (aka fixing bugs) may represent a large fraction of software development, and may contribute significantly to software costs [27, 87, 120]. It is time consuming, disrupts schedules, and hurts the reputation of software products. Moreover, it is generally accepted that fixing bugs costs more the later they are found, and that maintenance is costlier than initial development [24, 25, 27, 39, 51]. The effort invested in bug fixing therefore reflects on the health of the development process. If you are constantly putting out fires, instead of making progress according to plan, your project might be in trouble.

If one were to just count fixed bugs, the comparison between a single-developer one week project and a 20-years 100-developers project would be misleading. Hence, normalization is needed. The natural normalization is by the total effort invested in the project. Ideally, the vast majority of the investment should go to development, and as little as possible on fixing bugs [86]. The literature contains vastly varying reports regarding the relative fraction of corrective maintenance, from 17.4% to 56.7% on average [80, 87, 120]. Thus it is desirable to collect more data on this issue.

Based on these considerations, we suggest the Corrective Commit Probability (CCP, the probability that a given commit is a bug fix) as an estimate of the relative effort invested in fixing bugs. Having such data can improve our understanding of the factors that affect the basic attributes of software development, and specifically the division of effort between fixing previous problems and making progress with new features. CCP can also serve as a metric for the health of a project and its code. Note that we do not claim this is THE metric for project health. Project health is a multifaceted concept, yet a metric does not have to cover all possible considerations in order to be useful. We see it as a software engineering equivalent of blood pressure measurement. Many medical conditions are not captured by blood pressure. Nevertheless, it can be applied easily at scale, and helps in investigating the nature of the patient's situation.

There are three main factors that may be expected to affect CCP:

- The difficulty of the domain and the application's complexity (Section 8.1). The more complex the task, the more bugs that can be expected to be found, leading to a higher CCP. This is unavoidable when a project undertakes a very challenging task.
- The bug detection efficiency (Section 6). Higher efficiency leads to the detection of more bugs and a higher CCP. Thus a high CCP may reflect a conscientious organization which invests in code quality.

– The quality of the code (Section 7). High coupling (Section 7.3), and file length (Section 7.1) are indications of low code quality and indicate higher CCP.

Due to these different contributions, it is overly simplistic to label projects with a high CCP as potentially in trouble. However, we can study the factors that contribute to the CCP. For example, we find that 'security' projects tend to have a higher CCP than user interface oriented projects. This is interpreted as reflecting the difficulty of the problem: a project that is intrinsically hard leads to the creation and fixing of more bugs.

We identify corrective commits using a linguistic model applied to commit messages, an idea that is commonly used for defect prediction [37, 52, 113]. The linguistic-model prediction marks commit messages as corrective or not, in the spirit of Ratner et al. labelling functions [112]. Though such predictions are not completely accurate and therefore the model hits do not always coincide with the true corrective commits, our accuracy is significantly higher than previous work [4, 6, 7, 60, 83].

Given an implementation of the CCP metric, we perform a large-scale assessment of GitHub projects. We analyze all 7,557 large active projects (defined to be those with 200+ commits in 2019, excluding redundant projects which might bias our results [22]). We use this, inter alia, to build the distribution of CCP, and find the ranking of each project relative to all others. The results indicate that significant differences occur between the CCP of different projects: those in the top 10% invest more than 6 times as much in bug fixing as those in the bottom 10%. Software developers can easily know their own project's CCP. They can thus find where their project is ranked with respect to the community.

Note that CCP provides a retrospective assessment of a project's state. It only applies *after bugs are found*, unlike code metrics which can be applied as the code is written. The CCP metric can be used as a research tool for the study of different software engineering issues. A simple approach is to observe the CCP given a certain phenomenon (e.g., programming language, coupling). For example, we show below that CCP appears to be inversely correlated with developer productivity: The average productivity is higher in low CCP projects.

Our **main contributions** in this research are as follows:

– We define the Corrective Commit Probability (CCP) metric for the relative investment in bug fixing. The metric is easy to compute, indifferent to programming language, and is applicable at all granularities.
– We develop a linguistic model to identify corrective commits that performs significantly better than prior work and is close to human level.
– We show how to perform a maximum likelihood computation to improve the accuracy of the CCP estimation, also removing the dependency on the implementation of the linguistic model.
– We establish a scale of CCP across projects, which provides a calibration for practitioners who can compare their effort on bug fixing with the industry. The scale shows that projects in the top decile spend at least six times the effort on bug correction as projects in the bottom decile.
– We show that CCP correlates with various other effects, e.g. successful onboarding of new developers and productivity. This solidifies the scientific basis for software engineering, specifically the understanding of factors that shift division of effort towards bug fixing.

– We present twin experiments and co-change analysis in order to investigate relations beyond mere correlation.

The paper is structured as following: We present the related work (Section2), explain the construction of CCP (Section 3), and in Section 4 presents our experimental methodology. The distribution of CCP is presented in Section 5, and the effect of detection efficiency on CCP in Section 6. Section 7 covers the interaction of CCP with coding practices, and Section 8 the relation with the project profile. We then check to which extent the discussed factors cover CCP in Section 9. We end with threats to validity in Section 10 and conclusions in Section 11.

2 Related work

The number of bugs in a program could have been a great quality metric. However, Rice's theorem [116] tells us that bug identification, like any non-trivial semantic property of programs, is undecidable. Nevertheless, bugs are being found, providing the basis for the CCP metric.

Capers Jones defined software quality as the combination of low defect rate and high user satisfaction [65, 67]. He went on to provide extensive state-of-the-industry surveys based on defect rates and their correlation with various development practices, using a database of many thousands of industry projects. Our work applies these concepts to the world of GitHub and open source, using their accessibility to investigate defect rates' possible causes and implications.

Software metrics can be divided into three groups: product metrics, code metrics, and process metrics. Product metrics consider the software as a black box. A typical example is the ISO/IEC 25010:2011 standard [63]. It includes metrics like fitness for purpose, satisfaction, freedom from risk, etc. These metrics might be subjective, hard to measure, and not applicable to white box actionable insights, which makes them less suitable for our research goals. Indeed, studies of the ISO/IEC 9126 standard [62] (the precursor of ISO/IEC 25010) found it to be ineffective in identifying design problems [1].

Code metrics measure properties of the source code directly. Typical metrics are lines of code (LOC) [88], the Chidamber and Kemerer object oriented metrics (aka CK metrics) [31], McCabe's cyclomatic complexity [92], Halstead complexity measures [53], etc. [14, 50, 109]. They tend to be specific, low level and highly correlated with LOC [47, 94, 118, 123]. Some specific bugs can be detected by matching patterns in the code [61]. But this is not a general solution, since depending on it would bias our data towards these patterns.

Process metrics focus on the code's evolution. The main data source is the source control system. Typical metrics are the number of commits, the commit size, the number of contributors, etc. [49, 96, 109]. Process metrics have been claimed to be better predictors of defects than code metrics for reasons like showing where effort is being invested and having less stagnation [96, 109].

Working with commits as the entities of interest is also popular in just in time (JIT) defect prediction [70]. Unlike JIT, we are interested in the probability and not in a specific commit being corrective. We also focus on analyzing periods of a whole year, rather than comparing the versions before and after a bug fix, which probably reflects an improvement. Examining results in consecutive years we show that CCP is stable, so projects that are prone to errors stay so, despite prior efforts to fix bugs.

Focusing on commits, we need a way to know if they are corrective. If one has access to both a source control system and a ticket management system, one can link the commits to the tickets [20] and reduce the CCP computation to mere counting. Yet, the links between commits and tickets might be biased [20]. The ticket classification itself might have 30% errors [58], and may not necessarily fit the researcher's desired taxonomy. And integrating tickets with the code management system might require a lot of effort, making it infeasible when analysing thousands of projects. Moreover, in a research setting the ticket management system might be unavailable, so one is forced to rely on only the source control system.

When labels are not available, one can use linguistic analysis of the commit messages as a replacement. This is often done in defect prediction, where supervised learning can be used to derive models based on a labeled training set [37, 52, 113].

In principle, commit analysis models can be used to estimate the CCP, by creating a model and counting hits. That could have worked if the model accuracy was perfect. We take the model predictions and use the hit rate, the probability that the classifier will label a commit as corrective, and the model confusion matrix to derive a maximum likelihood estimate of the CCP. Without such an adaptation, the analysis might be invalid, and the hits of different models would have been incomparable.

Our work is also close to Software Reliability Growth Models (SRGM) [49,139,141]. In SRGM one tries to predict the number of future failures, based on bugs discovered so far, and assuming the code base is fixed. The difference between us is that we are not aiming to predict future quality. We identify current software quality improvement in order to investigate its causes and implications.

The number of bugs was used as a feature and indicator of quality before as absolute number [73, 115], per period [135], and per commit [4, 124]. We prefer the per commit version since it is agnostic to size and useful as a probability.

3 Definition and Computation of the Corrective Commit Probability

We now describe how we build the mechanism to estimate the Corrective Commit Probability, in three steps:

1. Sect. 3.1: Constructing a gold standard data set of labeled commit samples, identifying those that are corrective (bug fixes). These are later used to learn about corrective commits and to evaluate the model.
2. Sect. 3.2: Building and evaluating a supervised learning linguistic model to classify commits as either corrective or not. Applying the model to a project yields a hit rate for that project.
3. Sect. 3.3: Using maximum likelihood estimation in order to find the most likely CCP given a certain hit rate.

The need for the third step arises because the hit rate may be biased, which might falsify further analysis like using regression and hypothesis testing. By working with the CCP maximum likelihood estimation we become independent of the model details and its hit rate. We can then compare the results with earlier versions of the model, or even with results based on other researchers' models. We can also identify outliers deviating from the common linguistic behavior (e.g., non-English projects), and remove them to prevent erroneous analysis.

Note that we are interested in the *overall probability* that a commit is corrective. This is different from defect prediction, where the goal is to determine whether a specific commit is corrective. Finding the probability is easier than making detailed predictions. In analogy to coin tosses, we are interested only in establishing to what degree a coin is biased, rather than trying to predict a sequence of tosses. Thus, if for example false positives and false negatives are balanced, the estimated probability will be accurate even if there are many wrong predictions.

3.1 Building a Gold Standard Data Set

The most straightforward way to compute the CCP is to use a change log system for the commits and a ticket system for the commit classification [20], and compute the corrective ratio. However, for many projects the ticket system is not available. Therefore, we base the commit classification on linguistic analysis, which is built and evaluated using a gold standard.

A gold standard is a set of entities with labels that capture a given concept. In our case, the entities are commits, the concept is corrective maintenance [130], namely bug fixes, and the labels identify which commits are corrective. Gold standards are used in machine learning for building models, which are functions that map entities to concepts. By comparing the true label to the model's prediction, one can estimate the model performance. In addition, we also used the gold standard in order to understand the data behavior and to identify upper bounds on performance.

We constructed the gold standard as follows. Google's BigQuery has a schema for GitHub[1] were all projects' commits are stored in a single table. We sampled uniformly 840 (40 duplicate) commits as a train set. The first author then manually labeled these commits as being corrective or not based on the commit content using a defined protocol. The full protocol is included in the supplementary materials. In brief, fixes to documentation, style, typos, etc. are not considered to be bug fixes. Tangled commits, commits serving several goals [57, 59], are considered to be bug fixes even if they also include a refactor or introduce new code. Tests are considered to be a part of the system and its requirements. Therefore, a bug fix in the tests is a bug fix.

To assess the subjectiveness in the labeling process, two students were recruited to independently label 400 of the commits. When there was no consensus, we checked if the reason was a deviation from the protocol or an error in the labeling (e.g., missing an important phrase). In these cases, the annotator fixed the label. Otherwise, we considered the case as a disagreement and its final label was a majority vote of the annotators. The Cohen's kappa scores [32] among the different annotators were at least 0.9, indicating excellent agreement. Similarly consistent commit labeling was reported by Levin and Yehudai [83].

Of the 400 triple-annotated commits, there was consensus regarding the labels in 383 (95%) of them: 105 (27%) were corrective, 278 were not. There were only 17 cases of disagreement. An example of disagreement is "correct the name of the Pascal Stangs library." It is subjective whether a wrong name is a bug.

In addition, we also noted the degree of certainty in the labeling. The message "mysql_upgrade should look for .sql script also in share/ directory" is clear, yet it is

[1] https://console.cloud.google.com/bigquery?d=github_repos&p=bigquery-public-data&page=dataset

unclear whether the commit is a new feature or a bug fix. In only 7 cases the annotators were uncertain and could not determine with high confidence the label from the commit message and content. Of these, in 4 they all nevertheless selected the same label.

Two of the samples (0.5%) were not in English. This prevents English linguistic models from producing a meaningful classification.

Finally, in 4 cases (1%) the commit message did not contain any syntactic evidence for being corrective. The most amusing example was "When I copy-adapted handle_level_irq I skipped note_interrupt because I considered it unimportant. If I had understood its importance I would have saved myself some ours of debugging" (the typo is in the origin). Such cases set an upper bound on the performance of any syntactic model. In our data set, all the above special cases (uncertainty, disagreement, and lack of syntactic evidence) are rather rare (just 22 samples, 5.5%, since many behaviors overlap), and the majority of samples are well behaved. The number of samples in each misbehavior category is very small so ratios are very sensitive to noise.

3.2 Syntactic Identification of Corrective Commits

Our linguistic model is a supervised learning model, based on indicative terms that help identify corrective commit messages. Such models are built empirically by analyzing corrective commit messages in distinction from other commit messages.

Many prior language models suggest short lists made up of obvious terms like 'bug', 'bugfix', 'error', 'fail', 'fix' [55, 113]. Such a list reached 88% accuracy on our data set. A commonly suggested alternative approach today is to employ machine learning. We tried many machine learning classification algorithms and only the plain decision tree algorithm reached such accuracy. More importantly, as presented later, we are not optimizing for accuracy.

The main reason for the limited performance of the machine learning classification algorithms was that we are using a relatively small labeled data set, and linguistic analysis tends to lead to many features (e.g., in a bag of words, word embedding, or n-grams representation). In such a scenario, models might overfit and be less robust [56]. One might try to cope with overfitting by using models of low capacity. However, the concept that we would like to represent (e.g., include "fix" and "error" but not "error code" and "not a bug") is of relatively high capacity. The need to cover many independent textual indications and count them requires a large capacity, larger than what can be supported by our small labeled data set. We therefore elected to construct the model manually based on several sources of candidate terms and the application of semantic understanding. Note that though we did not use classification algorithms, the goal, the structure, and the usage of the model are of supervised learning.

We began with a private project in which the commits could be associated with a ticket-handling system that enabled determining whether they were corrective. We used them in order to differentiate the word distribution of corrective commit messages and other messages and find an initial set of indicative terms. In addition, we used the gold-standard data-set presented above. This data set is particularly important because our target is to analyze GitHub projects, so it is desirable that our train data will represent the data on which the model will run. This train data set helped tuning the indicators by identifying new indications and nuances and alerting to bugs in the model implementation.

To further improve the model we used some terms suggested by Ray et al. [113] (variants of 'deadlock', 'race condition', 'memory leak', 'buffer overflow', 'heap overflow', 'missing switch case', 'faulty initialization', 'segmentation fault', and 'double free'), tough we did not adopt all of them (e.g., we do not consider a typo to be a bug). This model was used in Amit and Feitelson [4], reaching an accuracy of 89%. We then added additional terms from Shrikanth and Menzies [125] (variants of 'proper', 'broke', 'vulnerab', and 'defect'). We also used labeled commits from Levin and Yehudai [83] to further improve the model based on samples it failed to classify.

The last boost to performance came from the use of active learning [122] and specifically the use of classifiers discrepancies [5]. Once the model's performance is high, the probability of finding a false negative, $positive\_rate \cdot (1 - recall)$, is quite low, requiring a large number of manually labeled random samples per false negative. Amit and Feitleson [4] provided models for a commit being corrective, perfective, or adaptive. A commit not labeled by any of the models is assured to be a false negative (of one of them). Sampling from this distribution was an effective method to find false negatives, and improving the model to handle them increased the model recall from 69% to 84%. Similarly, while a commit might be both corrective and adaptive, commits marked by more than one classifier are more likely to be false positives. Using them improved the precision from 84% to 87%.

The resulting model uses regular expressions to identify the presence of different indicator terms in commit messages. We base the model on straightforward regular expressions because this is the tool supported by Google's BigQuery relational database of GitHub data, which is our target platform.

The final model is based on three distinct regular expressions. The first identifies about 50 terms that serve as indications of a bug fix. Typical examples are: "bug", "failure", and "correct this". The second identifies terms that indicate other fixes, which are not bug fixes. Typical examples are: "fixed indentation" and "error message". The third is terms indicating negation. This is used in conjunction with the first regular expression to specifically handle cases in which the fix indication appears in a negative context, as in "this is not an error". It is important to note that fix hits are also hits of the other fixes and the negation. Therefore, the complete model counts the indications for a bug fix (matches to the first regular expression) and subtracts the indications for not really being a bug fix (matches to the other two regular expressions). If the result is positive, the commit message was considered to be a bug fix. The results of the model evaluation using a 1,100 samples test set built in Amit and Feitelson [4] are presented in the confusion matrix of Table 1.

Table 1 Confusion matrix of model on test data set.

	Classification	
Concept	True(Corrective)	False
True	228 (20.73%) TP	43 (3.91 %) FN
False	34 (3.09%) FP	795 (72.27%) TN

Supervised learning metrics shed light on the common behaviour of a classifier and a concept. The cases in which the concept is true are called 'positives'(P) and the positives rate is denoted $P(positive)$. Similarly, 'negatives' (N) are samples for which the concept is false. Cases in which the classifier is true are called 'hits' and the hit

rate is $P(hit)$. For example, in a classifier of defect prediction, a high hit rate means that the developer will have to examine many files.

Ideally, hits correspond to positives but usually they differ. Precision, defined as $P(positive|hit)$, measures a classifier's tendency to avoid false positives (FP). Precision might be high simply since the positive rate is high. Precision lift, defined as $Precision/P(positive) - 1 = \frac{P(positive|hit) - P(positive)}{P(positive)}$, copes with this difficulty and measures the additional probability of having a true positive relative to the base positive rate. Thus a useless random classifier will have precision equal to the positive rate, but a precision lift of 0.

Recall, defined as $P(hit|positive)$, measures how many of the positives are also hits; in our case, this is how many of the fixes the classifier identifies. Accuracy, $P(positive = hit)$ is probably the most common supervised learning metric. False Positive Rate, Fpr, is $\frac{FP}{N}$. We show in Section 3.3 how we can use the hit rate and the classifier's recall and Fpr in order to better estimate the positive rate.

The confusion matrix of Table 1 contains all the data needed to calculate these supervised learning metrics:

- Positive rate (real corrective commit rate): 24.6%
- Accuracy (model is correct): 93.0%
- Precision (ratio of hits that are indeed positives): 87.0%
- Precision lift ($\frac{precision}{positive\ rate} - 1$): 253.2%
- Hit rate (ratio of commits identified by model as corrective): 23.8%
- Recall (positives that were also hits): 84.1%
- Fpr (False Positive Rate, negatives that are hits by mistake): 4.2%

Though prior work was based on different protocols and data sets and therefore hard to compare, our accuracy is significantly better than prior results of 68% [60], 70% [6], 76% [83] and 82% [7], and also better than our own previous result of 89% [4]. The achieved accuracy is close to the well-behaving commits ratio in the gold standard.

3.3 Maximum Likelihood Estimation of the Corrective Commit Probability

Let hr be the hit rate (probability that the model will identify a commit as corrective) and pr be the positive rate, the true corrective rate in the commits (this is what CCP estimates). In prior work it was all too common to use the hit rate directly as the estimate for the positive rate. However, since model predictions are not perfect, the hit rate and positive rate may differ. By considering the model performance *we can derive a better estimate* of the positive rate given the hit rate. From a performance modeling point of view, the Dawid-Skene [38] modeling is an ancestor of our work. But note that the Dawid-Skene framework represents a model by its precision and recall, and we use Fpr and recall.

There are two distinct cases that can lead to a hit. The first is a true positive (TP): There is indeed a bug fix and our model identifies it correctly. The probability of this case is $\Pr(TP) = pr \cdot recall$. The second case is a false positive (FP): There was no bug fix, yet our model mistakenly identifies the commit as corrective. The probability of this case is $\Pr(FP) = (1 - pr) \cdot Fpr$. Adding them gives

$$hr = \Pr(TP) + \Pr(FP) = (recall - Fpr)pr + Fpr \tag{1}$$

Extracting pr leads to

$$pr = \frac{hr - Fpr}{recall - Fpr} \tag{2}$$

We want to estimate $\Pr(pr|hr)$. Let n be the number of commits in our data set, and k the number of hits. As the number of samples increases, $\frac{k}{n}$ converges to the model hit rate hr. Therefore, we estimate $\Pr(pr|n,k)$. We will use maximum likelihood for the estimation. The idea behind maximum likelihood estimation is to find the value of pr that maximizes the probability of getting a hit rate of hr.

Note that if we were given p, a single trial success probability, we could calculate the probability of getting k hits out of n trails using the binomial distribution formula

$$\Pr(k;n,p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{3}$$

Finding the optimum requires the computation of the derivative and finding where it equals to zero. The maximum of the binomial distribution is at $\frac{k}{n}$. Equation (2) is linear and therefore monotone. Therefore, the maximum likelihood estimation of the formula is

$$pr = \frac{\frac{k}{n} - Fpr}{recall - Fpr} \tag{4}$$

For our model, $Fpr = 0.042$ and $recall = 0.84$ are fixed constants (rounded values taken from the confusion matrix of Table 1). Therefore, we can obtain the most likely pr given hr by

$$pr = \frac{hr - 0.042}{0.84 - 0.042} = 1.253 \cdot hr - 0.053 \tag{5}$$

3.4 Validation of the CCP Maximum Likelihood Estimation

George Box said: "All models are wrong but some are useful" [28]. We would like to see how close the maximum likelihood CCP estimations are to the actual results. Note that the model performance results we presented in Table 1, using the gold standard test set, do not refer to the maximum likelihood CCP estimation. We need a new independent validation set to verify the maximum likelihood estimation. Therefore, we uniformly sampled another set of 400 commits and the first author manually labeled them. We are interested in the estimation of two parameters, recall and Fpr. While 30 samples are considered to provide reasonable sample size for one parameter, our sample size is larger, improving the estimation. The model performance is presented in a confusion matrix shown in Table 2.

Table 2 Confusion matrix of model on validation data set.

	Classification	
Concept	True(Corrective)	False
True	91 (22.75%) TP	18 (4.5%) FN
False	34 (8.5%) FP	257 (64.25%) TN

In this data set the positive rate is 27.2%, the hit rate is 31.2%, the recall is 83.5%, and the Fpr is 11.7%. Note that the positive rate in the validation set is 2.6 percentage

points different from our test set. The positive rate has nothing to do with MLE and shows that statistics tend to differ on different samples. In this section we would like to show that the MLE method is robust to such changes.

In order to evaluate how sensitive the maximum likelihood estimation is to changes in the data, we used the bootstrap method [42]. We sampled with replacement 400 items from the validation set, repeating the process 10,000 times. Each time we computed the true corrective commit rate, the estimated CCP, and their difference. Figure 1 shows the distribution of these differences.

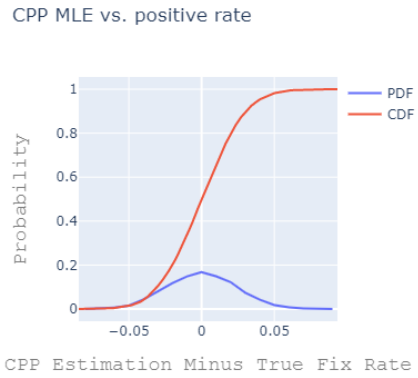

Fig. 1 Difference distribution in validation bootstrap.

In order to cover 95% of the distribution we can trim the 2.5% tails from both sides. This will leave us with differences ranging between -0.044 to 0.046, so the estimated CCP has a confidence interval of ±4.5 percentage points. One can use the boundaries related to 95%, 90%, etc. in order to be extra cautious in the definition of the valid domain.

Another possible source of noise is in the model performance estimation. If the model is very sensitive to the test data, a few anomalous samples can lead to a bad estimation. Again, we used bootstrap in order to estimate the sensitivity of the model performance estimation. For 10,000 times we sampled *two* data sets of size 400. On each of the data sets we computed the recall and Fpr and built an MLE estimator. We then compared the difference in the model estimation at a few points of interest: [0,1] – the boundaries of probabilities, [0.042, 0.84] – the boundaries of the valid domain, and [0.06, 0.39] – the p_{10} and p_{90} percentiles of the CCP distribution. Since our models are linear, so are their differences. Hence their maximum points are at the ends of the examined segments. When considering the boundaries of probabilities [0,1], the maximal absolute difference is 0.34 and 95% of the differences are lower than 0.19. When considering the boundaries of the valid domain [0.042, 0.84], the maximal absolute difference is 0.28 and 95% of the differences are lower than 0.15. When considering the p_{10} and p_{90} percentiles of the CCP distribution [0.06, 0.39], the maximal absolute difference is 0.13 and 95% of the differences are lower than 0.07.

Using the validation set estimator on the test set, the CCP is 0.168, 7.7 percentage points off the actual positive rate. In the other direction, using the CCP estimator

test data performance on the validation set, the CCP is 0.39, 11.8 points off. Since our classifier has high accuracy, the difference between the hit rate and the CCP estimates in the distribution deciles, presented below in Table 3, is at most 4 percentage points. Hence the main practical contribution of the MLE in this specific case is the identification of the valid domain (Section 3.5) rather than an improvement in the estimate.

3.5 Sensitivity to the Fixed Linguistic Model Assumption

The maximum likelihood estimation of the CCP assumes that the linguistic model performance, measured by its *recall* and Fpr, is fixed. Hence, a change in the hit rate in a given domain is due to a change in the CCP in the domain, and not due to a change in the linguistic model performance.

Yet, this assumption does not always hold. Both hr and pr are probabilities and must be in the range $[0, 1]$. Equation (2) equals 0 at $hr = Fpr$ and 1 at $hr = recall$. For our model, this indicates that the range of values of hr for which pr will be a probability is $[0.042, 0.84]$. Beyond this range, we are assured that the linguistic model performance is not as measured on the gold standard. An illustrative example of the necessity of the range is a model with $recall = 0.5$ and $Fpr = 0$. Given $hr = 0.9$ the most likely pr is 1.8. This is an impossible value for a probability, so we deduce that our assumption is wrong.

As described in Section 4.1, we initially estimated the CCP of all 8,588 large active projects in 2019. In 1,031 of them the CCP estimate was invalid, leaving us with a set of 7,557 projects to study.

In 10 of the invalid projects the estimated CCP was above 1. Checking these projects, we found that they have many false positives, e.g. due to a convention of using the term "bug" for general tasks, or starting the subject with "fixes #123" where ticket #123 was not a bug fix but some other task id.

The bulk of the invalid projects (11.8% of the original set) had an estimated CCP below 0. This could indicate having extremely few bugs, or else a relatively high fraction of false negatives (bug fixes we did not identify). One possible reason for low identification is if the project commit messages are not in English. To check this, we built a simple linguistic model in order to identify if a commit message is in English. The model was the 100 most frequent words in English longer than two letters (see details and performance in supplementary materials). The projects with negative CCP had a median English hit rate 0.16. For comparison, the median English hit rate of the projects with positive CCP was 0.54, and 96% of them had a hit rate above 0.16.

Interestingly, another reason for many false negatives was the habit of using very terse messages. We sampled 5,000 commits from the negative CCP projects and compared them to the triple-annotated data set used above. In the negative CCP commits, the median message length was only 27 characters, and the 90th percentile was 81 characters. In the annotated data set the median was 8 times longer, and the 90th percentile was 9 times longer.

It is also known that not all projects in GitHub (called there repositories) are software projects [69,97]. Since bugs are a software concept, other projects are unlikely to have such commits and their CCP will be negative. Hence, the filtering also helps us to focus on software projects. Git is unable to identify the language of 6% of the projects with negative CCP, more than 14 times the ratio in the valid domain. The

languages 'HTML', 'TeX', 'TSQL', 'Makefile', 'Vim script', 'Rich Text Format' and 'CSS' are identified for 22% of the projects with negative CCP, more than 4 times as in the valid range. Many projects involve few languages and when we examined a sample of projects we found that the language identification is not perfect. However, at least 28% of the projects that we filtered due to negative CCP are not identified by GitHub as regular software projects.

Pull requests and issue management are typical of software projects. Therefore another check is to see how many of the projects in the negative domains use them. We used the GHTorrent [48] BigQuery schema that collects pull requests and issues of GitHub projects. We wanted to examine whether projects in the valid domain tend to use pull requests more than those in the negative domain. The result was that none of the negative domain projects existed in the GHTorrent schema. 68% of the projects in the valid domain appeared there, and 75% of them used pull requests. The absence of all the negative domain projects is another indication of not being a typical software project.

To summarize, in the projects with invalid CCP estimates, below 0 or above 1, the behavior of the linguistic model changes and invalidates the fixed performance assumption. We believe that the analysis of projects in the CCP valid domain is suitable for software engineering goals. The CCP distribution in Table 3 is presented for both the entire data set and only for projects with valid CCP estimates. The rest of the analysis is done only on the valid projects.

3.6 CCP as a Quality Metric

There is no debate that bugs are bad, especially bugs reported by customers [51]. Moreover, assessing quality based on bugs is a process metric and therefore conditionally independent [23, 85] from code metrics. In particular, that makes them conditionally independent from the programming language, file length, code smells, etc. This can be applied at various resolutions, e.g., a project, a file, or a method, and help spot entities that are bug prone, improving future bug identification [76, 110, 136].

It is therefore interesting to check to what degree CCP can be used as a quality metric. As noted above, quality is *one* of the factors that affect CCP: low quality code is expected to have more bugs. But whether these bugs are actually found depends also on other factors, most obviously the bug detection efficiency [65, 67]. So the actual relation between CCP and low quality is a-priori uncertain.

A simple approach is to check references to low quality in commit messages and correlated them with CCP (Figs. 2 and 3). The specific terms checked were direct references to "low quality", and related terms like "code smell" [44, 74, 131, 133, 142] and "technical debt" [35, 79, 132]. In addition, swearing is also a common way to express dissatisfaction, with more than 200 thousand occurrences compared to only hundreds or thousands for the technical terms. The probability of a commit containing swearing to be reverted are 69% higher than the average. In the same spirit, Romano et al. showed that negative sentiment increases the probability of a commit to be bug introducing [117].

The commit meta-data contains the files involved in it. This enables us to aggregate the commits *per file* and compute the file's CCP — the ratio of corrective commits out of the commits that modified the file. We considered files with 10 or more commits, in

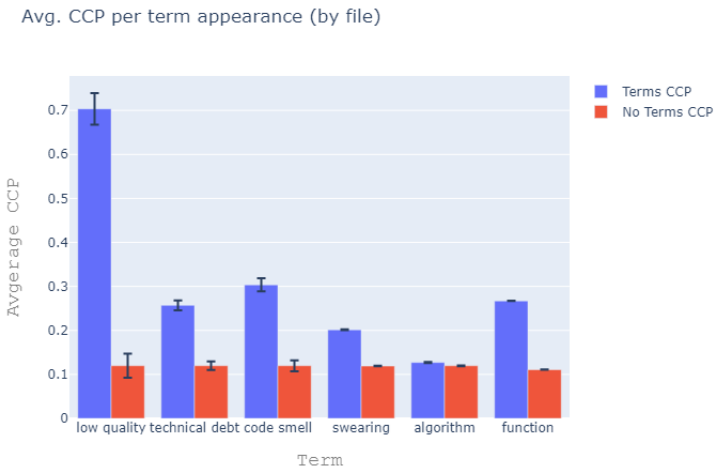

Fig. 2 CCP of files with or without different quality terms.

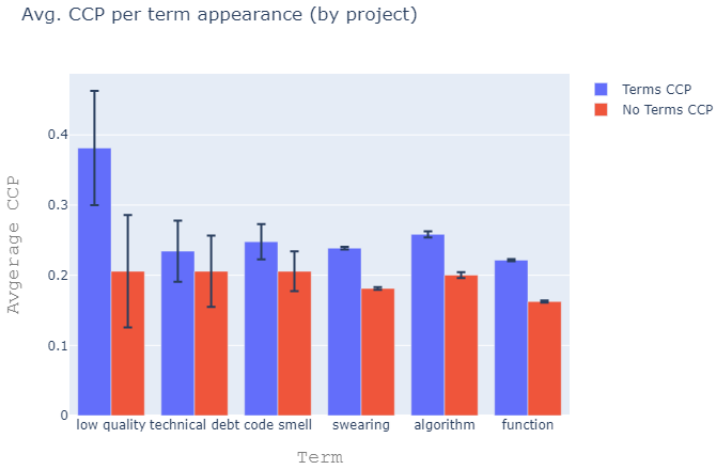

Fig. 3 CCP of projects with or without different quality terms.

order to be more robust to noise. We compared those with at least 10% occurrences of the term to the rest.

Projects may have a lot of commits, and the vast majority do not contain the terms. Instead of a ratio, we therefore consider a project to contain a term if it has at least 10 commits in which the term occurs. As the figures show, when the terms appear, the CCP is higher (sometimes many times higher), especially for the explicit term "low quality". Thus the high-CCP metric agrees with the opinions of the projects' developers regarding quality.

To verify this result, we attempted to use negative controls. A negative control is an item that should be indifferent to the analysis. In our case, it should be a term not related to quality. We chose "algorithm" and "function" as such terms. The verification worked for "algorithm" at the file level: files with and without this term had practically the same CCP. But files with "function" had a much higher CCP than files without it, and projects with both terms had a higher CCP than without them. Possible reasons are some relation to quality (e.g., algorithmic oriented projects are harder) or biases (e.g., object-oriented languages tend to use the term "method" rather than "function"). Anyway, it is clear that the difference in "low quality" is much larger and there are some differences in the other terms too. Note that this investigation is not completely independent. While the quality terms used here are different from those used for the classification of corrective commits, we still use the same data source.

We further explored the relation using co-change analysis between swearing and CCP (the other terms are too rare; co-change analysis is explained in Sect. 4.2). The Pearson correlation of swearing rate over adjacent years is 0.74. The agreement of co-change is 54% for any change and 88% when requiring a significant change (0.1 for CCP, 0.01 for the rarer swearing). These results remain when we control for programming language, project age, or number of developers.

Another way that developers express dissatisfaction is "Self Admitted Technical Debt" [106]. We used the terms suggested by Rantala et al. [111] and measured the relative CCP of files containing these terms compared to all files. Files containing 'TODO' had average CCP 69% higher, 'FIXME' 116% higher, 'HACK' 28% higher, and 'XXX' 42% higher.

4 Experimental Methodology

Given the ability to estimate the CCP of any project given its development history, we can now investigate the relationship between CCP and various project attributes. Results are computed on GitHub projects active in 2019, selected according to the procedure outlined in Sect. 4.1, and specifically on projects whose CCP is in the valid domain. We did not work with version releases since we work with thousands of projects whose releases are not clearly marked. Note that in projects doing continuous development, the concept of release is no longer applicable.

Our results are in the form of correlations between CCP and such attributes. For example, we show that projects with shorter files tend to have a lower CCP. These correlations are informative and actionable, e.g., enabling a developer to focus on longer files during testing and refactoring. But correlation is not causation, so we cannot say conclusively that longer files *cause* a higher propensity for bugs that need to be fixed. Showing causality requires experiments in which we perform the change, which we leave for future work. The correlations that we find indicate that a search for causality might be fruitful and could motivate changes in development practices that may lead to improved software quality.

In order to make the results stronger than mere correlation, we use several methods in the analysis. We use co-change over time analysis in order to see to what extent a change in one metric is related to a change in the other metric (Sect. 4.2). Given factors A, B, and C, we control the results by making comparisons for fixed C to see that the relation between A and B is not due to C. We control project age, programming languages , number of developers, and detection efficiency as explained in Section 4.4.

We also control for the developer, by observing the behaviour of the same developer in different projects (Section 4.3). This allows us to separate the influence of the developer and the project.

The distributions we examined tended to have some outliers that are much higher than the mean and the majority of the samples. Including outliers in the analysis might distort the results. In order to reduce the destabilizing effect of outliers, we applied Winsorizing [54]. We used one-sided Winsorizing, where all values above a certain threshold are set to this threshold. We do this for the top 1% of the results throughout, to avoid the need to identify outliers and define a rule for adjusting the threshold for each specific case. In the rest of the paper we used the term capping (a common synonym) for this action. In addition, we check whether the metrics are stable across years. A reliable metric applied to clean data is expected to provide similar results in successive years.

4.1 Selection of Projects

In 2018 GitHub published that they had 100 million projects[2]. The BigQuery GitHub schema contains about 2.5 million *public* projects prior to 2020. But the vast majority are not appropriate for studies of software engineering, being small, non-recent, or not even code.

In order to omit inactive or small projects where estimation might be noisy, we defined our scope to be all open source projects included in GitHub's BigQuery data with 200+ commits in 2019. We selected a threshold of 200 to have enough data per project, yet have enough projects above the threshold. There are 14,749 such projects (Fig. 4).

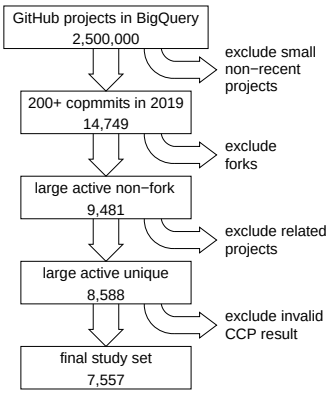

Fig. 4 Process for selecting projects for analysis.

However, this set is redundant in the sense that some projects are closely related [69]. The first step to reduce redundancy is to exclude projects marked in the GitHub API as being forks of other projects. This reduced the number to 9,481 projects. Sometimes extensive amounts of code are cloned without actual forking. Such code

[2] https://github.blog/2018-11-08-100m-repos/

cloning is prevalent and might impact analysis [2, 45, 89]. Using commits to identify relationships [93], we excluded dominated projects, defined to have more than 50 common commits with another, larger project, in 2019. Last, we identified projects sharing the same name (e.g., 'spark') and preferred those that belonged to the user with more projects (e.g., 'apache'). After the redundant projects removal, we were left with 8,588 projects. But calculating the CCP on some of these led to invalid values as described above in Section 3.5. For analysis purposes we therefore consider only projects where CCP is in the valid range, whose number is 7,557.

A possible additional filter is to exclude projects identified by the topic "student-project" on GitHub. However, this turned out to be redundant, as the previous steps already filtered all such projects.

4.2 Co-change Over Time

While experiments can help to determine causality, they are based on few cases and expensive. On the other hand, we have access to plenty of observations, in which we can identify correlations. While causal relations tend to lead to correlation, non-causal relations might also lead to correlations due to various reasons. We would like to use an analysis that will help to filter out non-causal relations. By that we will be left with a smaller set of more likely relations to be further investigated for causality. In this and the next subsection we present two methods to identify situations that are likely to be causal.

When two metrics change simultaneously, it is less likely to be accidental. Hence, we track the metrics over time in order to see how their changes match. We create pairs of the same project in two consecutive years. For each pair we check whether both the first and second metrics improved. The ratio of improvement match (the equivalent to accuracy in supervised learning) is an indication of related changes. Denote the event that metric i improved from one year to the next by $m_i\uparrow$. The probability $P(m_j\uparrow | m_i\uparrow)$, (the equivalent to precision in supervised learning), indicates how likely we are to observe an improvement in metric j knowing of an improvement in metric i. It might be that we will observe high precision but it will be simply since $P(m_j\uparrow)$ is high. In order to exclude this possibility, we also observe the precision lift, $\frac{P(m_j\uparrow \,|\, m_i\uparrow)}{P(m_j\uparrow)} - 1$. Note that lift cannot be used to identify the causality direction since it is symmetric:

$$\frac{P(m_j\uparrow \,|\, m_i\uparrow)}{P(m_j\uparrow)} = \frac{P(m_i\uparrow \,\wedge\, m_j\uparrow)}{P(m_i\uparrow) \cdot P(m_j\uparrow)} = \frac{P(m_i\uparrow \,|\, m_j\uparrow)}{P(m_i\uparrow)} \tag{6}$$

If an improvement in metric i indeed causes the improvement in metric j, we expect high precision and lift. Since small changes might be accidental, we also investigate improvements above a certain threshold. There is a trade-off here since given a high threshold the improvement is clearer, yet the number of cases we consider is smaller. Another trade-off comes from how far in the past we track the co-changes. The earlier we will go the more data we will have. On the other hand, this will increase the weight of old projects, and might subject the analysis to changes in software development practices over time and to data quality problems. We chose a scope of 5 years, avoiding looking before 2014.

4.3 Controlling the Developer

Measured metric results (e.g., development speed, low coupling) might be due to the developers working on the project (e.g., skill, motivation) or due to the project environment (e.g., processes, technical debt). To separate the influence of developers and environment, we checked the performance of developers active in more than one project in our data set. By fixing a single developer and comparing the developer's activity in different projects, we can investigate the influence of the project. Note that a developer active in n projects will generate $O(n^2)$ project pairs ("twins") to compare.

Consider development speed as an example. If high speed is due to the project environment, in high speed projects every developer is expected to be faster than himself in other projects. This control resembles twin experiments, popular in psychology, where a behavior of interest is observed on twins. Since twins have a very close genetic background, a difference in their behavior is more likely to be due to another factor (e.g., being raised in different families).

Assume that performance on project A is in general better than on project B. We consider developers that contributed to both projects, and check how often they are better in project A than themselves in project B (formally, the probability that a developer is better in project A than in project B given that project A is better than project B). This is equivalent to precision in supervised learning, where the project improvement is the classifier and the developer improvement is the concept. In some cases, a small difference might be accidental. Therefore we require a large difference between the projects and between the developer performance (e.g., at least 10 commits per year difference, or more formally, the probability that a developer committed at least 10 times more in project A than in project B given that the average number of commits per developer in project A is at least 10 commits higher than in project B).

We considered only involved developers, which we define as those committing at least 12 times per year (at least one commit per month on average), otherwise the results might be misleading. While this omits 62% of the developers, they are responsible for only 6% of the commits. This also correlates with the probability to continue to contribute to the project in the next year. The probability of developers contributing less than 12 commits to continue with the project is 0.22, while the probability of an involved developer to continue is 0.73. Developers contributing exactly 12 commits are balanced with a probability of 0.53 to continue.

4.4 Controlling variables

We might see a relation between two variables that is actually due to a third confounding variable, influencing both of them. For example, a relation between high quality and high productivity might be due to the use of a more suitable language. The statistical method to avoid this is to control the confounding variable. Instead of examining the relation between quality and productivity in general, we will also examine if the relation holds separately for Java projects, C++ projects, etc.

When considering the control variables we first check their own relation with CCP. Later, we also control it as part of the analysis of the relation of other variables and CCP.

The control variables that we use are number of developers (Section 8.2), programming languages (Section 8.3), project age (Section 8.4), and detection efficiency (Section

6). Though influential, we don't control the project domain, for reasons explained in Section 8.1.

5 The Distribution of CCP

Given the ability to identify corrective commits, we can classify the commits of each project and estimate the distribution of CCP over the projects' population.

Table 3 CCP distribution in active GitHub projects.

	Full data set (8,588 projects)		CCP $\in [0, 1]$ (7,557 projects)	
Percentile	Hit rate	CCP est.	Hit rate	CCP est.
10	0.34	0.38	0.35	0.39
20	0.28	0.30	0.29	0.32
30	0.24	0.25	0.26	0.27
40	0.21	0.21	0.22	0.23
50	0.18	0.18	0.20	0.20
60	0.15	0.14	0.17	0.17
70	0.12	0.10	0.15	0.13
80	0.09	0.06	0.12	0.10
90	0.03	-0.02	0.09	0.06
95	0.00	-0.05	0.07	0.04

Table 3 shows the distribution of hit rates and CCP estimates on the GitHub projects with 200+ commits in 2019, with redundant repositories (representing the same project) excluded. The hit rate represents the fraction of commits identified as corrective by the linguistic model, and the CCP is the maximum likelihood estimation. The lowest 10% of projects have a CCP of up to 0.06. The median project has a CCP of 0.2, more than three times the lowest projects' CCP. Interestingly, Lientz at el. reported a median of 0.17 in 1978, based on a survey of 69 projects [87]. The highest 10% have a CCP of 0.39 or more, more than 6 times higher than the lowest 10%. This shows that there is a wide spectrum of levels of investment in bug fixing, from just a few percents to more than a third of the total effort (as quantified by number of commits).

An additional important attribute of metrics is that they are stable. In the case of CCP, the above distribution would be meaningless if the CCP of a project fluctuates wildly from yer to year. We estimate stability by comparing the CCP of the same project in adjacent years, from 2014 to 2019. Overall, the CCP of the projects is reasonably stable over time. The Pearson correlation between the CCP of the same project in two successive years, with 200 or more commits in each, is 0.86. The average CCP, using all commits from all projects, was 22.7% in 2018 and 22.3% in 2019. Looking at projects, the CPP grew on average by 0.6 percentage points from year to year, which might reflect a slow decrease in quality. This average hides both increases and decreases; the average absolute difference in CPP was 5.5 percentage points. Compared to the CCP distribution presented in Table 3 such changes are not very big.

Given the distribution of CCP, any developer can find the placement of his own project relative to the whole community. The classification of commits can be obtained by linking them to tickets in the ticket-handling system (such as Jira or Bugzilla). For

projects in which there is a single commit per ticket, or close to that, one can compute the CCP in the ticket-handling system directly, by calculating the ratio of bug-fixing tickets. Hence, having full access to a project, one can compute the exact CCP, rather than its maximum likelihood estimation.

Comparing the project's CCP to the distribution in the last column of Table 3 provides an indication of the project's division of effort calibrated with respect to other projects.

6 Effect of Detection Efficiency on CCP

A major factor that affects CCP is the bug detection efficiency. The higher the efficiency the more bugs are detected, leading to a higher CCP [65, 67]. The two main factors leading to higher detection efficiency are using more tests, and having more developers and users who spot defects and correct them.

6.1 Linus's Law

Linus's law, "given enough eyeballs, all bugs are shallow" [114], suggests that a large community might lead to more effective bug identification, and as a consequence also to higher CCP. Our methodology was to focus on GitHub users (which can be companies or communities of developers) that have enough very popular projects and less popular projects, and compare their bug detection efficiency. The users we selected as most suitable in our data set were Google, Facebook, Apache, Angular (led by Google), Kubernetes (designed by Google), and Tensorflow (led by Google). Note that this requirement is very restrictive and even Microsoft and Amazon were not found to be suitable. As a byproduct, these are companies and communities known for their high standards. Note that three of the communities are actually part of Google. However, since Google is a huge company, they decided to separate these projects, and we followed their decision.

For each such source, we compared the average CCP of projects in the top 5% as measured by stars (7,481 stars or more), with the average CCP of projects with fewer stars. This reflects levels of interest in the projects, because GitHub stars are both 'like' functionality and a mechanism to track projects.

Table 4 Linus's Law: CCP in projects with many or fewer stars.

	top 5% (>7,481 stars)		bottom 95% (<7,481 stars)	
Source	N	avg. CCP (lift)	N	avg. CCP
Google	8	0.32 (27%)	66	0.25
Facebook	9	0.30 (12%)	9	0.27
Apache	10	0.37 (44%)	35	0.26
Angular	3	0.49 (34%)	32	0.37
Kubernetes	3	0.21 (35%)	3	0.16
Tensorflow	5	0.26 (32%)	26	0.20

The results were that the most popular projects of high-reputation sources indeed have CCP higher than less popular projects of the same organization (Table 4). The

popular projects tend to be important projects: Google's Tensorflow and Facebook's React received more than 100,000 stars each. It is not likely that such projects have lower quality than the organization's standard. Apparently, these projects attract large communities which provide the eyeballs to identify the bugs efficiently, as predicted by Linus's law.

Note that these communities' projects, including those without so many stars, have an average CCP of 0.26, 21% more than all projects' average. Their average number of authors is 219, 144% more than the others. And the average number of stars is 5,208 compared to 1,428, a lift of 364%. It is possible that while the analysis we presented is for extreme numbers of stars, Linus's law kicks in already at much lower numbers and contributes to the difference.

There are only a few such projects (we looked at the top 5% from a small select set of sources). The effect on the CCP is modest (raising the level of bug corrections by around 30%, both a top and a median project will decrease in one decile). Thus, we expect that they will not have a significant impact on the results presented below.

6.2 Truck Factor Developers Detachment

The mirror image of projects that have enough users to benefit from Linus's law is projects that lose their core developers. The "Truck Factor" originated in the Agile community. Its informal definition is "The number of people on your team who have to be hit with a truck before the project is in serious trouble" [138]. In order to analyze it, we used the metric suggested by Avelino et al. [12]. Truck Factor Developers Detachment (TFDD) is the event in which the core developers abandon a project as if a virtual truck had hit them [11]. We used instances of TFDD identified by Avelino et al. and matched them with the GitHub behavior [11]. As expected, TFDD is a traumatic event for projects, and 59% of them do not survive it.

When comparing 1-month windows around a TFDD, the average number of commits is reduced by 1 percentage point. There is also an average reduction of 3 percentage points in refactoring, implying a small decrease in quality improvement effort. At the same time, the CCP decreases by 5 percentage points. Assuming that quality is not improved as a result of a TFDD, a more reasonable explanation is that bug detection efficiency was reduced. But even the traumatic loss of the core developers damage is only 5 percentage points.

6.3 Use of Tests

The use of tests [15, 99] is a common method to increase bug detection efficiency, so we checked its relation to CCP.

We identify test files by looking for the string "test" in the file path [4, 84, 143]. The positive rate of the "test" pattern is 15% of files contained in commits. We manually labeled 50 files and only one of them was wrong (a non-robust estimation of 97.5% accuracy). We labelled another 20 hits of the labeling function. While only 35% of the hits were test files, another 60% were related files (e.g., test data or make file), leading to a precision of 95%. The only labeled false positive had the pattern "test" as part of the string "cuttest". Note that Berger et al. reported 100% precision of the same pattern, based on 100 samples [16].

The average CCP of a project with hardly any tests is 0.20, compared to 0.22 in projects with at least some tests. Figure 5 shows CCP by test presence deciles. The CCP in the first decile is 0.20, compared to 0.25 in the last one. Hence we have a 10% difference between no tests and the rest, and 25% difference between the two extremes.

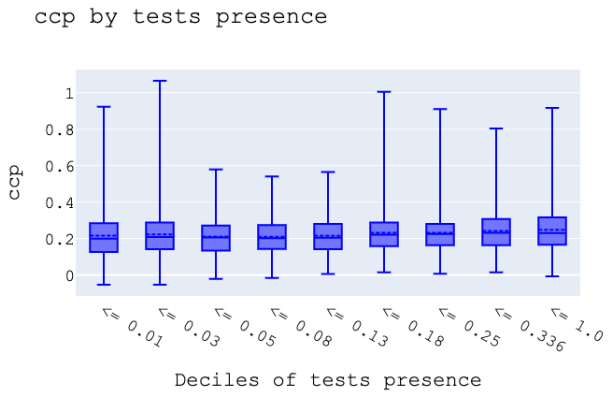

Fig. 5 CCP by Tests Presence. In this and following figures, each boxplot shows the 5, 25, 50 (solid line), 75, and 95 percentiles. The dashed line represents the mean.

In order to understand whether tests increase bug detection efficiency or that projects with more tests actually have more bugs, we use the time to identify a bug.

Kim and Whitehead reported that the median time to fix a bug is about 200 days [75]. In order to compute bug fix time one should identify the bug inducing commit, for example by using the SZZ algorithm [127] which requires access to the source code in each commit. We used the GitHub BigQuery schema so we do not have such access. Instead we use the last time the file was touched before the bug-fixing commit, 'Time from Previous Update'. This is a *lower bound* on the time to fix the bug since the bug inducing commit is this one or an earlier one. Note that a single unrelated commit between the bug inducing commit and the fixing commit is enough to reduce metric estimation and miss the true time to detect the bug. Looking at fixed bugs, the average time between a bug correction and the previous time the involved file was modified is 131 days, more than 4 months. But there is a wide distribution. In 24% of the cases the file was updated at most 1 day ago, in 41% at most a week, the median is 15 days, and in 59% at most 30 days. Assuming an exponential model, time to miss only 1% of the current bugs is $15 \cdot \log_2(100) = 15 \cdot 6.64$, which is 100 days.

Projects that hardly have tests (less than 1% test files in commits, allowing few false positives) fix a bug after 72 day on average, 33% more than the rest. This indicates that reduced testing is indeed related to inefficiency in bug-detection, and not to less bugs.

Using the same analysis we did for Linus's law, the extremely popular projects identify bugs in 51 days on average, compared to 78 days for the others from the same organizations (53% higher). Comparing per organisation the result holds for all but Angular and Kubernetes, each with only 3 extremely popular projects. Project age is on one hand an upper bound on the time to find a bug, and on the other hand

correlated with popularity. Age explains part of the behaviour but the analysis is based on a single project in some cases.

As another metric for bug detection efficiency, we used time to revert a commit. We identify reverts using git's default commit message for reverts 'This reverts commit XXX'. Identifying these messages enable us to identify pairs of reverted and reverting commits and compute the duration between them. Unlike 'Time form Previous Update', the revert duration is exact and not a lower bound. However, reverting a commit is a small part of the ways to fix a bug and only 0.2% of the commits are reverted. In popular projects the average time to revert is 15 days, compared to 27 day in the rest, 44% lower. In the projects with at least minimal tests the average time to revert is 21 days, compared to 38 days in those lacking tests, 45% lower.

Hence, we show that detection efficiency improves due to the use of tests and a high number of eyeballs. CCP increases by about 30% due to the improved detection efficiency, a small ratio compared to the 6 times gap between the 10 and 90 percentiles.

7 Effects of Coding on CCP

To further study effects related to CCP, and see what is related to higher investment in bug fixing, we studied the correlations of CCP with various project attributes. To strengthen the results beyond mere correlations we control for variables which might influence the results, such as project age and the number of developers. We also use co-change analysis and "twin" analysis, which show that the correlations are consistent and unlikely to be accidental (Sect. 4.2 and 4.3).

7.1 File Length

The correlation between high file length and an increase in bugs has been widely investigated and considered to be a fundamental influencing metric [47, 88]. The following analysis first averages across files in each project, and then considers the distribution across projects, so as to avoid giving extra weight to large projects. In order to avoid sensitivity due to large values, we capped large file lengths at 181KB, the 99th percentile.

In our projects data set, the mean file length was 8.1 KB with a standard deviation of 14.3KB, a ratio of 1.75 (capped values). Figure 6 shows that the CCP increases with the length. Projects whose average capped file size is in the lower 25% (below 3.2KB) have average CCP of 0.19. The last five deciles all have CCP around 0.23, as if from a certain point a file is "just too long".

We did not perform a co-change analysis of file length and CCP since the GitHub BigQuery database stores only the content of the files in the HEAD (last version), and not previous ones. Controlling by project age and developers support the results. When controlling for language, in most languages projects with low CCP indeed have shorter files. On the other hand, in PHP they are 10% longer, and in JavaScript the lengths in the 10% low-CCP projects are 31% higher than the rest.

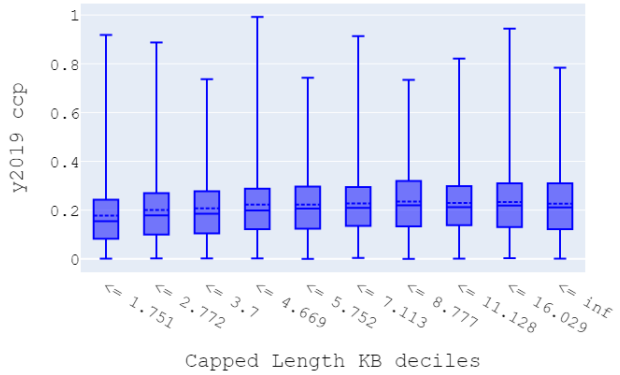

Fig. 6 CCP distribution for files with different lengths (in KB, capped).

7.2 Smells

Code smells are properties of the source code that indicate low quality and potential problems [8, 133]. They are usually found by static analysis and are programming language dependent. We used a data set of 677 Java repositories parsed by the CheckStyle tool, which identifies 151 smells [3]. Table 5 shows the Pearson correlation of several example smells with CCP, the lift of co-change with CCP, and the adjacent year stability of these smells. In addition to the individual smells, we consider the sum of all the smells that CheckStyle identifies.

Table 5 CCP and Selected Smells

Metric	Pearson with CCP	Co-change Lift	Stability
NPathComplexity	0.15	0.16	0.88
MethodLength	0.14	0.26	0.94
VisibilityModifier	0.14	0.15	0.43
AvoidInlineConditionals	0.10	0.18	0.76
Sum of Smells	0.06	0.11	0.81
NestedIfDepth	0.05	0.06	0.90

The analysis of smells and quality require a delicate analysis, considering the specific smell, the popularity, etc. An unpopular smell simply does not appear in most cases and cannot have high correlation with bugs. However, it is easy to see that smells are slightly correlated and co-change with CCP.

7.3 Coupling

A commit is a unit of work ideally reflecting the completion of a task. It should contain only the files relevant to that task. Many files needed for a task means coupling. Therefore, the average number of files in a commit can be used as a metric for coupling [4, 144]. To validate that this metric captures the way developers think about coupling, we compared it to the appearance of the terms “coupled” or “coupling” in

messages of commits containing the file. Out of the files with at least 10 commits, those with a hit rate of at least 0.1 for these terms had average commit size 45% larger than the others.

When looking at the size of commits, it turns out that corrective commits involve significantly fewer files than other commit types: the average corrective commit size is 3.8, while the average non-corrective commit size is 5.5 (median 2 for both, a longer tail for non-corrective). Therefore, comparing files with different ratios of corrective commits will influence the apparent coupling. To avoid this, we will compute the coupling using only non-corrective commits. We define the coupling of a project to be the average coupling of its files (all files, including tests).

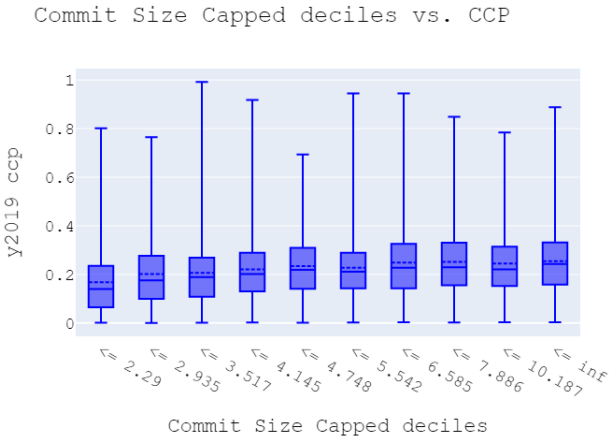

Fig. 7 CCP distribution for projects with different average commit sizes (number of files, capped, in non-corrective commits).

Figure 7 presents the results. There is a large difference in the commit sizes: The 25% quantile is 3.1 files and the 75% quantile is 7.1. Similarly to the relation of CCP to file sizes, here too the distribution of CCP in commits above the median size appears to be largely the same, with an average of 0.24. But in smaller commits there is a pronounced correlation between CCP and commit size, and the average CCP in the low coupling 25% is 0.18. Projects that are in the lower 25% in both file length and coupling have 0.15 average CCP and 29.3% chance to be in the bottom 10% of files ranked by CCP, 3 times more than expected.

When we analyze CCP and coupling co-change, the match for any improvement is 52%. A 10-percentage point reduction in CCP and a one file reduction in coupling are matched 72% of the time. Given a reduction of coupling by one file, the probability of a CCP reduction of 10 percentage points is 9%, a lift of 32%. Results hold when controlling for language, number of developers, detection efficiency, and age, though in some settings the groups are empty or very small.

In twin experiments, the probability that the developer's coupling is better (lower) in the project with lower CCP was 49%, a lift of 15%. When the coupling in the low-CCP project was better by at least one file, the developer coupling was better by one file in 33% of the cases, a lift of 72%.

8 The Project Profile and CCP

8.1 Effect of Project Domain

The effect of the domain — what the project is supposed to do — comes in two levels. First, some domains are more complicated and have more stringent requirements than others. For example, software for rocket science or flying a jet plane require more effort than a standard game on a smartphone or software for maintaining a blog. Second, some projects are simply bigger than other projects in the same domain. Bigger projects require more effort, and may contain more bugs just due to their size.

GitHub allows labeling projects with their topics. We extracted the topics and present in Table 6 some of the indicative topics out of the 100 most popular ones, showing a large CCP range. Hence, it is possible that a program will have more bugs and higher CCP since it tries to tackle a hard topic. Note, however, that from a black-box point of view, as expected from end users, quality should be indifferent to the domain. A crashing program is a problem even when the domain is hard.

Table 6 CCP by Topic (partial list of topics)

Topic	Repositories	CCP
minecraft	31	0.28
microservices	33	0.26
security	75	0.26
cloud	61	0.24
hacktoberfest	532	0.24
blockchain	46	0.24
database	84	0.24
devops	39	0.24
audio	30	0.23
kubernetes	103	0.23
linux	143	0.23
windows	93	0.23
raspberry-pi	31	0.23
bioinformatics	40	0.23
editor	37	0.23
bot	32	0.22
deep-learning	48	0.22
ui	38	0.22
ios	70	0.22
gui	31	0.22
android	146	0.22
machine-learning	96	0.21
docker	119	0.21
framework	84	0.21
library	50	0.20
wordpress	40	0.19

On the other hand, comparing hard and easy domains might introduce noise in analysis of source code and be considered an unfair benchmark. But even if topics influence CCP, this is not a suitable variable to use as control. The most popular topic, hacktoberfest[3], appears in only 7% of the projects. There are 13,581 topics and

[3] https://hacktoberfest.digitalocean.com/

the average number of topics per project is 4.5. Hence, even if the controlled groups will be large enough, it is not clear what is the proper control group of a project labeled as 'framework', 'linux', and 'machine-learning'. Comparison of projects from different domains is a threat, yet combinations of topics are very diverse and have a low popularity, so one can hope that the influence on the analysis is low.

Other than the topic itself, a given topic might indicate other differences, like the programming language. We checked the influence of Java programs' domains by observing the packages they use. Android applications have an average CCP of 0.24, Swing (a user interface library) programs have CCP of 0.22, and Serlvets and programs involving concurrency, databases, or security have average CCP of 0.23. Hence, given the same programming language, the CCP difference between rather different domains is rather small.

8.2 Number of Developers and CCP

The number of developers, via some influence mechanisms (e.g., ownership), was investigated as a quality factor and it seems that there is some relation to quality [21, 101, 137]. The number of developers and CCP have Pearson correlation of 0.12. The number of developers can reach very high values and therefore be very influential.

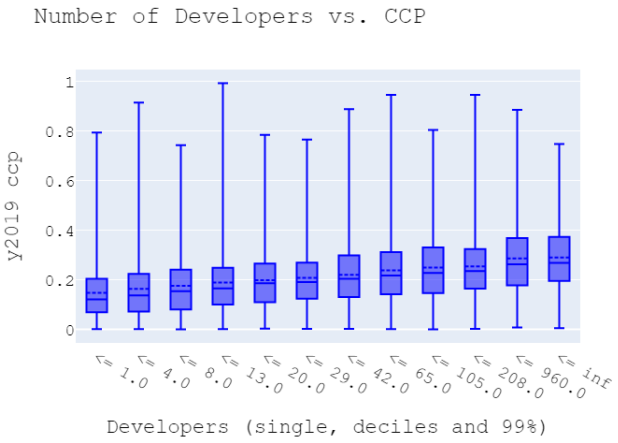

Fig. 8 CCP distribution for projects with different numbers of developers.

Fig. 8 shows that percentiles of the CCP distribution increase monotonically with the number of developers. There are several possible explanations for this phenomenon. It might be simply a proxy to the project size (i.e. to the LOC). It might be due to the increased communication complexity and the difficulty to coordinate multiple developers, as suggested by Brooks in the mythical "The Mythical Man Month" [29]. Part of it might also be a reflection of Linus's law, as discussed in Sect. 6.1.

When investigating other variables, we control for the number of developers by dividing the projects into 3 groups: the 25% of projects with the least developers have

few developers (at most 10), the next 50% are *intermediate* (at most 80), and the rest have *numerous* developers. We then check if the results hold for each such group.

8.3 Programming Languages and CCP

The influence of programming language on software attributes such as quality is a highly contentious topic [17, 19, 77, 100, 107, 113]. Other than the direct language influence, languages are often used in different domains, and indirectly imply programming culture and communities. Our investigation of programming languages is only superficial, and aims mainly to advise the possible control of language due to their influence on CCP.

We extracted the 100 most common file name extensions in GitHub, which cover 94% of the files. Of these, 28 extensions are of Turing-Complete programming languages (i.e., excluding languages like SQL). We consider a language to be the dominant language in a project if above 80% of files were in this language. There were 5,407 projects with a dominant language out of the 7,557 being studied. Figure 9 shows the CDFs of the CCP of projects in major languages.

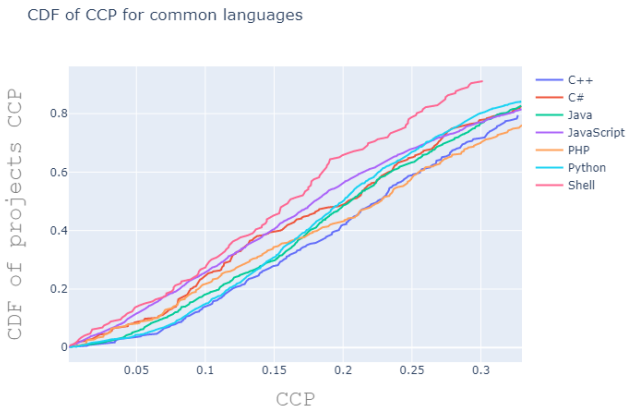

Fig. 9 Cumulative distribution of CCP by language. Distributions shifted to the right tend to have higher CCP.

The figure focuses on the low to medium CCP region (excluding the highest CCPs). For averages see Table 7. All languages cover a wide and overlapping range of CCP, and in all languages one can write code with few bugs. The least bugs occurred in Shell scripts. This is an indication of the need to analyze quality carefully, as Shell is used to write scripts and should not be compared directly with languages used to write, for example, real-time applications. Project in JavaScript, and to a somewhat lesser degree, in C#, also tend to have lower CCPs. Higher CCPs occur in C++, and, towards the tail of the distribution, in PHP. The rest of the languages are usually in between with changing regions of lower CCP.

In order to verify that differences are not accidental, we split the projects by language and examined their average CCP. An ANOVA test [43] led to an F-statistic of

Table 7 CCP and development speed (commits per year of involved developers) per language. Values are averages ± standard errors.

Language	Projects	Metric			
		CCP	Speed	Speed in low-CCP 10%	Speed in others
Shell	146	0.18 ± 0.010	171 ± 10	185 ± 29	169 ± 11
JavaScript	1342	0.20 ± 0.004	156 ± 3	166 ± 8	154 ± 3
C#	315	0.21 ± 0.008	181 ± 6	207 ± 27	178 ± 7
Python	1069	0.22 ± 0.004	139 ± 3	177 ± 19	137 ± 3
Java	764	0.22 ± 0.005	148 ± 4	205 ± 17	143 ± 4
C++	341	0.24 ± 0.007	201 ± 7	324 ± 33	196 ± 7
PHP	326	0.25 ± 0.009	168 ± 6	180 ± 22	167 ± 6

8.3, indicating that language indeed has a substantial effect, with a p-value around 10^{-9}. Hence, as Table 7 shows, there are statistically significant differences among the programming languages, yet compared to the range of the CCP distribution they are small.

Of course, the above is not a full comparison of programming languages (See [17, 100, 107, 113] for comparisons and the difficulties involving them). Many factors (e.g. being typed, memory allocation handling, compiled vs. dynamic) might cause the differences in the languages' CCP. Our results agree with the results of [17, 113], indicating that the difference between languages is usually small and that C++ has relatively high CCP.

8.4 Project Age

Lehman's laws of software evolution imply that quality may have a negative correlation with the age of a project [81, 82]. We checked whether more bugs are found in older projects in our dataset. We first filtered out projects that started before 2008 (GitHub beginning). For the remaining projects, we checked their CCP each year. Figure 10 shows that CCP indeed tends to increase slightly with age. In the first year, the average CCP is 0.18. There is then a generally upward trend, getting to an average of 0.23 in 10 years. Note that there is a survival bias in the data presented since many projects do not reach high age.

Wanting to control age, we divided the projects into 4 age groups. Those started earlier than 2008, GitHub's start, were excluded from the control. Those started in 2018–2019 (23%) are considered to be *young*, the next, from 2016–2017 (40%), are *medium*, and those from 2008–2015 (37%) are *old*. When we obtained a result (e.g., correlation between coupling and CCP), we checked if the result holds for each of the groups separately.

8.5 Developer Engagement and CCP

The relation between churn (developers abandoning the project) and quality steps out of the technical field and involves human psychology. Motivation influences performance [30, 140]. Argyle investigated the relation between developers' happiness and their job satisfaction and work performance, showing "modestly positive correlations with productivity, absenteeism, and labour turnover" [9]. In the other direction,

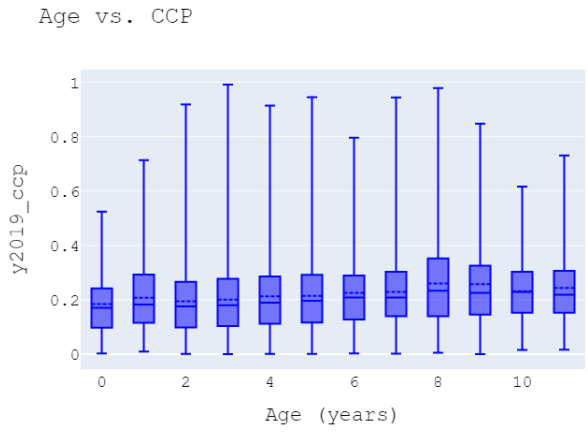

Fig. 10 CCP distribution (during 2019) in projects of different ages.

Ghayyur et al. conducted a survey in which 72% claimed that poor code quality is demotivating [46]. Hence, quality might be both the outcome and the cause of motivation.

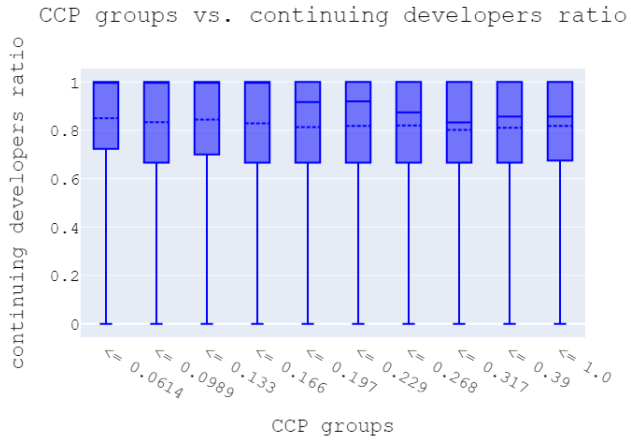

Fig. 11 Projects' developer retention per CCP decile. Note the change in the median.

We checked the retention of involved developers, where retention is quantified as the percentage of developers that continue to work on the project in the next year, averaged over all years (Figure 11). Note that the median is 100% retention in all four low-CCP deciles, decreases over the next three, and stabilizes again at about 85% in the last three CCP deciles.

When looking at co-change of CCP with churn ($1 - \textit{retention}$), the match is only 51% for any change but 79% for a change of at least 10 percentage points in each metric.

An improvement of 10 percent points in CCP leads to a significant improvement in churn in 21% of the cases, a lift of 17%. When controlling the language, age group, or developer number group, we still get matching co-change. When controlling for detection efficiency, we get a small -3% precision lift for high efficiency, and result holds for medium and low.

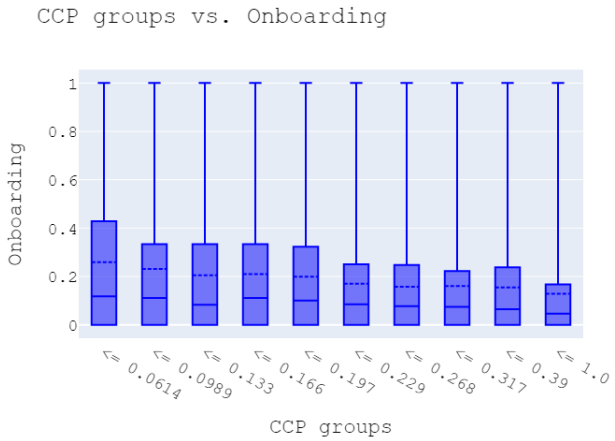

Fig. 12 On-boarding per CCP decile.

Acquiring new developers complements the retention of existing ones. We define the on-boarding ratio as the average percentage of new developers becoming involved. Figure 12 shows that the higher the CCP, the lower is the on-boarding, and on-boarding average is doubled in the first decile compared to the last.

In order to be more robust to noise, we consider projects that have at least 10 new developers. When looking at co-change of on-boarding and CCP, the match is only 53% for any change but 85% for a change of at least 10 percent points in both metrics. An improvement of 10 percent points in CCP leads to a significant improvement in on-boarding in 10% of the cases, a lift of 18%. When controlling on language, results fit the relation other than in PHP and Shell (which had a small number of cases). Results hold for all age groups. For size, they hold for intermediate and numerous numbers of developers; by definition, with few developers there are no projects with at least 10 new developers. When controlling for detection efficiency, we get -1% precision lift for low, and the result holds for medium and high.

8.6 Development Speed and CCP

The definition of productivity is subjective and ill-defined. Measures including LOC [91], modules [95], and function points [64, 90] per time unit have been suggested and criticized [71, 72]. We chose development speed by the number of commits per involved developer per year to be our main productivity metric. This is an output per time measure, and the inverse of time to complete a task, investigated in the classical work of Sackman et al. [119]. The number of commits is correlated with self-rated productivity

[98] and team lead perception of productivity [104]. Commits are also suitable as the output unit since a commit is a unit of work, its computation is easy and objective, and it is not biased toward implementation details.

The number of commits per project per year is relatively stable with a Pearson correlation of 0.71. The number of developers per year is also stable with a Pearson correlation 0.81. To study development speed we omit developers with fewer than 12 commits per year since they are non-involved developers. We also capped the number of commits per developer at 500, about the 99th percentile of the developers' contributions. While commits by users below the 99th percentile are only 73% of the total, excluding the long tail (which reaches 300,000 commits) is justified because it most probably does not represent usual manual human effort. Using both restrictions the correlation of commits per developer in adjacent years is 0.62 (compared to 0.59 without them), which is reasonably stable.

As Figure 13 shows, there is a steady decrease of speed with CCP. The average speed in the first decile is 56% higher than in the last one. Speed differs in projects written in different languages. Yet in all of them lower CCP goes with higher speed (see Table 7).

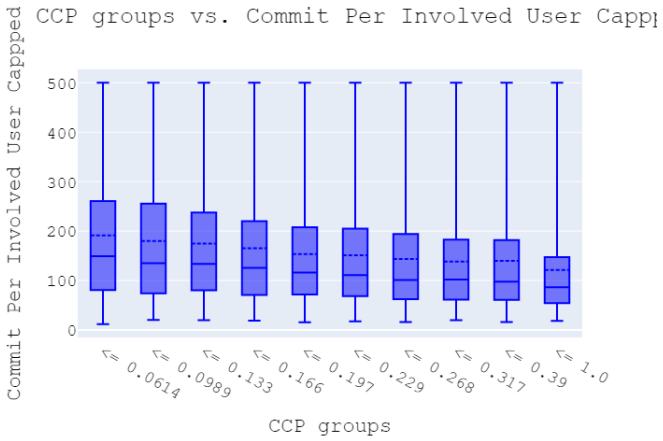

Fig. 13 Distribution of commits per year of involved developers (capped) per CCP decile.

We also conducted twin experiments, to control the developer. When a developer works in a faster project, he is faster than himself in other projects in 51% of the cases, 8% lift. When the project speed is 10 commits larger, the developer has 42% chance to be also 10 commits faster than himself, a lift of 11%.

Investigating co-change of CCP and speed, in 52% of the cases, an improvement in CCP goes with an improvement in speed. Given a CCP improvement, there is a speed improvement in 53% of the cases, a lift of 4%. Given a 10 percent points improvement in CCP, the probability of 10 more commits per year per developer is 53%, and the lift is 2%. In the other direction, given an improvement in speed the probability of a significant improvement in CCP drops to 7%. Hence, knowing of a significant improvement in CCP, a speed improvement is likely, but knowing of a speed improvement a significant CCP improvement is very unlikely.

When controlling for age or language, results hold. Results also hold for intermediate and numerous developer groups, with a positive lift when the change is significant, but a -3% lift in the few developers group for any change. When controlling for detection efficiency, the precision lift is -2% for low efficiency, and holds for medium and high.

The GitHub Torrent BigQuery schema enabled us to also use productivity metrics based on pull requests and issues [48]. Note that while our project selection represents large projects active during 2019, the selection criteria of Gousios and Spinellis [48] were different, and covered 2011–2016. There are 5,165 projects in the schemas intersection, 68% of our data set. Note that this selection has a strong bias towards relatively old and long living projects.

The first interesting result from this investigation is that output measures are not very correlated, though they should represent the same concept. We measure Pearson correlation with commits, the metric available to us on all projects. Merged issues have correlation of 0.17, merged pull requests 0.51, and developers (output producers) have correlation of 0.42. Despite the differences between output metrics, Table 8 shows that all of them co-change and have positive precision lift with respect to CCP. Note that in some cases the lift is higher than that with commits (output metric) and commits per involved developer (productivity metric).

We also measured a new productivity metric, the average duration between a commit and the prior one on the same day. The requirement for the same day overcomes large gaps of inactivity of open source developers that could be misinterpreted as long tasks. We manually labeled 50 such cases to validate that they fit the duration needed for the code change. Table 8 shows that the same-day duration is highly stable with 0.87 adjacent-years Pearson correlation, and co-changes with both commits and CCP. Commit duration is an investment metric, and the co-change indicates that the less bugs, the faster are the commits per involved developer (CPID). Therefore, the relation between quality and productivity holds for various productivity metrics.

Table 8 CCP and Productivity

Metric	Stability	CCP lift	Commits lift	CPID lift
CCP	0.83	–	0.13	0.27
Commits	0.81	0.13	–	0.32
Commits per involved developer	0.91	0.27	0.32	–
Merged issues per involved developer	0.50	0.33	0.16	-0.12
Merged PRs per involved developer	0.76	0.04	0.13	0.25
Same day duration avg	0.87	0.11	0.19	0.13

The above results can also be used to reflect on the relationship between quality and productivity. There are two opposing theories regarding this relationship. The classical Iron Triangle [103] sees them as a trade-off: investment in quality comes *at the expense of* productivity. On the other hand, "Quality is Free" claims that investment in quality is beneficial in general and leads to *increased* productivity [34]. Our results in Fig. 13 and Table 7 indicate that productivity (as operationalized by commits per year per involved developer) is correlated with lower CCP, namely with lower relative investment in bug fixing, leaving a larger fraction of the effort to making progress with the project.

Note that this is not a tautology: it is not just that you produce more non-corrective commits when CCP is low, it is that you produce more commits overall. So low-CCP projects enjoy a win-win increase in productivity: they produce more commits, and more of these commits are not wasted on fixing bugs. The twin experiments help to reduce noise, demonstrating that development speed is a characteristic of the project. In case that this correlation is indeed due to causality, then when you have fewer bugs you also gain speed, enjoying both worlds. This relation between quality and development speed is also supported by Jones's research on time wasted due to low quality [66, 68] and developers performing the same tasks during "Personal Software Process" training [126].

9 Is CCP Redundant?

We examined many variables of projects, coupling, length, programming language, etc. If CCP is a function of these variables and can be predicted by them, it adds no additional value. In this section we examine to what extent the variables that we consider can predict CCP. We do it by building a machine learning model based on them, trying to predict CCP.

Note that unlike the regular use case of machine learning, we do not need a model to predict CCP. Unless it is a new project with very few commits, we can measure the CCP directly. But if we can build a model that predicts a project's CCP, this means that we have identified all the relevant factors that affect CCP. The model that we built has a modest goal, and attempts to predict only whether a project will be in the 50% with the lowest CCP or not. We chose to divide to two equal-size groups at 50% in order to avoid problems due to imbalanced data sets [78, 102, 134].

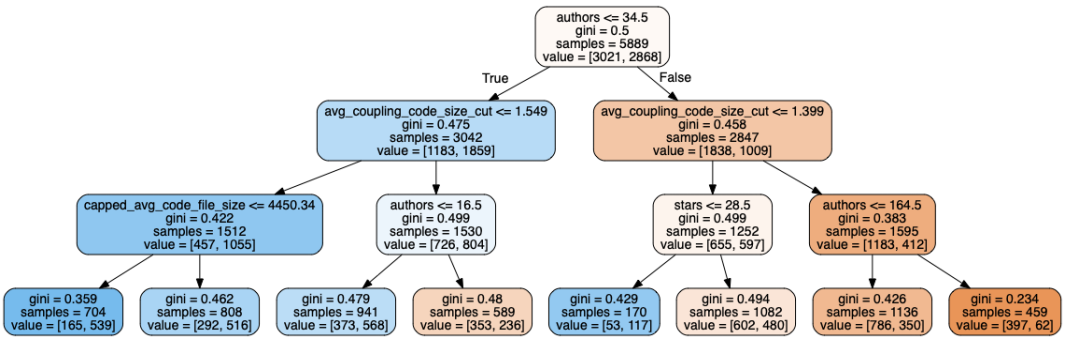

Fig. 14 Model Predicting High Quality Median

We present in Figure 14 a small tree model generated using scikit-learn [105] that is simple to understand, of accuracy 63%. We present it to demonstrate that a very small model can predict quite well. Blue represents high quality probability and orange represents low quality.

Decision trees [108] are greedy algorithms. Hence the model uses the number of the authors as its root, since this is the most informative feature. Note that one cannot deduce that the next levels features are the next most indicative ones since their utility

is evaluated when they are conditioned on the prior nodes. Also note that decision tree classifiers use stooping rules and pruning rules that limit the evolution of the tree. If a feature is not included in the tree, it does not imply that it is not informative. It means that it was not informative enough at the given structure of the tree. The model demonstrates that authors (size, Section 8.2), coupling (quality, Section 7.3), file length (quality, Section 7.1) and stars (detection efficiency, Section. 6) are informative. Yet, even when using them, the leaves are rather diverse as demonstrated by the Gini metric values [40] and the class distribution (the value property). Note that many other models and features are also useful, this tree is an example, not the only model.

Our goal is to understand to what extent the features we discussed above explain the CCP. There are several possible outcomes. If we cannot make good predictions, this means that the features are not useful, although they might be useful combined with more features or in a different model. If the prediction accuracy is perfect, not only that we can predict, but we also know that there are no other causal features that are missing. If any additional causal feature was missing, the concept would not be a function of the given features, and accuracy would not be perfect.

Our results are in between. The best accuracy we got was 68%. We also trained and evaluated the accuracy of a high capacity random forest on the same data set, in order to reach overfitting deliberately [10]. We reached an accuracy of 72%, which serves as an estimate of the upper bound on the best possible classifier results on this data set. Thus the result of 68% is close to that level. Though the features discussed improve the predictive power significantly relative to the majority rule, even combined they cannot capture the CCP. Therefore CCP is influenced by other factors too and cannot be replaced by a function of the discussed ones.

10 Threats to Validity

We set out to measure the Corrective Commit Probability and do so based on a linguistic analysis, what poses a construct validity threat. We investigated whether it is indeed accurate and precise in Section 3.4. The number of test labeled commits is small, about 1,000, hence there is a question of how well they represent the underlying distribution. We evaluated the sensitivity to changes in the data. Since the model was built mainly using domain knowledge and a different data set, we could use a small training set. Therefore, we preferred to use most of the labels as a test set for the variables estimation and to improve the estimation of the *recall* and Fpr.

The labeling was done manually by humans who are prone to error and subjectivity. In order to make the labeling stricter, we used a labeling protocol, provided in the supplementary materials. Out of the samples, 400 were labeled by three annotators independently. The labels were compared in order to evaluate the amount of uncertainty. Other than uncertainty due to different opinions, there was uncertainty due to the lack of information in the commit message. For example, the message "Changed result default value to False" describes a change well but leaves us uncertain regarding its nature. We used the gold standard labels to verify that this is rare.

Our main assumption is the conditional independence [23, 85] between the corrective commits (code) and the commit messages describing them (process) given our concept (the commit being corrective, namely a bug fix). This means that the model performance is the same over all the projects, and a different hit rate is due to a different CCP. This assumption is invalid in some cases. For example, projects documented

in a language other than English will appear to have no bugs. Non-English commit messages are relatively easy to identify; more problematic are differences in English fluency. Native English speakers are less likely to have spelling mistakes and typos. A spelling mistake might prevent our model from identifying the textual pattern, thus lowering the recall. This will lead to an illusive benefit of spelling mistakes, misleading us to think that people who tend to have more spelling mistakes tend to have fewer bugs.

Another threat to validity is due to the family of models that we chose to use. We chose to represent the model using two parameters, *recall* and *Fpr*, following the guidance of Occam's razor and resorting to a more complex solution only when a need arises. However, many other families of models are possible. We could consider different sub-models for various message lengths, a model that predicts the commit category instead of the Boolean "Is Corrective" concept, etc. Each family will have different parameters and behavior. More complex models will have more representative power but will be harder to learn and require more samples.

A common assumption in statistical analysis is the IID assumption (Independent and Identically Distributed random variables). This assumption clearly does not hold for GitHub projects. We found that forks, projects based on others and sharing a common history, were 35% of the active projects. We therefore removed forks, but projects might still share code and commits. Also, older projects, with more commits and users, have higher weight in twin studies and co-change analysis.

Our metric focuses on the fraction of commits that *correct* bugs. One can claim that the fraction of commits that *induce* bugs is a better metric when one is interested in quality. In principle, this can be done using the SZZ algorithm (the common algorithm for identifying bug inducing commits [127]). But note that SZZ is applied after the bug was identified and fixed. Thus, the inducing and fixing commits are actually expected to give similar results.

Another major threat concerns internal validity. As we noted, a low CCP can result from a disregard for fixing bugs or an inability to do so. On the other hand, in extremely popular projects, Linus's law "given enough eyeballs, all bugs are shallow" [114] might lead to more effective bug identification and high CCP. Likewise, improvements in bug detection (e.g., by doubling the QA department) can have a large effect on the CCP. We identify such cases and discuss them in Section 6.

Focusing on corrective commits also leads to several biases. Most obviously, existing bugs that have not been found yet are unknown. Finding and fixing bugs might take months [75]. Different policies and methods, such as the adoption of continuous integration [18], might change the time to merge fixes. When projects differ in the time needed to identify a bug, our results will be biased.

Tasks differ in size and difficulty and their translation to commits might differ due to the project or developer habits. Commits may also include a mix of different tasks. In order to reduce the influence of project culture we aggregated many of them. In order to eliminate the effect of personal habits, we used twins experiments. Other than that, the number of commits per time is correlated to developers' self-rated productivity [98] and team lead perception of productivity [104], hence it provides a good computable estimator.

Software development is usually done subject to lack of time and resources. Due to that, many times known bugs of low severity are not fixed. While this leads to a bias, it can be considered to be a desirable one, by focusing on the more important bugs. In the other direction, we give all fixes the same weight. When such data is available,

it might be more proper to give higher weight to important bugs, distinguish between bugs by their cause, etc.

A threat to external validity might arise due to the use of open source projects that might not represent projects done in software companies. We feel that the open source projects are of significant interest on their own. Other than that, the projects we analyzed include projects of Google, Microsoft, Apple, etc. so at least part of the area is covered.

The decision to use commits as the basic entity and assuming they are atomic is another threat. One could choose pull requests as the basic entity, which reflect 6 commits on average. On the other hand, it is known that many commits are tangled [57, 59], serving more than one goal, for example including both a fix and a refactor. Moreover, the relation is not one-to-one: a bug might be fixed in more than a single commit, and a commit might resolve several bugs or tasks. We showed in Section 8.6 that pull requests, issues, and commits are only moderately related with respect to the number of them done in a year. Hence these entities are indeed different, and one should use the one that best serves one's interests. Commits are the only entities available in the BiqQuery schema of GitHub repositories, enabling up-to-date analysis. Commits are also smaller units than pull requests and issues, and directly reflect the work of the developer modifying the code. We therefore find them to be the better choice.

Time, cost, and development speed are problematic to measure. We use commits as a proxy to work since they typically represent tasks. However, quality, productivity, and their relations can be defined in many different ways and should be further investigated. For example, it is possible to measure productivity as time to merge a fix, as done by da Costa et al. [36], who report lower productivity after the introduction of continuous integration, that should increase quality. We do not have the needed data to replicate their analysis. We extended the analysis with pull requests and issues when this data was available, showing that the result holds with a variety of productivity metrics.

11 Conclusions

We presented the Corrective Commit Probability (CCP), a metric for the relative effort invested in fixing bugs, reflecting on the health of a project and its code. We started off with a linguistic model to identify corrective commits, significantly improving prior work [4, 6, 60, 83], and developed a mathematical method to find the most likely CCP given the model's hit rate.

The CCP metric has the following properties:

- It is stable: it reflects the character of a project and does not change much from year to year.
- It is informative in that it has a wide range of values and distinguishes between projects.

We estimated the CCP of all 7,557 independent large active projects in 2019 in BigQuery's GitHub data. This created a quality scale, enabling observations on the state of the practice. Projects at the top of the scale spend more than 6 times as much effort on bug fixing as projects at the bottom of the scale. Using this scale developers can compare their project's relative investment in fixing bugs (as reflected by CCP) to the community. A low percentile may suggest the need to invest more effort.

We furthermore show a correlation between CCP and various project attributes, including long files, coupling, occurrence of code smells, low perceived quality, lower productivity, developer churn, and less effective onboarding. These can serve as starting points for research on how such project attributes may affect the division of effort between bug fixing and making continued progress with the project's development.

Supplementary Materials

The language models are available at `https://github.com/evidencebp/commit-classification`. Utilities used for the analysis (e.g., co-change) are at `https://github.com/evidencebp/analysis_utils`. Database construction code is available at `https://github.com/evidencebp/general`. All other supplementary materials can be found at `https://github.com/evidencebp/corrective-commit-probability`.

Acknowledgements

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 832/18). We thank Amiram Yehudai and Stanislav Levin for providing us their data set of labeled commits [83]. We thank Guilherme Avelino for drawing our attention to the importance of Truck Factor Developers Detachment (TFDD) and providing a data set [11]. Many thanks to the reviewers whose comments were instrumental in improving the focus of the paper.

References

1. H. Al-Kilidar, K. Cox, and B. Kitchenham. The use and usefulness of the ISO/IEC 9126 quality standard. In *Intl. Synp. Empirical Softw. Eng.*, pages 126–132, Nov 2005.
2. M. Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 143–153, New York, NY, USA, 2019. Association for Computing Machinery.
3. I. Amit, N. B. Ezra, and D. G. Feitelson. Follow your nose – which code smells are worth chasing?, 2021.
4. I. Amit and D. G. Feitelson. Which refactoring reduces bug rate? In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'19, pages 12–15, New York, NY, USA, 2019. ACM.
5. I. Amit, E. Firstenberg, and Y. Meshi. Framework for semi-supervised learning when no labeled data is given. U.S. patent application #US20190164086A1, 2017.
6. J. J. Amor, G. Robles, J. M. Gonzalez-Barahona, and A. Navarro. Discriminating development activities in versioning systems: A case study, Jan 2006.
7. G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, New York, NY, USA, 2008. Association for Computing Machinery.
8. F. Arcelli Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In *IEEE International Conference on Software Maintenance, ICSM*, pages 260–269, 09 2013.
9. M. Argyle. Do happy workers work harder? the effect of job satisfaction on job performance. In R. Veenhoven, editor, *How harmful is happiness? Consequences of enjoying life or not*. Universitaire Pers, Rotterdam, The Netherlands, 1989.

10. D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, et al. A closer look at memorization in deep networks. *arXiv preprint arXiv:1706.05394*, 2017.
11. G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. *CoRR*, abs/1906.08058, 2019.
12. G. Avelino, L. T. Passos, A. C. Hora, and M. T. Valente. A novel approach for estimating truck factors. *CoRR*, abs/1604.06766, 2016.
13. R. Baggen, J. P. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Softw. Quality J.*, 20(2):287–307, Jun 2012.
14. V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.
15. B. Beizer. *Software testing techniques*. Dreamtech Press, 2003.
16. E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek. On the impact of programming languages on code quality. *CoRR*, abs/1901.10220, 2019.
17. E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.*, 41(4), Oct 2019.
18. J. H. Bernardo, D. A. da Costa, and U. Kulesza. Studying the impact of adopting continuous integration on the delivery time of pull requests. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 131–141, New York, NY, USA, 2018. Association for Computing Machinery.
19. P. Bhattacharya and I. Neamtiu. Assessing programming language impact on development and maintenance: a study on C and C++. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 171–180, 2011.
20. C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM.
21. C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.
22. C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, May 2009.
23. A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 92–100, New York, NY, USA, 1998. ACM.
24. B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, Jan 2001.
25. B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
26. B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Intl. Conf. Softw. Eng.*, number 2, pages 592–605, Oct 1976.
27. B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, Oct 1988.
28. G. Box. Robustness in the strategy of scientific model building. In R. L. LAUNER and G. N. WILKINSON, editors, *Robustness in Statistics*, pages 201 – 236. Academic Press, 1979.
29. F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.
30. J. P. Campbell, R. A. McCloy, S. H. Oppler, and C. E. Sager. A theory of performance. In N. Schmitt, W. C. Borman, and Associates, editors, *Personnel Selection in Organizations*, pages 35–70. Jossey-Bass Pub., 1993.
31. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, Jun 1994.
32. J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

33. L. Corral and I. Fronza. Better code for better apps: A study on source code quality and market success of android applications. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 22–32, 2015.
34. P. Crosby. *Quality Is Free: The Art of Making Quality Certain*. McGrawHill, 1979.
35. W. Cunningham. The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, page 29–30, New York, NY, USA, 1992. Association for Computing Machinery.
36. D. A. da Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan. The impact of rapid release cycles on the integration delay of fixed issues. *Empirical Software Engineering*, 23(2):835–904, 2018.
37. M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, May 2010.
38. A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):20–28, 1979.
39. M. Dawson, D. Burrell, E. Rahim, and S. Brewster. Integrating software assurance into the software development life cycle (SDLC). *Journal of Information Systems Technology and Planning*, 3:49–53, 2010.
40. R. Dorfman. A formula for the gini coefficient. *The review of economics and statistics*, pages 146–149, 1979.
41. G. Dromey. A model for software product quality. *IEEE Trans. Softw. Eng.*, 21(2):146–162, Feb 1995.
42. B. Efron. *Bootstrap Methods: Another Look at the Jackknife*, pages 569–593. Springer New York, New York, NY, 1992.
43. R. Fisher. The correlation between relatives on the supposition of mendelian inheritance. *Transactions of the Royal Society of Edinburgh*, 52(2):399–433, 1919.
44. M. Fowler, K. Beck, and W. R. Opdyke. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.
45. M. Gharehyazie, B. Ray, M. Keshani, M. S. Zavosht, A. Heydarnoori, and V. Filkov. Cross-project code clones in github. *Empirical Software Engineering*, 24(3):1538–1573, Jun 2019.
46. S. A. K. Ghayyur, S. Ahmed, S. Ullah, and W. Ahmed. The impact of motivator and demotivator factors on agile software development. *International Journal of Advanced Computer Science and Applications*, 9(7), 2018.
47. Y. Gil and G. Lalouche. On the correlation between size and metric validity. *Empirical Softw. Eng.*, 22(5):2585–2611, Oct 2017.
48. G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, 2012.
49. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.
50. T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct 2005.
51. R. Hackbarth, A. Mockus, J. Palframan, and R. Sethi. Improving software quality as customers perceive it. *IEEE Softw.*, 33(4):40–45, Jul/Aug 2016.
52. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov 2012.
53. M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.
54. C. Hastings, F. Mosteller, J. W. Tukey, and C. P. Winsor. Low moments for small samples: A comparative study of order statistics. *Ann. Math. Statist.*, 18(3):413–426, 09 1947.
55. L. P. Hattori and M. Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 63–71. IEEE, 2008.
56. D. M. Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

57. S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodríguez-Pérez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaeej, I. Amit, B. Turhan, S. Eismann, A.-K. Wickert, I. Malavolta, M. Sulir, F. Fard, A. Z. Henley, S. Kourtzanidis, E. Tuzun, C. Treude, S. M. Shamasbi, I. Pashchenko, M. Wyrich, J. Davis, A. Serebrenik, E. Albrecht, E. U. Aktas, D. Strüber, and J. Erbel. Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling. arXiv:2011.06244 [cs.SE], 2020.
58. K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.
59. K. Herzig and A. Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013.
60. A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39, May 2009.
61. D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec 2004.
62. I. IEC. 9126-1 (2001). software engineering product quality-part 1: Quality model. *International Organization for Standardization*, page 16, 2001.
63. International Organization for Standardization. Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models, 2011.
64. Z. Jiang, P. Naudé, and C. Comstock. An investigation on the variation of software development productivity. *IEEE Transactions on Software Engineering*, 1(2):72–81, 2007.
65. C. Jones. *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, Inc., New York, NY, USA, 1991.
66. C. Jones. Social and technical reasons for software project failures. *CrossTalk, The J. Def. Software Eng.*, 19(6):4–9, 2006.
67. C. Jones. Software quality in 2012: A survey of the state of the art, 2012. [Online; accessed 24-September-2018].
68. C. Jones. Wastage: The impact of poor quality on software economics. retrieved from http://asq.org/pub/sqp/. *Software Quality Professional*, 18(1):23–32, 2015.
69. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian. The promises and perils of mining github (extended version). *Empirical Software Engineering*, 01 2015.
70. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, June 2013.
71. C. F. Kemerer. Reliability of function points measurement: A field experiment. *Commun. ACM*, 36(2):85–97, Feb 1993.
72. C. F. Kemerer and B. S. Porter. Improving the reliability of function point measurement: An empirical study. *IEEE Trans. Softw. Eng.*, 18(11):1011–1024, Nov 1992.
73. F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality?: An empirical case study of mozilla firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 179–188, Piscataway, NJ, USA, 2012. IEEE Press.
74. F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.
75. S. Kim and E. J. Whitehead, Jr. How long did it take to fix bugs? In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 173–174, New York, NY, USA, 2006. ACM.
76. S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.
77. P. S. Kochhar, D. Wijedasa, and D. Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573, 2016.

78. B. Krawczyk. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence*, 5(4):221–232, 2016.
79. P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.
80. T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.
81. M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
82. M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution – the nineties view. In *Intl. Software Metrics Symp.*, number 4, pages 20–32, Nov 1997.
83. S. Levin and A. Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, pages 97–106, New York, NY, USA, 2017. ACM.
84. S. Levin and A. Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–46. IEEE, 2017.
85. D. D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In C. Nédellec and C. Rouveirol, editors, *Machine Learning: ECML-98*, pages 4–15, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
86. B. P. Lientz. Issues in software maintenance. *ACM Comput. Surv.*, 15(3):271–278, Sep 1983.
87. B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Comm. ACM*, 21(6):466–471, Jun 1978.
88. M. Lipow. Number of faults per line of code. *IEEE Transactions on software Engineering*, (4):437–439, 1982.
89. C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.
90. K. D. Maxwell and P. Forselius. Benchmarking software development productivity. *IEEE Software*, 17(1):80–88, Jan 2000.
91. K. D. Maxwell, L. Van Wassenhove, and S. Dutta. Software development productivity of european space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, Oct 1996.
92. T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, Jul 1976.
93. A. Mockus, D. Spinellis, Z. Kotti, and G. J. Dusing. A complete set of related git repositories identified via community detection approaches based on shared commits, 2020.
94. A.-J. Molnar, A. NeamŢu, and S. Motogna. Evaluation of software product quality metrics. In E. Damiani, G. Spanoudakis, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 163–187, Cham, 2020. Springer International Publishing.
95. S. Morasca and G. Russo. An empirical study of software productivity. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 317–322, Oct 2001.
96. R. Moser, W. Pedrycz, and G. Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 309–311, New York, NY, USA, 2008. ACM.
97. N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22, 04 2017.
98. E. Murphy-Hill, C. Jaspan, C. Sadowski, D. C. Shepherd, M. Phillips, C. Winter, A. K. Dolan, E. K. Smith, and M. A. Jorde. What predicts software developers' productivity? *Transactions on Software Engineering*, 2019.
99. G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.
100. S. Nanz and C. A. Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788, May 2015.

101. B. Norick, J. Krohn, E. Howard, B. Welna, and C. Izurieta. Effects of the number of developers on code quality in open source software: a case study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–1, 2010.
102. R. Oak, M. Du, D. Yan, H. Takawale, and I. Amit. Malware detection on highly imbalanced data through sequence modeling. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19, page 37–48, New York, NY, USA, 2019. Association for Computing Machinery.
103. R. Oisen. Can project management be defined? *Project Management Quarterly*, 2(1):12–14, 1971.
104. E. Oliveira, E. Fernandes, I. Steinmacher, M. Cristo, T. Conte, and A. Garcia. Code and commit metrics of developer productivity: a study on team leaders perceptions. *Empirical Software Engineering*, 04 2020.
105. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
106. A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100. IEEE, 2014.
107. L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct 2000.
108. J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar 1986.
109. F. Rahman and P. Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441, May 2013.
110. F. Rahman, D. Posnett, A. Hindle, E. T. Barr, and P. T. Devanbu. Bugcache for inspections: hit or miss? In *SIGSOFT FSE*, 2011.
111. L. Rantala, M. Mäntylä, and D. Lo. Prevalence, contents and automatic detection of kl-satd, 2020.
112. A. J. Ratner, C. M. De Sa, S. Wu, D. Selsam, and C. Ré. Data programming: Creating large training sets, quickly. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3567–3575. Curran Associates, Inc., 2016.
113. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM.
114. E. Raymond. The cathedral and the bazaar. *First Monday*, 3(3), 1998.
115. S. Reddivari and J. Raman. Software quality prediction: An investigation based on machine learning. *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 115–122, 2019.
116. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
117. S. Romano, M. Caulo, G. Scanniello, M. T. Baldassarre, and D. Caivano. Sentiment polarity and bug introduction. In *International Conference on Product-Focused Software Process Improvement*, pages 347–363. Springer, 2020.
118. J. Rosenberg. Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*, pages 137–142. IEEE, 1997.
119. H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, Jan 1968.
120. S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt. Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Softw. Eng.*, 8(4):351–365, Dec 2003.
121. N. F. Schneidewind. Body of knowledge for software quality measurement. *Computer*, 35(2):77–83, Feb 2002.
122. B. Settles. Active learning literature survey. Technical report, University of Wisconsin–Madison, 2010.
123. M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering J.*, 3(2):30–36, Mar 1988.

124. E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 62:1–62:11, New York, NY, USA, 2012. ACM.
125. N. C. Shrikanth and T. Menzies. Assessing practitioner beliefs about software defect prediction. In *Intl. Conf. Softw. Eng.*, number 42, May 2020.
126. N. C. Shrikanth, W. Nichols, F. M. Fahid, and T. Menzies. Assessing practitioner beliefs about software engineering. arXiv:2006.05060, June 2020.
127. J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.
128. D. Spinellis. *Software Quality: The Open Source Perspective*. Pearson Education Inc., 2006.
129. I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Inf. Syst. J.*, 12(1):43–60, Jan 2002.
130. E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
131. S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*, pages 270–279, 2013.
132. E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.
133. E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002.
134. J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942, 2007.
135. B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA, 2015. ACM.
136. N. Walkinshaw and L. Minku. Are 20% of files responsible for 80% of defects? In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 2:1–2:10, New York, NY, USA, 2018. ACM.
137. E. Weyuker, T. Ostrand, and R. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13:539–559, 10 2008.
138. L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.
139. A. Wood. Predicting software reliability. *Computer*, 29(11):69–77, Nov 1996.
140. T. A. Wright and R. Cropanzano. Psychological well-being and job satisfaction as predictors of job performance. *Journal of Occupational Health Psychology*, 5:84–94, 2000.
141. S. Yamada and S. Osaki. Software reliability growth modeling: Models and applications. *IEEE Transactions on Software Engineering*, SE-11(12):1431–1437, Dec 1985.
142. A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.
143. A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, Jun 2011.
144. T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 73–83, Sept 2003.