Machine Learning Methodology for Software Development Improvement

Thesis submitted for the degree of "Doctor of Philosophy"

by

Idan Amit

Submitted to the Senate of the Hebrew University of Jerusalem

5/2024

Machine Learning Methodology for Software Development Improvement

Thesis submitted for the degree of "Doctor of Philosophy"

by

Idan Amit

Submitted to the Senate of the Hebrew University of Jerusalem

5/2024

This work was carried out under the supervision of

Prof. Dror G. Feitelson

Abstract

Software quality has a significant impact, so we are interested in improving it. This leads to two challenges - the definition of quality and identification of ways to improve it. Therefore, we developed a methodology for the representation of ill-defined concepts and identification of relations between them. We began with a definition of quality and in providing metrics that quantify it. First, we explored technical aspects of software development. We discovered large performance gaps between software projects. We found that many code properties affect performance, but only moderately. We noticed gaps similar to those in projects only when we studied developers. Therefore, we chose to focus on developers and their motivation as a leading factor. We found recommendations for increasing motivation and quantified the effect of motivation on performance in software development. Thus, we provide a way to improve performance in the software through improving motivation.

Now we will explain the work in more detail. Software quality has many aspects but is difficult to quantify due to its complexity and subjective nature (e.g., user satisfaction, giving due weight to different aspects). Sometimes software quality is assessed indirectly through other patterns that are generally accepted as indicating low quality, such as code smells or Self Admitted Technical Debt. It is also agreed that bugs are indicative of low quality, as evidenced by the extensive research on defect prediction. Bugs have been used as an indicator of quality in the past, in different variations. We developed the Corrective Commit Probability metric, the number of corrections normalized by the number of changes, so it is size agnostic and useful as a probability. The difference in CCP between projects is hundreds of percents.

During the search for factors that explain the quality differences between projects, we investigated the effect of feedback (e.g., code review) and the software structure (e.g., file length, coupling). We then investigated which types of refactoring reduce the probability of bug fixes. We also investigated over 170 types of static alerts.

We found that many best practices are not helpful, and that even the effect of the helpful ones among them was only moderate. To evaluate these effects, we used a reduction to supervised learning, which allowed us to leverage the work done in this area. We built models that use co-change, predicting *the change in the target* metric accord-

ing to *changes in the object variables*. We also used monotonicity, controls, and twin experiments in order to reduce the threat that the results are due to other factors.

Only when we examined developers we found large performance gaps, as in projects. Thus, a simple way to achieve better performance is by better developers. However, good developers are hard to find. An indication of this is the close levels of performance of all programmers, of graduates of prestigious universities, and of employees of prestigious companies. At the same time, performance is strongly influenced by motivation. Thus, our method to improve performance goes through improving motivation.

A survey of motivation research over seventy-five years indicates that the primary research tool is questionnaires, which suffer from many potential problems. The answers are subjective and not necessarily related to actual behavior. The questionnaire is relevant to a single point in time. The cost of finding respondents leads to limited scope and therefore limited reliability and resolution.

Contrary to the use of questionnaires, we wanted to investigate motivation using computer science methods — in a quantitative way, on the actual behavior, and based on large data, covering a long period of time. To do this we conducted a large survey, where we asked developers questions about motivation but also asked for their GitHub profile. The profile allowed us to link the answers to actual behavior.

In the survey we investigated eleven motivators from the literature. We identified validity problems such as moderate correlation between answers to related questions, moderate correlation in answers to the same question in the original and follow-up surveys, as well as self-promotion and errors in answers. Despite this, we found that the different motivators do predict high motivation. Comparing the original and follow-up surveys allowed us to discover that improvement in most motivators predicts improvement in motivation.

To explore motivation in GitHub's behavioral database, we need to differentiate motivated developers from others. We identify motivated developers using several labeling functions, such as working diverse hours and writing long commit messages. We verified that the labeling functions are weak classifiers for motivation by comparing them to answers about motivation in the survey. We did another verification using the GitHub database, where we tested the agreement between the labeling functions in general and the agreement in the co-change analysis.

After verifying that the functions are weak classifiers for motivation, we used them to investigate and quantify the relationship between motivation and performance. The difference in performance between motivated and unmotivated developers can reach hundreds of percents. This gap suggests that it is very beneficial to increase motivation. We also built a model predicting retention, identifying current developers that will stay in the project next year. The model had higher predictive power than the model using only our

control variables, indicating that the labeling functions have their own predictive power. The performance of the model was high, which makes the model valuable for itself.

The methodology we developed in this research is flexible and can be applied in many domains. It provides a way to measure fuzzy concepts that lack precise definition, such as quality and motivation. It also allows us to leverage different definitions of the same concept, compensating for the limitations of each individual definition. Given the definitions of the concepts, we can explore the relationships between them beyond the correlation, attribute changes to a person or a project, and predict the change in the target concept (for example, performance) given a change in the source concept (for example, motivation).

Contents

Abstract **v**

1 Introduction **1**

1.1 Positioning Relative to Related Work 4
1.2 Goals and Research Questions 5

2 Methodology and Research Infrastructure **9**

2.1 The BigQuery GitHub Schema 9
2.2 Metrics and Labeling Functions 10
2.3 Relation Identification 11

3 From Technical Aspects of Code to Developer Motivation **15**

4 Papers **19**

4.1 Which Refactoring Reduces Bug Rate? 19
4.2 Corrective Commit Probability: A Measure of the Effort Invested in Bug Fixing 24
4.3 A Large Scale Survey of Motivation in Software Development and Analysis of its Validity 70
4.4 Motivation Research Using Labeling Functions 116
4.5 Other Papers 127

5 Discussion and Conclusions **129**

5.1 Contributions 129
5.2 Future Work 130
5.3 Conclusions 131
5.4 Acknowledgments 132

References **133**

Chapter 1

Introduction

Today, one no longer needs to establish the importance of software. Due to the importance of software, the importance of high-quality software is derived. The impact of low quality is usually demonstrated by catastrophes, leading to immediate large loss. In our work, we point out that the every-day state of software development is also bad. The median software project spends 20% of the commits on bug fixes, the equivalent of dedicating a day in each week for rework.

However, quality is hard to define. A main contribution of our work is the ability to represent ill-defined concepts, like quality, yet we first review existing definitions. A common approach is using white-box code metrics like code length [51], McCabe's Cyclomatic Complexity (MCC) [54], Chidamber and Kemerer's metrics (CK metrics) [17], and others [38, 8, 61, 36]. While they tend to provide precise measurements, they need a justification of why and how they reflect quality. Some patterns are accepted by the community as bad: Self Admitted Technical Debt [60], code smells [25, 71], and static analysis alerts [57, 7]. An important advantage of the code level metrics is that they are usually actionable, and one can intervene and modify the unwanted pattern.

Another approach is product metrics, looking at the software as a black-box. Examples are software quality standards [42, 41] and properties like user satisfaction [43, 44]. Their benefit is that they represent the requirements from the software. Their downside is that the metrics might be subjective, hard to measure, and not applicable for white-box actionable insights.

The software development process is another source of valuable metrics. Typical metrics are the number of commits, the commit size, the number of contributors, etc. [35, 61, 55]. Process metrics have the advantage of showing where effort is being invested [55, 61]. Another advantage that is important to us is that process metrics provide a point of view that is external to the code. This facilitates an investigation of code patterns without the threat of circular referencing.

We wanted metrics that have a desired value associated with them, as in product

metrics, yet can be attributed to specific entities, as in code metrics. This is beneficial since even if we cannot identify the cause of the symptom, we can limit the area to investigate. We chose to use mainly process metrics. The application of these metrics is versatile, and they can be measured at the project, developer, and file level, in general or in a specific period. We developed some black-box metrics, like the Corrective Commit Probability (CCP), which measures the effort invested in bugs, and commit duration, measuring ease of modification. They are more related to the goal (e.g., less bugs), yet leave the way to reach the goal unknown. Other metrics are white-box and more actionable, like commit size, that measures coupling. As an example of the use of coupling, consider a pair of files where one provides a function and the other uses it. Assuming that the function should have been in the using file, the files will appear together in commits, indicating their coupling. Moving the function to the using file will reduce the coupling, and possibly the tendency for bugs resulting from the more complex structure.

Another contribution is development of a method for identifying beneficial recommendations. Assume that high coupling is a good predictor of high CCP. However, the predictive power might not be due to a causal relation, and the reason for the high CCP may not be the high coupling but a different one. In order to reduce this threat, we extended the analysis from mere predictive power. We used monotonicity to investigate if the concept increases with the classifier (e.g., the higher the coupling, the higher the CCP). We used controls to validate that the results are not due to confounders (e.g., coupling predicts CCP for different programming languages). We developed the method of co-change analysis to model the influence of changes (An increase in coupling from the previous year, predicts an increase in CCP).

Using this analysis, we found practical justified recommendations for quality improvement [5, 4, 3, 1]. An early model based on some of the patterns predicted the project CCP being above all projects' median CCP with an accuracy of 68%. Hence, while there is still a lot of uncertainty, our factors explain a significant part of CCP.

Different projects may have different CCP with a gap of hundreds of percents, which should be explained. However, each of the factors that we investigated improves CCP in only dozens of percents. We observed gaps of hundreds of percents in performance only when investigating developers (Chapter 3). Hence, instead of chasing technical features one might focus on well-performing developers. Though many talented developers exist, we choose to focus on motivation and not on talent. This is since we expected that motivation will be easier to influence.

Our goal was to develop a method that will allow us to leverage supervised learning tools for motivation research. Supervised learning requires labels of the concept, which are hard to obtain for motivation. This constraint puts us in the framework of weak supervision, where one is required to perform supervised learning, yet the labels are

limited in some way, from non existing, to few, noisy, etc. In order to cope with the problem, we built labeling functions, predictors of motivation, that are only required to be better than a guess. The labeling functions serve as a proxy for the motivation, allowing its investigation. In order to validate that the labeling functions indeed predict motivation we used a small set of labels that we collected in a survey.

In order to obtain labels about motivation, the survey contained questions about it. We also asked for the GitHub profile, a key that allowed us to match developers that answered the survey with the records describing their activity on GitHub. Our labeling functions are defined on the GitHub dataset, hence we could match them with the answers on motivation. We found dozens of developers in the matching of the survey and GitHub datasets, enough to validate functions yet not to learn them. We used the labeling functions as predictors of motivation and found that their accuracy and precision are better than a guess, above the level required from labeling functions. This allowed us to measure the influence of motivation on a large scale in natural activity.

After this small-scale initial validation, we used the labeling functions to validate each other on the large scale of the GitHub dataset. We validated that they co-change together, predict each other when the developer is factored out in twin experiments, etc.

We evaluated performance using the metrics that we developed in relation to software quality above. We evaluate motivation using our labeling functions. When examining the relation between motivation and performance, we investigated each metric with respect to all four labeling functions. This reduced the threat of an accidental result due to an artifact of one of the functions.

The survey that we conducted served us not only as a source of labels but also to investigate motivation itself. We asked about eleven known motivators and used them as predictors of high motivation. We did a follow up survey, asking the same developer the same questions on the same project, a year after. This allowed us to do co-change analysis and model how motivator improvement predicts motivation improvement. We also examined the validity of the answers, comparing related questions, the same question in both surveys, and answers and actual behavior. We found many amusing ways in which answers might be wrong. However, although the validity was moderate, the data was reliable enough to gain insights on motivation. We found out that the motivators predict motivation, and improvement in most of them predicts improvement in motivation. This led to recommendations about increasing motivation. Since motivation tends to increase performance, the motivation increasing recommendations are expected to increase performance too.

1.1 Positioning Relative to Related Work

Machine learning is widely used in software engineering. Here we position our work and explain the main difference between our approach and the common approaches. The related work of the specific investigations that we conducted appear in the section of each paper.

We will use defect prediction [75] as a working example of a typical use of machine learning in software engineering. The goal of defect prediction is to identify defective objects. The classified objects are simple (e.g., file, commit) and in typical research only one is used. The classification is usually binary, defective or not. Defects are identified using historical data, after they are fixed. The defect fixes themselves are usually identified by textual analysis of commit messages [63, 21, 37]. This data is used for training. The features used for modeling are usually code metrics. The framework is supervised learning and the goal is merely to predict whether a defect exists. The defect prediction use case assumes that the code owner who becomes aware of the prediction will modify the code in order to improve it — a causality framework.

Our work differs from defect prediction in several aspects. We use continuous and not binary metrics. This allows us to investigate stability, correlations, monotonicity, etc. We are interested in multiple concepts, such as effort invested in bugs, productivity, and bug detection efficiency. We examined multiple objects appearing in software development: not only commit and file, but also project, developer, and even a developer's activity in a project in a given year. Even when we were interested in a single concept (e.g., motivation), we used multiple metrics representing it, allowing us to investigate the relations between them.

There is a general agreement on the importance of software quality, yet disagreement about its essence, definition, and evaluation. In practice, it seems that the community agrees that low quality is indicated by bugs, code smells, static analysis alerts, and Self Admitted Technical Debt (SATD). They serve as a reference to other artifacts, though without exact quantification and estimation of reliability and validity. An alternative approach that was popular in early days was the quantitative evaluation of quality [12, 43], a path that we continue. The main difference between the early work and ours is that they considered the ability to automate metric computation as an advantage, while for us, who mine software repositories, this is a necessity. As agreed in the community, we use bugs as indicators of low quality. Our leading metric, Corrective Commit Probability, is essentially normalized bug count.

Software engineering deals in construction and ways to construct better. Some of the concepts that we consider are from the user point of view, seeing the software as a black-box, like requiring reliability. Others are from the developer point of view, looking for ways to improve the software structure. Works on technical debt, code smells, defect

prediction and more suggest a framework in which an indicator is found, *intervention is done*, and the situation is improved. This framework assumes causality. However, predictive analysis often just mentions the good old "correlation is not causality" and leaves the reader to decide whether to act upon the recommendations.

We aim to build stepping stones leading from mere correlation to causality. We would like to reduce the likelihood of providing a recommendation whose implementation does not lead to a benefit. But until we conduct intervention experiments, there is still a threat of attributing the benefit of other factors to the false recommendation. We reduce this threat by requiring additional properties beyond predictive power, like controls, monotinicity, and predictive power in co-change.

When we moved to the domain of motivation, the major difference between us and prior work was the methodology that we developed and used. Motivation has been investigated for years using questionnaires [53], interviews, and case studies. These methods allow direct input from the people involved, yet rely on self-reporting which is known as problematic, in a single point in time, and on a very small scale. An additional methodology is experiments which are exact yet usually investigate motivation in an artificial setting, and on an even smaller scale due to their cost.

Mining large datasets was found to be a powerful method in many domains. But investigating motivation by mining datasets such as GitHub requires the ability to represent it there. We provided a way to represent motivation on GitHub, *adding a new research tool that facilitates scalable, long term, high resolution, quantitative, reliable, and reproducible research at low cost.* Our motivation research was focused on OSS developers, differing from most work that investigated the general population. However, our method can be applied just as well to other populations, given datasets that represent their behavior. This is a significant contribution of our work that might benefit fields besides software engineering.

1.2 Goals and Research Questions

The attempt to improve software quality (e.g., pointing out harmful patterns [24]) is continuous and extensive. Our work is part of this effort. One can claim that a large portion of the work on software engineering is related to this effort. Different definitions for quality are discussed mainly in Sections 1.1 and 4.2.

Our specific goal was to answer "Which methods help to develop high quality software?". To answer it, we should first answer "What is software quality?" After that, we should investigate methods and answer "Does method X help to develop high quality software?".

We approached the question on a meta-level. Different people might desire different

properties of software (e.g., good running time performance, robustness to security vulnerabilities). We developed a methodology enabling defining metrics of such properties which are reliable and valid. We applied the methodology to define black-box properties such as low CCP, and white-box ones such as low coupling. This allowed us to answer the question "What is the quality of this software?" with respect to various properties.

People interested in a property of software are usually also interested in improving it. Being able to answer "Does method X *cause* an improvement in the property?" would be especially valuable for this need. But there is no canonical and comprehensive definition of causality. However, one can suggest partial models of causality, evaluated by the existence of certain attributes. We choose the following attributes, with examples using coupling improvement as the method and CCP improvement as the software quality metric:

- Single feature predictive power [65, 68, 46]: the ability to identify the desired concept. For example, high coupling predicts high CCP.
- Monotonicity [40, 70]: An increase in the concept (or its probability), given an increase in classifier. For example, the higher the coupling, the higher the CCP.
- Controlling [59]: have predictive power (regular or in co-change), even when controlling different variables, hence validate that the influence is not due to them. A special type of control is twins experiments, factoring out an entity (e.g., the developer). For example, high coupling predicts high CCP, for Java, Python, etc.
- Co-change [5]: an improvement in one metric is predictive of an improvement in another. For example, if the coupling in a project improved from the previous year, so is the CCP.
- Full reduction to supervised learning. Using many features and controls when building the model, both leveraging all features and considering many controls matching. For example, a model including coupling, programming language, and more code features predicts CCP change well, and better than a model without the coupling.

Monotonicity was suggested in Hill's seminal work [40]. Predictive power and controls only partially support causality (e.g., a relation that is due to two confounders might be misclassified). However, they are strong enough to filter out many non-causal relations [3]. Co-change allows to identify the direction of influence between variables: A causes B, B causes A, or A and B cause each other as in the case of multiple representations of motivation. A reduction to a full model allows us to consider relations between all factors. If we reach a minimal model with perfect accuracy (a very high bar), we are

assured that no other factor influences changes and that all variables are relevant (see Section 4.3).

This allowed us to answer questions like "Is method X likely to improve Y?" Setting Y to be "software quality" (as measured by our chosen metric) allows us to answer questions of the type "Is method X likely to improve software quality?", as we wanted.

When we applied our methodology, we found many methods that had a positive yet moderate influence. Hence, the gap in performance is large, yet there is no single silver-bullet method that can close it. But we noticed that developers and their motivation do exhibit large performance gaps (see Chapter 3). This led us to focus on the developer and motivation as a powerful factor in performance improvement. Hence, we used the same methodology to answer "Does improving motivation improve performance?" and "How can we improve motivation?".

Our methodology *enables* the use of machine learning in software engineering. In Section 4.1 enabling finding beneficial refactoring types instead of hypothesis testing. In Section 4.2 we provide metrics for quality, productivity, etc. that can serve as labels for machine learning, and a way to validate new metrics. In Section 4.3 we provide a way to predict co-change and show its relation to causality. In Section 4.4, we enable the representation of ill-defined concepts and the investigation of relations between them.

Chapter 2

Methodology and Research Infrastructure

We base our methodology on the reduction of problems to supervised learning. Most of the analysis was done on data from the BigQuery GitHub dataset. We defined metrics, such as CCP, and used them as the concepts in supervised learning in order to model which project properties predict them.

2.1 The BigQuery GitHub Schema

Our main dataset was GitHub data provided by Google BigQuery's GitHub schema [33]. The schema provides the commit history and latest code version of millions of repositories. Therefore, the domain of our work is mining software repositories (MSR).

It is important to note the dramatic influence of the use of this dataset on our work. For years, software engineering research analyzed the same few projects. This was not done due to laziness or indifference to the implications but since the cost of collecting the data was huge. Effort needed to access projects is lower today but finding a proper set of projects and collecting the relevant ones is still hard [45, 56]. We dedicated a lot of effort to filter out irrelevant projects — forks, non software, tiny, or not up-to-date [5]. After the filtering we ended with thousands of suitable unique large active software projects.

BigQuery is a relational database, enabling complex analysis in a few lines of SQL. It is trivial to group metrics by various dimensions, perform twin experiments by finding developers working in several projects and comparing them, and investigate metrics' change over time. Not only that implementing the analysis was easy, it was based on thousands of projects and their developers over multiple years, making the results very robust. While other relational databases could be used too, BigQuery already has the large dataset, provides the database as a service, and has high performance.

We also used the GitHub dataset to represent motivation and its influence. We used

surveys in order to collect developers' attitudes on motivation and build an auxiliary dataset. We matched the survey data with the GitHub data, based on the developers' GitHub profiles, allowing us to validate labeling functions for motivation. The labeling functions represented motivation on the GitHub dataset, allowing us to examine the performance metrics with respect to them.

2.2 Metrics and Labeling Functions

We wanted to base our analysis on big datasets of raw activities (e.g., a commit). Since in most cases the specific activity and the relations between them were not the areas we were interested in, we chose to aggregate them into metrics. A metric is a measurement quantification into a numeric value. Examples are the average number of files in a commit and the probability of a commit being a bug fix. On its own, if the metric is computed correctly, it is correct. However, metrics are usually computed to represent some property. In this case we should show that the metric indeed represents the property (e.g., the average number of files in a commit represents coupling).

In order to benefit from a metric, we should verify that it is valid and reliable enough for our needs. Reliability means being stable, returning similar results in similar conditions. We compared the measurements of an object (e.g., a repository) in close periods (e.g., adjacent years) [5]. We noticed that the various metrics are rather stable in a year gap, indicating reliability.

We verify validity by comparing the metric to an indication of the concept that the metric should measure. For example, we validated CCP, the ratio of commits that fix bugs, by comparing it with references to "low quality" in the commit messages, Self Admitted Technical Debt (SATD), and code smells. We also reproduced results (e.g., the longer the code, the more bugs), to validate that CCP agrees with prior work.

We would like to reduce the investigation of motivation to supervised learning. Supervised learning requires labeling of samples with the concept. In our case, we need to label the developers in the GitHub behavior dataset according to their motivation level. In principle, we could survey all GitHub developers and investigate their performance given their answers about their motivation. However, using our survey we could match the answers and behavior for only dozens of developers. Therefore, instead of limiting the data to only these labels, we looked for labeling functions, validated heuristics that are computable on the GitHub data and enable the labeling of additional developers.

For example, we took the distinct working hours metric. The hypothesis is that developers who work many diverse hours are more motivated than those who work 9 to 5. We turned this metric into a binary function by setting a threshold of working in at least 18 distinct hours of day during a year. We validated that this heuristic is a weak

classifier (a good predictor of motivation performing better than a guess [67, 10, 62, 6]) using the matched developers from the survey. We repeated this for three additional heuristics. Then we moved to represent the motivation of *all* developers using these four validated heuristics. We further used them to validate each other for predictive power, twins, and co-change analyses (Section 4.4). Given the validated heuristics, we moved to investigate the performance given motivation.

2.3 Relation Identification

Once we have ways to measure concepts, we are interested in finding relations between the concepts. Two relations are of special interest. The first relation is causal, like showing that reduction in coupling reduces CCP. Note that while it is hard to validate such relations, in many use cases causality is assumed. For example, the use case of static alerts is to find an alert, fix its reason, and by that *cause* an improvement in the code [3]. The second relation is showing that several metrics measure the same concept, as we did with the labeling functions for motivation. If they measure the same concept, we expect a change in the concept to lead to a change in all metrics. Therefore, a change in one of the metrics should be predictive of a change in the others.

Causality is not well defined so there are no exact requirements from causal relations. For example, the intuitive definition of Lewis [50] "An event E causally depends on C if, and only if, (i) if C had occurred, then E would have occurred, and (ii) if C had not occurred, then E would not have occurred." fails when (C_1 or C_2) causes E. Probabilistic frameworks, like ours, require probabilistic assumptions. Different causal models might lead to the same observational data, preventing a distinction between them. Some choose a model by adding structural constraints and preferences.

We cope with this by using a set of properties that we expect a causal relation to have, allowing us to filter out non-causal relations. The properties that we require are predictive ability, additional predictive ability given controls, monotonicity, and co-change.

We start examining correlative relations using predictive ability measures (e.g., accuracy, precision) [65, 68, 46]. Causality implies predictive ability (possibly in specific contexts), hence not having predictive ability filters out non-causal relations.

It is possible that a predictive ability will exist not due to the investigated variable but due to the influence of additional variables, confounders influencing both variables of interest. The simplest way to detect such influence is to use control variables. For example, if we suspect that the programming language influences the relation between coupling and CCP, we can evaluate the relation separately for Java, Python, etc. However, controlling a single variable cannot assure lack of a more complex influence (e.g., in new single-developer Java projects). Since the number of options increases exponentially

in the number of controls used simultaneously, this is not a feasible option. The VC dimension theory [74, 73] also warns us from false results when using too many tests on limited data. Instead, we use supervised learning trying to predict the target using the influencing variable and controls (Section 4.4). This allows us to verify that the influencing variable has additional predictive power and in which context the controls change the relation between the variables.

Attributing the property to the correct object is another aspect that we are concerned about. Given a project with high CCP, the reason might be due to the project (e.g., code base, architecture, review processes) or due to the developers (e.g., lack of experience). We can analyze observations in the spirit of twin experiments [72] in order to attribute the property. Given a developer that works in two projects, we factor out the similar developer [5]. Then we compare the probability that the developer will have a personal higher CCP, knowing the project has a higher CCP than its twin. The higher the predictive ability, the stronger the attribution to the project.

Note that twin experiments allow us to factor out common entities, with their possibly unknown factors. For example, each given developer has a specific number of years of experience. By considering the same developer activity, in the same year yet in two different projects, we control the developer. We therefore also control the developer's years of experience without having to know their number.

Monotonicity [40, 70] adds to predictive ability by requiring a step by step increase and not only a general increase in both variables together. This is since even when there is conditional dependency between the variables, the probability of having an increase in every step by mere chance is low.

Usually, causality is investigated in a single frozen state, for example between the properties of a project at a given time. In co-change analysis we leverage temporal data: we investigate project properties per year and see how a change in one metric is related to changes in others. If A causes B in a certain context, then a change in A will lead to a change in B in that context. If there is no causal relation (direct or indirect) between A and B, the probability of accidental co-change is low. Therefore, co-change analysis is important in relation identification.

An important extension of the co-change modeling is the move from a pair of variables, possibly with controls, to a full supervised learning model (Section 4.3). Suppose that we assume that the concept that we would like to model is a function of a known set of variables and having the Markov property [30]. In this case, a model can be built on the current state and the previous state (and the change in state as auxiliary variables) to predict the change in the concept. Such a model actually predicts what a change in the state will cause.

Such modeling deserves special attention when having two properties: high accuracy

and minimal size. Given perfect accuracy, we are assured of the nonexistence of any other intervening variable (see Section 4.3 for details). Minimality, as in achieving a decision tree of minimal size, assures us that all the variables in the model are required and influence the prediction. Achieving these properties is hard. Perfect accuracy might not be achievable with noise. Minimality in modeling usually requires solving NP-complete problems [19]. Achieving high accuracy using a minimal model is a common objective in supervised learning settings and is not unique to our work.

Chapter 3

From Technical Aspects of Code to Developer Motivation

Here we explain the path that we took from technical aspects of software development to motivation research. In short, the performance gaps observed between projects are high, but the influence of each single technical aspect is relatively small. Prior work identified large performance gaps between developers [66, 15, 14] and indeed this was the only area where we noted large gaps in our analysis. This hints that the developer should be the focal point for software improvement. It is known that motivation has a high impact on performance. Therefore, we decided to investigate motivation, its causes, and its impact.

The CCP, the effort invested in fixing bugs, is more than 6 times bigger in the lowest decile than in the top one. Hence, there is a lot to gain as a project improves its quality. However, the improvement might require many small steps. File length, the classical factor influencing tendency to bugs [58, 51, 28, 32] is easy to identify and in many cases easy to fix. Files shorter than 100 lines have low CCP that increases monotonically from that point [3]. On average, projects having short files and low coupling have CCP lower by 25% than the median [5]. The influence of reuse is also positive yet moderate [1]. Also, out of the 170+ static analysis alerts of CheckStyle [16], only a handful seemed to improve either CCP or commit duration, and their themes were simplicity, defensive programming, and abstraction [3]. Even for them, the influence was moderate. Brooks warned us that there is no "silver-bullet", a single factor that will improve software development by an order of magnitude [14]. Indeed, we too did not find any.

However, when observing the developers there are large performance gaps as in projects [22, 13]. Table 3.1 shows the gaps in developers' performance. They are large not only in the extremes, which could be dismissed as a rare phenomenon, but also when comparing the 25th and 75th percentiles to the median.

"Corrective Commit Prob." computes the effort invested in fixing bugs, "Code Coupling" is the average number of code files in a commit, "Test Presence" is the probability

of having a test file in a commit [5]. "Commit Duration" is the average time between a developer's commit to the previous commit in the same day [5, 1]. "Refactor Prob." is the probability that a commit is a refactor (Section 4.4).

"Single File Fix Prob." is a metric that was not discussed in our papers, so we explain more about it here. It measures the probability that a bug fix will involve a single code file, possibly with a test file, indicating cohesion. It has 0.64 Pearson correlation in adjacent years, showing good stability. When using "Single File Fix Prob." to predict "Interface Involvement", the precision lift is 19%, showing agreement between different aspect of abstraction.[1].

Table 3.1: Developer Performance Spread

Metric	10%	25%	50%	75%	90%
Corrective Commit Prob.	0.05	0.12	0.21	0.32	0.45
Code Coupling	0.12	0.82	1.33	1.83	2.32
Test Presence	0.00	0.01	0.08	0.22	0.36
Commit Duration	41.37	59.17	85.71	124.49	180.45
Refactor Prob.	0.01	0.07	0.14	0.24	0.35
Single File Fix Prob.	0.08	0.29	0.43	0.56	0.68

Brooks suggested employing high quality developers as a simple way to reach high quality projects [14]. However, the ability of prestigious companies to hire such developers and the ability of universities to train them is limited. Table 3.2 shows the average performance of developers from select organizations, normalized by the value of the "Other" group. Attribution to affiliation was done by the developer's email address. We considered affiliations with at least 10 developers doing at least 200 commits each in a year. Though performance might differ significantly by affiliation, the diversity is lower than in the entire population (represented by the results in Table 3.1). There are high differences in test presence, yet it is more a cultural norm than a skill. Low code coupling and high single file fix probability are desirable and tend to be less influenced by project properties. Out of all the affiliations, only RedHat and MIT show these properties (indicating good decomposition to components), yet in very low values.

Experience improves developer skill moderately. Figure 3.1 shows the average performance of the same developer in successive years, compared to the performance of the same developer in the first year in that project. The highest difference is lower than 20% after

[1] Abstraction can be defined as a good separation between interface and implementation. We developed "Interface Involvement" in order to capture this concept. The programming languages C and C++ have separate files for interface (header) and implementation. The convention is to use the same file name with different extensions (e.g., 'math.h' for interface and 'math.c' for implementation). Given a good abstraction, we expect that a change in the implementation will not require a change in the interface. We therefore measure the probability that the interface file will be modified given that the implementation file was modified. Low probability indicates high abstraction. Its adjacent year Pearson correlation is 0.69.

Table 3.2: Performance by Affiliation, relative to "Other"

Affiliation	Developers	Corrective Commit Prob.	Code Coupling	Test Presence	Commit Duration	Refactor Prob.	Single File Fix Prob.
Amazon	52	0.71	1.31	1.29	1.69	0.83	0.85
Apache	366	1.15	1.14	1.79	1.26	1.12	0.95
Apple	90	1.87	1.20	2.67	1.95	1.71	1.13
Facebook	200	1.36	0.86	2.14	1.95	2.14	0.87
Google	1,677	1.91	1.06	1.72	1.67	1.41	0.91
IBM	62	0.61	1.06	1.60	1.60	0.98	0.88
Intel	122	0.86	1.13	0.80	0.91	2.46	1.39
Microsoft	320	1.31	1.09	2.03	1.50	0.91	0.82
RedHat	657	1.11	0.99	1.30	0.99	1.52	1.14
Cornell	29	0.56	1.09	0.61	0.94	0.96	1.22
Inria	38	1.01	0.66	1.10	0.85	1.25	0.64
MIT	32	1.12	0.99	1.21	1.19	0.99	1.03
Stanford	28	0.69	1.09	1.14	1.27	0.70	0.98
Uni. Michigan	28	0.96	0.94	1.14	1.04	0.87	0.89
Other	39,575	1.00	1.00	1.00	1.00	1.00	1.00

7 years. Only test presence and code coupling show a rather monotonic improvement, and both can be improved with discipline even without an exceptional skill.

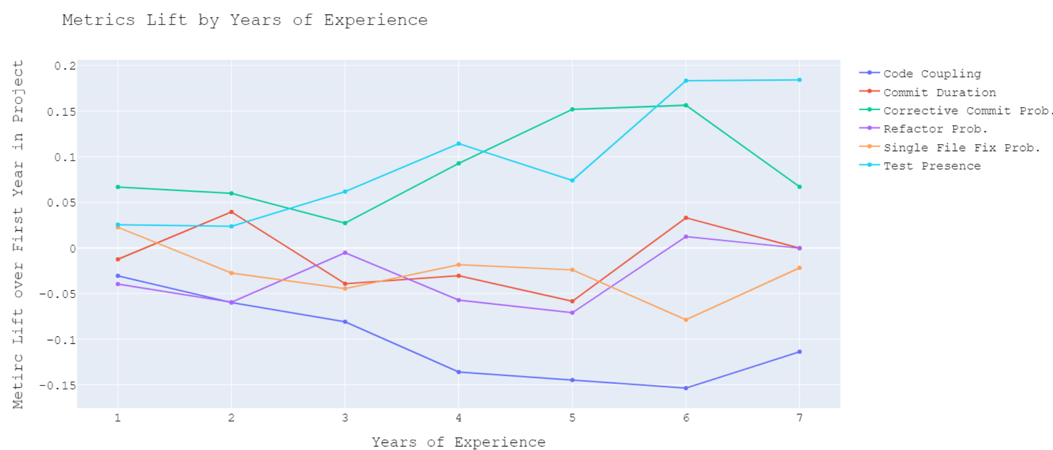


Figure 3.1: Relative performance of a developer in a given year, compared to the developer's performance in the first year in the project.

The insights regarding developers are not new. Some even see human problems as more important than technological ones [23, 26]. Specifically, many have investigated motivation as an important factor in software engineering [49, 9, 27, 31]. Therefore, we moved to investigating motivation.

Additional support for the importance of motivation came from our survey (Section

4.3). 73% of the participants answered that motivation has more influence on their productivity than skill (answers higher than neutral), and only 9% answered that their skill is more influential.

We examined 11 motivators appearing in the literature and their relation to motivation. In general, motivators indeed predict high motivation and improvement in most of them predicts motivation improvement. Hence, they provide ways to improve motivation. Note that none of them was sufficient or required, hinting that different individuals are motivated due to different reasons. We extended our investigation from quality to performance in general to capture more influences of motivation (e.g., higher retention). The impact of motivation on performance metrics was high, and these results were obtained with four different labeling functions, increasing validity. These results are obtained on developers in their actual work, not from lab experiments, probably exhibiting more representative behavior.

Chapter 4

Papers

4.1 Which Refactoring Reduces Bug Rate?

Published: Amit, Idan, and Dror G. Feitelson. "Which Refactoring Reduces Bug Rate?" Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering. 2019.

In "Which Refactoring Reduces Bug Rate?" [4] we checked which types of refactoring, identified by the description in the commit message, reduce CCP. We used the commit message to extract refactoring types by linguistic themes. This led to diverse types referring to different aspects of the refactoring, like the reason for the refactoring (e.g., 'TODO'), the programming mechanism involved (e.g., enum), the goal (e.g., reuse), or the process (e.g., using feedback).

Lehman's laws of software evolution imply that quality may have a negative correlation with the age of a project [47, 48]. Most refactoring types only reduce the rate of degradation but do not avoid it. This was the first indication in our research that no single activity is enough to improve quality. Also, refactoring that removes Self Admitted Technical Debt (SATD) [60] (e.g., 'TODO' comments) is recommended. SATD provides feedback by the developer on the code. Refactoring mentioning feedback (e.g., code review or pylint) have higher probability to be beneficial, regardless of content. Hence, feedback was found to be an important mechanism in software improvement.

This paper was the first to be published. It was published during the work on CCP, when the term was not coined yet [5]. The paper investigates changes in bug fix rate, which is equivalent to CCP.

Which Refactoring Reduces Bug Rate?

Idan Amit
idan.amit@mail.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

Dror G. Feitelson
feit@cs.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

ABSTRACT

We present a methodology to identify refactoring operations that reduce the bug rate in the code. The methodology is based on comparing the bug fixing rate in certain time windows before and after the refactoring. We analyzed 61,331 refactor commits from 1,531 large active GitHub projects. When comparing three-month windows, the bug rate is substantially reduced in 17% of the files of analyzed refactors, compared to 12% of the files in random commits. Within this group, implementing ‘todo’s provides the most benefits. Certain operations like reuse, upgrade, and using enum and namespaces are also especially beneficial.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

KEYWORDS

Code quality, refactoring, machine learning

ACM Reference Format:
Idan Amit and Dror G. Feitelson. 2019. Which Refactoring Reduces Bug Rate?. In *The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE’19), September 18, 2019, Recife, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3345629.3345631

1 INTRODUCTION

Our goal is to find actionable recommendations for software quality improvement. Refactoring is an established approach to improving code quality and promoting maintainability and continued development [6]. Many different refactoring operations have been proposed. But which ones provide the best benefits?

Refactors are change operations intended to improve code quality without changing functionality. We use a methodology that analyzes changes over time. The metric we use is bug fixing rate: Out of all the commits in a given period, what fraction are bug fixes. We identify corrective and refactor commits using linguistic models applied to the commit message. Once we identify the refactors, we compare the bug-fix rate as reflected by corrective commits before and after the refactor.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE’19, September 18, 2019, Recife, Brazil
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7233-6/19/09…$15.00
https://doi.org/10.1145/3345629.3345631

Note that even when we see an improvement around a change, it still doesn’t imply causality, and we cannot be sure that the operation is recommended. The change might be accidental, or due to other reasons. However, this methodology enables us to explore, identify candidates, and follow up with focused experiments.

Refactors are 12% of the commits in large active GitHub projects, but not all of them are equally suitable for analysis. Some don’t have enough other commits around them, some involve a huge number of files, and some are not done on identified source code. In order to explore and identify effective refactors, we built a dataset of ‘clean’ refactors—commits of one non-test file with enough context—and their messages. These commits were labeled as positive if they reduced the bug-fix rate substantially. This reduces the problem to supervised learning, where we can explore the attributes that characterize commits that reduce the bug rate.

1.1 Related Work

Refactoring, first suggested by Opdyke [15], is “improving the design of existing code” [6]. Prior work investigated the influence of refactoring on quality [1, 3, 4, 8, 14, 17, 19]. In general, this research showed mixed results of positive and negative influence, of small size. While prior work asks “Does refactoring have a positive influence?”, we ask “Which refactor operations have a positive influence?”, enabling us to identify such operations.

To check the influence of refactoring one has to identify refactoring. Most work focused on the refactor technique. Tsantalis et al. [18] developed RMiner for Java which has very high performance: 98% precision and 87% recall. However, they built an Abstract Syntax Tree of the code in order to identify changes. We did the analysis in Google’s BigQuery GitHub schema where the code per commit is not available. Other approaches are manual labeling, which is limited in scope, or Ref-Finder [9] with precision of only 35% [16]. We identify refactors and corrective commits using linguistic analysis of commit messages, an idea that is commonly used for defect prediction [7].

1.2 Our Contribution

We present a way to find bug-rate reducing operations. We provide scalable (analyzing millions of commits) accurate models to identify corrective and refactor commits, based only on their messages. We are agnostic to the refactor technique used and therefore can explore and identify new efficient operations. We provide empirical evidence for code rot [5, 10], and show that refactoring slows the decay rate. This is a demonstration of Lehman’s 7th law of software evolution, which states that software quality will appear to decline unless it is rigorously maintained [11]. Our conclusions recommend on refactor operations that improve quality.

2 CHANGE IDENTIFICATION AND SCOPE

2.1 Linguistic Models for Commit Types

There are 10,845 open source projects in GitHub with 500+ commits in 2018. We analyzed the 1,531 non-fork projects [2], since forks are very close to their origin, are redundant, and distort the distribution.

We sampled uniformly random commits from the large active projects. We labeled the commits following the taxonomy of Lientz et al. [13] into corrective, adaptive, and perfective. We also labeled refactor as a sub type of perfective. We then built a linguistic model for commit messages, identifying positive occurrences and removing invalid ones (e.g., due to negative context, modals, or being part of a different unrelated term). We used a test set of 1,100 samples on which we evaluated the classifiers. The corrective model has accuracy of 89%, precision of 84%, and recall of 69%. For the refactor model, the accuracy is 93%, precision is 80%, and recall is 61%. For our refactor use-case we need high precision. Moreover, knowing the precision value enables us to estimate the real influence, considering that 20% of the hits are not refactors. Our analysis doesn't depend directly on the recall, yet the higher the recall, the more cases we will have. See supplementary materials for more details.

2.2 Building a Refactors Dataset

Using the models we identified 749,603 commits as refactors. The commits differ by their suitability to change analysis. A refactor might be subject to noise due to few commits in context and application to many files. We look for clean, less noisy, commits. Yet, the more restrictive the definition of a clean refactor, the fewer cases we have to analyze. Hence, we are in a tradeoff between quantity and cleanness.

The following describes the 176,309 refactors done during 2018, on which we based our decisions regarding the scope of refactors to analyze and their context. The full distributions and generating code are part of the supplementary materials.

First, we are interested only in refactoring done on source code. We used the 28 major source code filename extensions, which cover 49% of GitHub files. Out of all the (file, refactor commit) pairs, 57% belong to a major source code extension.

Another set of files that we would like to scope out are test files, since the goal and behavior of tests is different from regular code. Moreover, a refactor involving a test is likely to refactor the tested file and not the test file. Since the code is not available in BigQuery, we identify test files by matching their path with the pattern 'test'[12, 20], which matched 21%.

We define the context of a refactor as the commits made to the same file before and after the refactor. In order to identify a change, we need at least one commit before and after the refactor, and more are needed for a robust analysis. In a context that spans three months, 57% had fewer than 10 commits before the refactor and 56% had fewer than 10 commits after it. In a context that spans six months the figures were 56% and 48%. Aiming for clean results we examined the 61,331 refactor commits with at least 10 commits in 3 months on either side, and involving a single non-test file.

3 REFACTOR INFLUENCE ANALYSIS

We first explain the methodology and then present results.

3.1 Linguistic Identification of Useful Refactors

Refactors are diverse in type, size, goal, subject, and implementation. As we show below, their effect on bug rate has a high standard deviation. This hints that there might be different types of refactors with different effects.

In order to identify the most effective refactors, we retrieved the commit messages from our refactor dataset and tokenized them. We labeled them by whether the refactor reduced the bug rate by at least 0.1. This enabled us to use machine learning to predict improvement and evaluate influence. However, the number of tokens is very high, the same semantic meaning might be represented by different tokens, and all probabilities are subject to noise. Therefore, we used linguistic and domain knowledge to map the tokens into groups, e.g. considering 'reorganizes', 'reorganized' and 'repackaging' as members of the 'reorganize' group.

3.2 Using Coupling as a Metric

Linguistic analysis as described above is based on what the developer intended to do in the refactor, and documented in the commit message. But the refactor might fail to fulfill the intent or have other side effects. It is therefore interesting to also look at the effect of software properties that just happened to change. We use coupling, which is a metric of software quality, and examine the influence of a reduction in the coupling on the bug-fix rate.

Zimmermann et al. [21] showed that co-changes analysis, namely files that change in the same commit, can be used to detect coupling. We use the idea as a file-level metric for coupling based on the size of co-changes. A commit is a unit of work ideally reflecting the completion of a task. It should contain only the files relevant to that task. A large number of files needed for a task means coupling. Therefore, the metric is the average number of files in a commit.

The same approach can be applied to other software metrics such as length, readability, and complexity.

3.3 Influence of Refactoring

In order to analyze the influence of refactors, we should first know what happens without them. We compared the bug-fix rates of files in two adjacent three-month windows and saw that only 42% of the files had a lower bug-fix rate in the later window. The average difference in the bug-fix rate is only 0.004, with standard deviation of 0.2. Probability of improving by 0.1 or more was 12% (with 10 commits, 0.1 means one commit). These results indicate code rot, yet show that code quality decreases slowly and with a variance that is much larger than the change.

We present the influence of refactors in table 1. We examined clean refactors with at least 10 commits in a three-month window before them and in two such windows after them. The requirement for the second window reduces the dataset by about 35%.

The average change in the bug rate is small for all refactor types, and in all but one it is positive (an increase in bug rate). However, the standard deviation is large, so there are many cases where the refactoring does help. The results are sorted according to the probability of a substantial reduction in the bug rate (emphasized), where "substantial" is taken to be at least 0.1 in a 3-month context. In all

Table 1: Effect of refactor operations on bug rate

	Metric:	difference	probability to improve			
		Avg±stddev	by >0.1	by >0	by >0	by >0
	Window before:	3mon	3mon	3mon	3mon	6mon
	Window after:	3mon	3mon	3mon	4-6mn	6mon
Action	Commits					
todo	256	-0.003±0.137	**0.248**	0.493	0.523	0.500
feedback	149	0.005±0.137	**0.228**	0.479	0.407	0.419
reuse	122	0.027±0.136	**0.211**	0.398	0.451	0.511
upgrade	79	0.002±0.134	**0.205**	0.446	0.566	0.506
SE goals	995	0.003±0.137	**0.205**	0.485	0.527	0.506
sp/tab	358	0.000±0.132	**0.202**	0.486	0.469	0.499
improve	3,193	0.005±0.135	**0.196**	0.482	0.484	0.486
optimization	1,141	0.005±0.135	**0.196**	0.465	0.475	0.457
refactor	4,241	0.006±0.133	**0.186**	0.465	0.505	0.489
enum/ns	261	0.004±0.121	**0.184**	0.494	0.494	0.523
unneeded	7,167	0.001±0.129	**0.180**	0.501	0.496	0.502
rework	850	0.013±0.133	**0.178**	0.449	0.466	0.401
baseline	**39,944**	**0.005±0.127**	**0.172**	**0.475**	**0.486**	**0.476**
reorganize	423	0.013±0.144	**0.168**	0.410	0.487	0.462
rename	2,492	0.003±0.123	**0.168**	0.463	0.481	0.459
simplify	2,605	0.004±0.122	**0.164**	0.473	0.471	0.461
clean	595	0.009±0.134	**0.162**	0.460	0.482	0.482
coupling	10,180	0.028±0.133	**0.135**	0.403	0.447	0.459
style	196	0.036±0.137	**0.132**	0.397	0.470	0.420
spelling	47	0.046±0.128	**0.080**	0.380	0.500	0.360

cases but one this was larger than the 12% found for general commits. The next column shows the probability for *any* improvement, and the last two consider alternative time windows. "Commits" is the number of refactors we analyze. Note that some of the results are based on a small number of cases.

The rows of the table represent the refactoring operations. The baseline row (emphasized) presents the influence of a general refactor. Operations that may require explanation are 'feedback' - with external feedback (human as code review or mechanical as pylint), 'enum/ns' - software engineering constructs like enum or namespace, 'SE goals' - explicit reference to a software goal like abstraction, 'unneeded' - removing unneeded code, 'sp/tab' - the whitespace/tabs wars common in the Python community, 'refactor' - explicitly mentioning the term. 'Coupling' is refactors that had a side-effect of reducing the number of non-test files in the commit by at least one file. The rest of the operations are a textual evidence of the name of the operation and its semantically equivalent terms. A refactor can involve several operations.

Only one operation had a probability for improvement of 50% in the next three months. However, in 75% of them the probability of improvement was better than the 42% expected for general commits. Observing the difference average and standard deviation it is clear that all changes are small, usually reducing the quality and with a high variance. This explains the mixed results in prior work. None of the operations is a silver bullet or even reaches 60% probability of improvement and a substantial reduction in bug rate.

Another result is that four to six months after the refactor many operations are better than after one to three months. This might suggest that a refactor disrupts the system and might cause more bugs in the short term. But good refactors have a return on investment in the long term, and for some operations the six-month bug rate shows improvement.

3.4 Possible Explanations

Once we can identify influential operations, the first question that comes to mind is "Why does it influence in this way?" We don't claim to provide answers here but to suggest ideas that should be investigated on their own.

The probability of reducing the bug rate by 0.1 between two 3-month periods is 12%. Doing a refactor raises the probability to 17%. The leading operation is 'todo', though based on a small number of cases. It is interesting to further investigate why it is so influential. A possible reason is that 'todo's actually reflect what the developers themselves think should be done, based on their knowledge of the code, and it is indeed advisable to act on this input.

'Feedback' and 'SE goals' indicate a change that is supported by an external influence and guided. Maybe the identification of a proper target contributes to the success. 'Feedback' is not a specific change but a result of consulting, and its substantial improvement probability is 32% higher than that of a baseline refactor.

'Reuse' and 'upgrade' should have been a free lunch, using an already-existing component. However, the change itself led to short term bugs and returning the investment only in the longer term.

The influence of 'improve' is lower, suggesting that we might not be as good as we think in identifying needed improvements. This is even more so with 'clean' and 'reorganize', which are slightly less beneficial than a general refactor, and 'rework', which is perhaps marginally better.

Simplification is one of the most advocated principles in software engineering and in general. One would expect that the influence of simplification would be high, but our results indicate it is lower than a general refactor. It will be interesting to investigate if simplification refactors indeed improve complexity metrics.

Interestingly, other than 'rename' and 'unneeded', we didn't identify the linguistic expression of any refactor technique (e.g., 'extract method', 'pull up/down'). This might suggest that the context and goal, and not the technique, are the cause of influence. A focused study is needed to verify this.

Reduced coupling is slightly worse than no refactor in the first 3 months, and improves later. This might hint that a large reduction in coupling is somewhat positive but involves destabilizing the system.

3.5 Influence on the System and Influence of the Developer

The linguistic exploration led to some surprising results. Consider the removal of unneeded code, whitespace wars, style, rename, and spelling. The compiler is indifferent to all of them, yet we see that they have an effect. Hence, they influence via the developer, who might make more bugs, e.g. due to confusing variable names.

The terms of whitespace wars, style, and spelling were not part of our definition of a refactor. But their influence was big enough to be observed in our refactor model hits. It is also possible that developers who pay attention to style, pay attention to quality in

general. In order to investigate their influence regardless of our refactor model, one should develop a model for them and consider all their occurrences.

4 THREATS TO VALIDITY

We checked differences in bug-fix rates before and after a refactor as a measure of quality improvement. The time windows used might be too long or too short. Since our context is rather long, it is possible that different causes interfere with the change, a threat we try to control by using a large number of cases.

The use of bug fixes might be misleading in case the refactor does not lead to more bugs but helps to find bugs more efficiently. For example, simplifying the code might reveal old bugs and increase the future bug-fix rate.

We don't know how a developer decides to do a refactor. However, it is not uniformly selecting a random file each day. The refactors done depend on the history and goals, and this might bias our data. If one decides to refactor a file after a period that had many fixes by chance, the regression towards the mean might be seen as an improvement. External causes, like a quality assurance blitz, might also change the probability of finding and fixing a bug.

In order to analyze the refactors we needed a suitable context. The refactors in the scope, and even more the clean refactors, are a small part of all the refactors. A refactor done on an extensively changing file might not represent other refactors. Finding recommendable operations using clean commits and verifying them on all commits will enable validation of the results.

The linguistic models for corrective and refactor commits were built and estimated using labeled data sets. The models reach high precision by not only identifying a term occurrence but also noticing that "error message" and "this is a feature and not a bug" do not indicate a corrective commit. While refactor operations are part of the model, we don't have a labeled data set and performance evaluation for them. A text occurrence is usually positive, but validation is needed.

5 FUTURE WORK AND CONCLUSIONS

We presented a method to identify refactor operations and evaluate their effectiveness in reducing the bug rate. We provide results on known refactoring operations and identify new ones.

In general, refactors are instrumental in reducing bug rates due to code rot. A recommendation depends on the metric of interest. Almost all refactor operations have a better probability for a substantial improvement or improvement after four to six months than a general commit. Being conservative, we recommend operations with at least 50% for improvement in the next six months: do your 'todo's, remove unneeded code, aim to improve software goals, upgrade and reuse, and use enums and namespaces.

Many other metrics can be used as both the changed metric in the refactor or the target metric we value. Some of them have immediate influence (e.g., a refactor that shortens a file length), making the change cleaner and the analysis results more robust. Many other aspects might influence the effectiveness of refactoring: developer's familiarity with the code, experience, file age, etc. The high variance of influence gives hope to finding more effective operations. Using this method, and further developing the method itself, can lead to more actionable recommendations.

Supplementary Materials

See https://github.com/evidencebp/Which-Refactoring-Reduces-Bug-Rate

Acknowledgements

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 832/18).

REFERENCES

[1] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and Software Technology*, 51(9):1319–1326, 2009.
[2] 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 *6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, May 2009.
[3] B. D. Bois and T. Mens. Describing the impact of refactoring on internal program quality. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications*, pages 37–38, 2003.
[4] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta. On the impact of refactoring operations on code quality metrics. In *IEEE International Conference on Software Maintenance and Evolution*, pages 456–460, Sep. 2014.
[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, Jan 2001.
[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2018.
[7] 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.
[8] S. H. Kannangara and W. M. J. I. Wijayanayake. Impact of refactoring on external code quality improvement: An empirical evaluation. In *Intl. Conf. Advances in ICT for Emerging Regions (ICTer)*, pages 60–67, Dec 2013.
[9] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 371–372, 2010.
[10] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. & Softw.*, 1:213–221, 1980.
[11] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, number 5, pages 108–124. Springer Verlag, Oct 1996. Lect. Notes Comput. Sci. vol. 1149.
[12] S. Levin and A. Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *IEEE International Conference on Software Maintenance and Evolution, (ICSME)*, pages 35–46, 2017.
[13] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Comm. ACM*, 21(6):466–471, Jun 1978.
[14] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In B. Meyer, J. R. Nawrocki, and B. Walter, editors, *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer Berlin Heidelberg, 2008.
[15] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, 1992.
[16] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86:1006–1022, 04 2013.
[17] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, WoSQ '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.
[18] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, 2018.
[19] D. Wilking, U. F. Kahn, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1(1):27–42, 2007.
[20] 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.
[21] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *6th Intl. Workshop Principles of Software Evolution*, Sept. 2003.

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

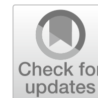
Published: Amit, Idan, and Dror G. Feitelson. "Corrective commit probability: a measure of the effort invested in bug fixing." Software Quality Journal 29(4):817-861, 2021.

In this work [5] we presented metrics for software development. The main metric of interest is the corrective commit probability (CCP), which reflects the effort invested in fixing bugs. We also introduced commit duration for productivity, commit size for coupling, and time to revert and time to fix a bug for bug detection efficiency.

We were surprised to note that despite the extensive work on software quality and software development, there are no metrics suitable for our methodology of improvement investigation. We looked for reliable metrics, that predict well a desired concept, and can be computed in different contexts (e.g., for a developer, or for a given project in a given year), enabling actionable intervention based on them. The desired concepts, such as quality or productivity, are not only not well defined but also a source of many disagreements. We cope with this by choosing metrics with value of their own (e.g., less investment in fixing bugs, smaller commits) that are aligned with other agreed measures of the concepts (e.g., code smell, references to coupling).

We also had to develop methodologies in order to justify our recommendations. We introduced twin experiments in order to factor out developers. We introduced co-change analysis, in this paper limited to a classifier, a concept, and a single control variable, in order to investigate response to a change. These both reduce the threat of identifying false relations and are a step towards establishing causality.

Software Quality Journal (2021) 29:817–861
https://doi.org/10.1007/s11219-021-09564-z



Corrective commit probability: a measure of the effort invested in bug fixing

Idan Amit[1,2] · Dror G. Feitelson[1]

Accepted: 13 June 2021 / Published online: 5 August 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

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

1 Introduction

Software quality is an important area in software engineering (Spinellis, 2006; Baggen et al., 2012; Stamelos et al., 2002; Ray et al., 2014; Corral & Fronza, 2015). Despite decades of work on this issue, there is no agreed definition of "software quality". For some, this term

✉ Idan Amit
idan.amit@mail.huji.ac.il

Dror 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

refers to the quality of the software product as perceived by its users (Schneidewind, 2002; Hackbarth et al., 2016). Others use the term in reference to the code itself, as perceived by developers, for example as reflected by code smells (Fowler et al., 1997; Van Emden & Moonen, 2002; Yamashita & Moonen, 2012; Khomh et al., 2009; Taba et al., 2013). Non-functional properties of code (e.g., reliability, modifiability) were also used as quality measures (Boehm et al., 1976). Others include correctness as the foremost property (Dromey, 1995). 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 (Lientz, 1983). In fact, corrective maintenance (aka fixing bugs) may represent a large fraction of software development, and may contribute significantly to software costs (Lientz et al., 1978; Schach et al., 2003; Boehm & Papaccio, 1988). 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 (Boehm, 1981; Boehm & Basili, 2001; Boehm & Papaccio, 1988; Dawson et al., 2010; Hackbarth et al., 2016). 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 1-week project and a 20-year 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 (Lientz, 1983). The literature contains vastly varying reports regarding the relative fraction of corrective maintenance, from 17.4% to 56.7% on average (Lientz et al., 1978; Schach et al., 2003; Latoza et al., 2006). 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 (Sect. 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 (Sect. 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 (Sect. 7). High coupling (Sect. 7.3), and file length (Sect. 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 (Ray et al., 2014; D'Ambros et al., 2010; Hall et al., 2012). The linguistic-model prediction marks commit messages as corrective or not, in the spirit of Ratner et al. labelling functions (Ratner et al., 2016). 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 (Hindle et al., 2009; Amor et al., 2006; Levin & Yehudai, 2017; Antoniol et al., 2008; Amit & Feitelson, 2019).

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 (Bird et al., 2009)). 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 (Sect. 2), explain the construction of CCP (Sect. 3), and in Sect. 4 presents our experimental methodology. The distribution of CCP is presented in Sect. 5, and the effect of detection efficiency on CCP in Sect. 6. Sect. 7 covers the interaction of CCP with coding practices, and Sect. 8 the relation with the project profile. We then check to which extent the discussed factors cover CCP in Sect. 9. We end with threats to validity in Sect. 10 and conclusions in Sect. 11.

2 Related work

The number of bugs in a program could have been a great quality metric. However, Rice's theorem (Rice, 1953) 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 (Jones, 1991, 2012). 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 (ISO, 2011). 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 (ISO, 2001) (the precursor of ISO/IEC 25010) found it to be ineffective in identifying design problems (Al-Kilidar et al., 2005).

Code metrics measure properties of the source code directly. Typical metrics are lines of code (LOC) (Lipow, 1982), the Chidamber and Kemerer object-oriented metrics (aka CK metrics) (Chidamber & Kemerer, 1994), McCabe's cyclomatic complexity (McCabe, 1976), Halstead complexity measures (Halstead, 1977), etc. (Basili et al., 1996; Rahmann & Devanbu, 2013; Gyimothy et al., 2005). They tend to be specific, low level and highly correlated with LOC (Shepperd, 1988; Rosenberg, 1997; Gil & Lalouche, 2017; Molnar et al., 2020). Some specific bugs can be detected by matching patterns in the code (Hovemeyer & Pugh, 2004). 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. (Graves et al., 2000; Rahmann & Devanbu, 2013; Moser et al., 2008). 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 (Moser et al., 2008; Rahmann & Devanbu, 2013).

Working with commits as the entities of interest is also popular in just in time (JIT) defect prediction (Kamei et al., 2013). 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 (Bird et al., 2009) and reduce the CCP computation to mere counting. Yet, the links between commits and tickets might be biased (Bird et al., 2009). The ticket classification itself might have 30% errors (Herzig et al., 2013), 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 analyzing 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 (Ray et al., 2014; D'Ambros et al., 2010; Hall et al., 2012).

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) (Wood, 1996; Graves et al., 2000; Yamada & Osaki, 1985). 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 (Khomh et al., 2012; Reddivari & Raman, 2019), per period (Vasilescu et al., 2015), and per commit (Shibab et al., 2012; Amit & Feitelson, 2019). 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 (Bird et al., 2009), 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 (Swanson, 1976), 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 (Herzig & Zeller, 2013; Herbold et al., 2020), 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 (Cohen, 1960) among the different annotators were at least 0.9, indicating excellent agreement. Similarly consistent commit labeling was reported by Levin and Yehudai (2017).

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

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 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' (Hattori & Lanza, 2008; Ray et al., 2014). 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 (Hawkins, 2004). 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

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

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. (2014) (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 (2019), reaching an accuracy of 89%. We then added additional terms from Shrikanth and Menzies (2020) (variants of 'proper', 'broke', 'vulnerab', and 'defect'). We also used labeled commits from Levin and Yehudai (2017) to further improve the model based on samples it failed to classify.

The last boost to performance came from the use of active learning (Settles, 2010) and specifically the use of classifiers discrepancies (Amit et al., 2017). 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 (2019) 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 constructed in (Amit & Feitelson, 2019) are presented in the confusion matrix of Table 1.

Supervised learning metrics shed light on the common behavior 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 Sect. 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% (Hindle et al., 2009), 70% (Amor et al., 2006), 76% (Levin & Yehudai, 2017) and 82% (Antoniol et al., 2008), and also better than our own previous result of 89% (Amit & Feitelson, 2019). 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 (Dawid & Skene, 1979) 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" (Box, 1979). 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.

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

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

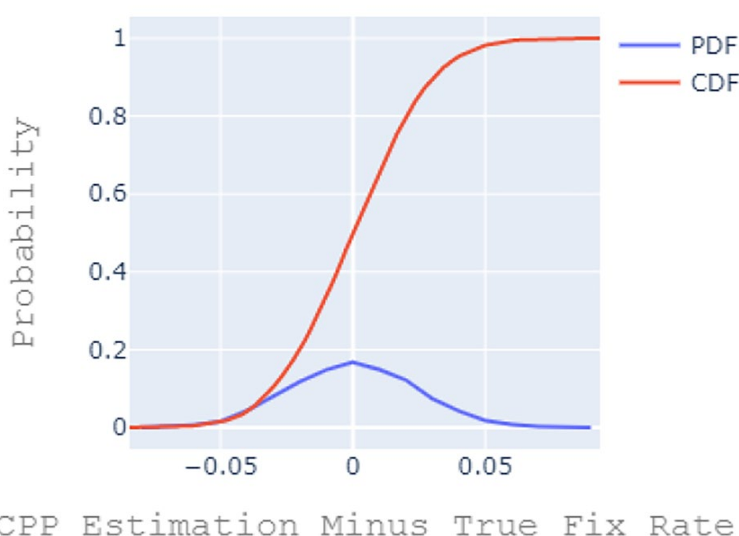
statistics tend to differ on different samples. In this section, we would like to show that the MLE method is robust to such changes.

In order to evaluate how sensitive the maximum likelihood estimation is to changes in the data, we used the bootstrap method (Efron, 1992). 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.

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

Fig. 1 Difference distribution in validation bootstrap



[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 (Sect. 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 Sect. 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 (Munaiah et al., 2017; Kalliamvakou et al., 2015). 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 (Gousios & Spinellis, 2012) 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 (Hackbarth et al., 2016). Moreover, assessing quality based on bugs is a process metric and therefore conditionally independent (Blum & Mitchell, 1998; Lewis, 1998) 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 (Walkinshaw & Minku, 2018; Kim et al., 2007; Rahman et al., 2011).

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 (Jones, 1991, 2012). 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” (Fowler et al., 1997; Van Emden & Moonen, 2002; Yamashita & Moonen, 2012; Khomh et al., 2009; Taba et al., 2013) and “technical debt” (Cunningham, 1992; Tom et al., 2013; Kruchten et al., 2012). 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 (Romano et al., 2020).

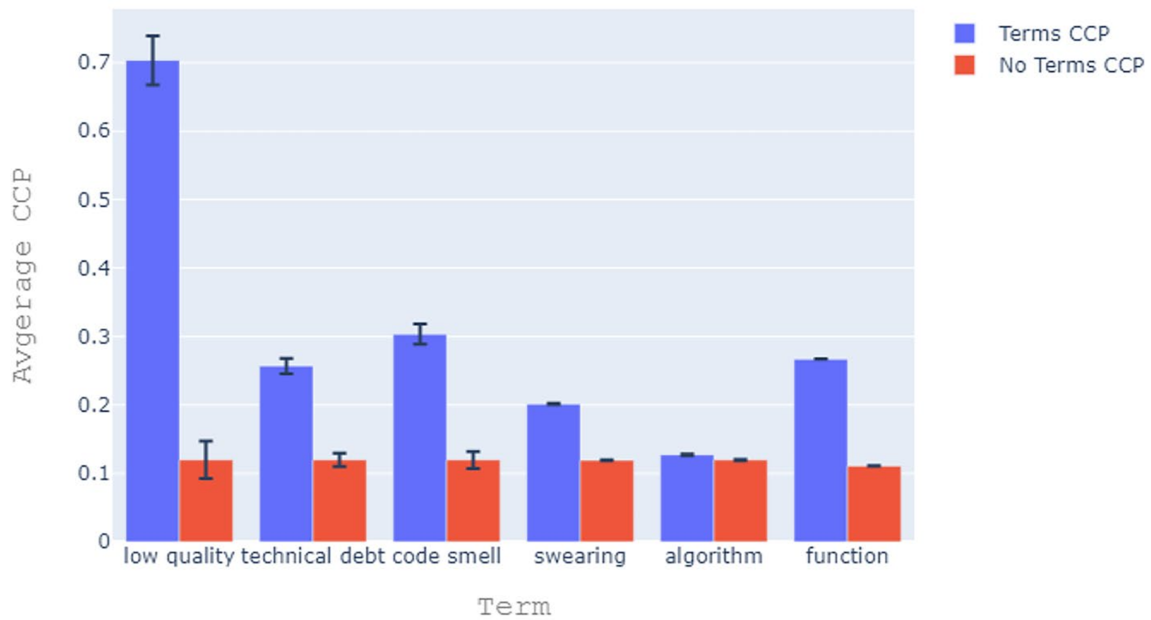


Fig. 2 CCP of files with or without different quality terms

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

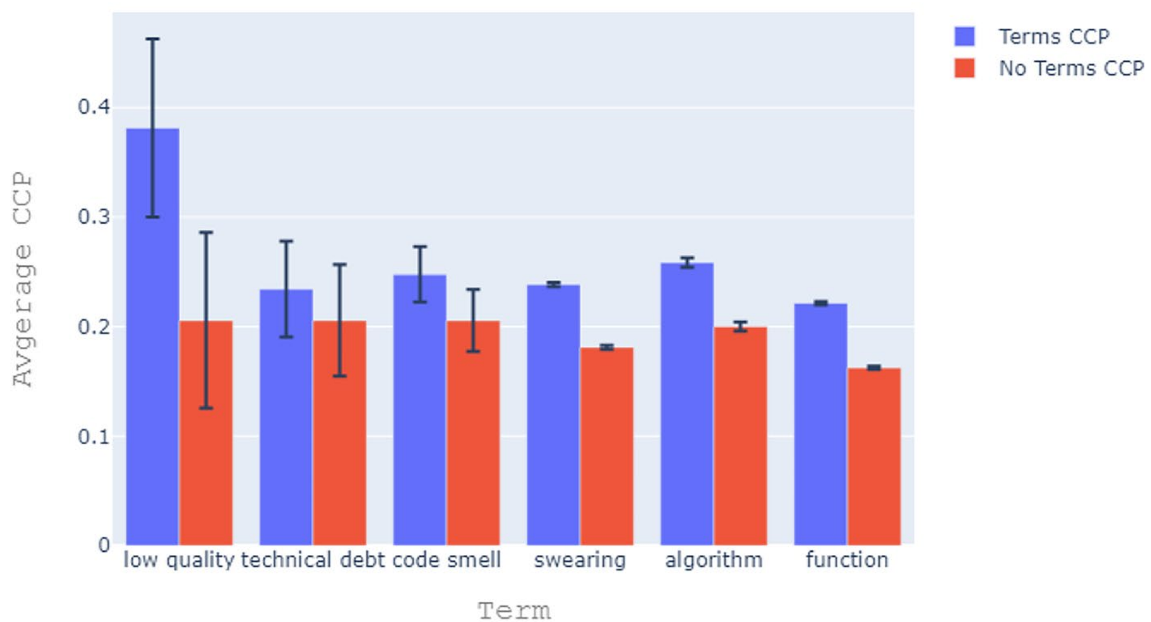


Fig. 3 CCP of projects with or without different quality terms

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” (Potdar & Shibab, 2014). We used the terms suggested by Rantala et al. (2020) 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 Sect. 4.4. We also control for the developer, by observing the behavior of the same developer in different projects (Sect. 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 (Hastings et al., 1947). 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).

However, this set is redundant in the sense that some projects are closely related (Kalliamvakou et al., 2015). 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 cloning is prevalent and might impact analysis (Gharehyazie et al., 2019; Lopes et al., 2017; Allamanis, 2019). Using commits to identify relationships (Mockus et al., 2020), 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 Sect. 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.

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

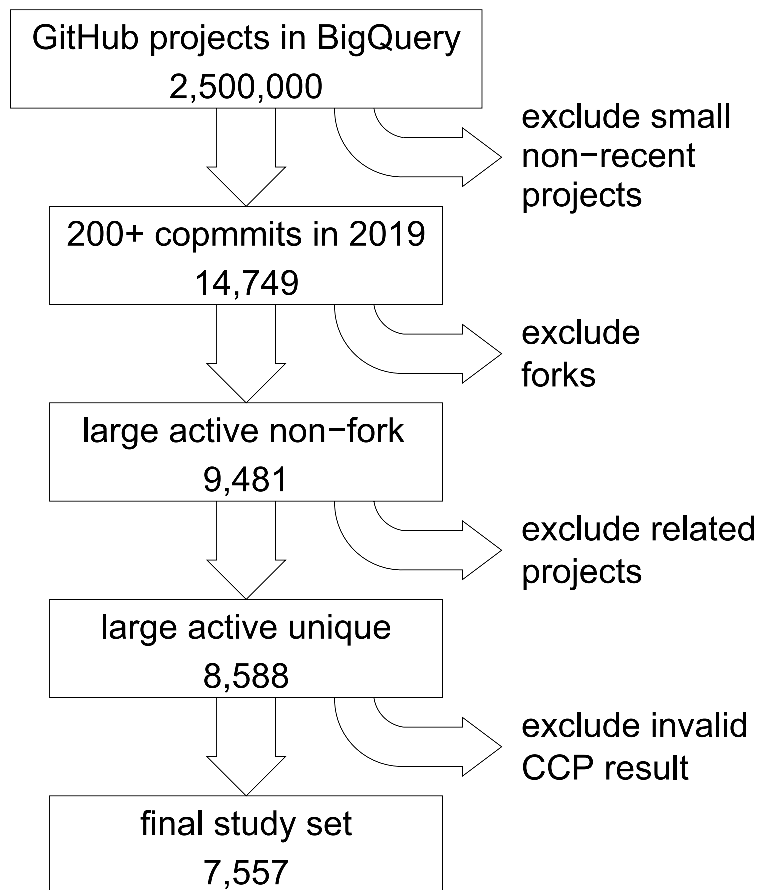


Fig. 4 Process for selecting projects for analysis

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:

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

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.

Table 3 CCP distribution in active GitHub projects

	Full data set (8,588 projects)		CCP ∈ [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

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 (Sect. 8.2), programming languages (Sect. 8.3), project age (Sect. 8.4), and detection efficiency (Sect. 6). Though influential, we don't control the project domain, for reasons explained in Sect. 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 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 (Lientz et al., 1978). 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 year 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 (Jones, 1991, 2012). 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" (Raymond, 1998), 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.

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

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

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" (Williams & Kessler, 2002). In order to analyze it, we used the metric suggested by Avelino et al. (2016). Truck Factor Developers Detachment (TFDD) is the event in which the core developers abandon a project as if a virtual truck had hit them (Avelino et al., 2019). We used instances of TFDD identified by Avelino et al. and matched them with the GitHub behavior (Avelino et al., 2019). 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.

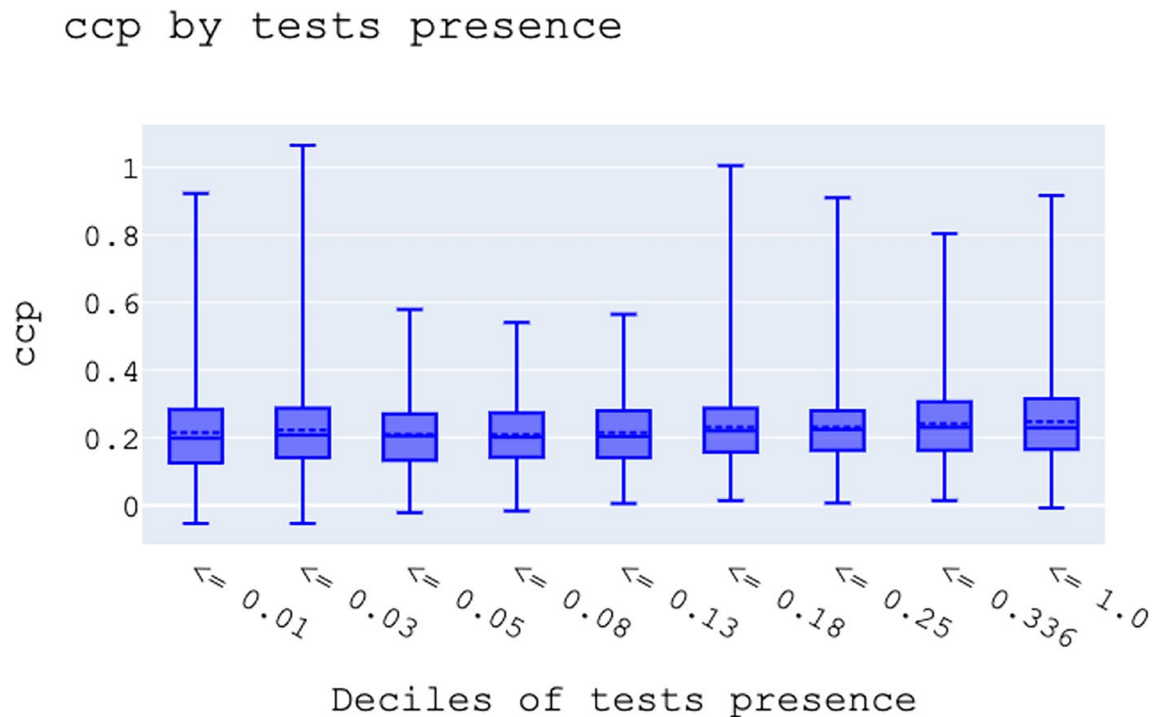


Fig. 5 CCP by Tests Presence. In this and following figures, each boxplot shows the 5, 25, 50 (solid line), 75, and 95 percentiles. The dashed line represents the mean

6.3 Use of tests

The use of tests (Myers et al., 2004; Beizer, 2003) 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 (Zaidman et al., 2011; Levin & Yehudai, 2017; Amit & Feitelson, 2019). 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 (Berger et al., 2019).

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.

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 Whithead reported that the median time to fix a bug is about 200 days (Kim & Whitehead, 2006). In order to compute bug fix time one should identify the bug-inducing commit, for example by using the SZZ algorithm (Śliwerski et al., 2005) 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 organization 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 behavior 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 (Sects. 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 (Lipow, 1982; Gil & Lalouche, 2017). 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

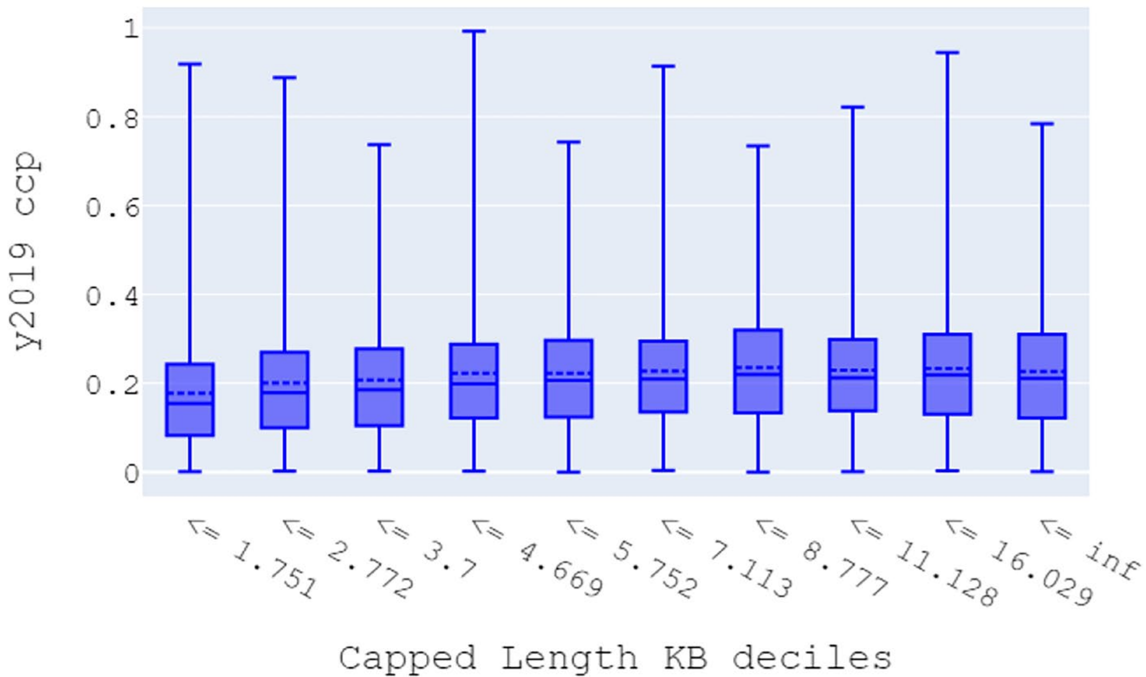


Fig. 6 CCP distribution for files with different lengths (in KB, capped)

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.

7.2 Smells

Code smells are properties of the source code that indicate low quality and potential problems (Arcelli Fontana et al., 2013; Van Emden & Moonen, 2002). 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 (Amit et al., 2021). 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.

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.

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

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 (Zimmermann et al., 2003; Amit & Feitelson, 2019). 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).

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%.

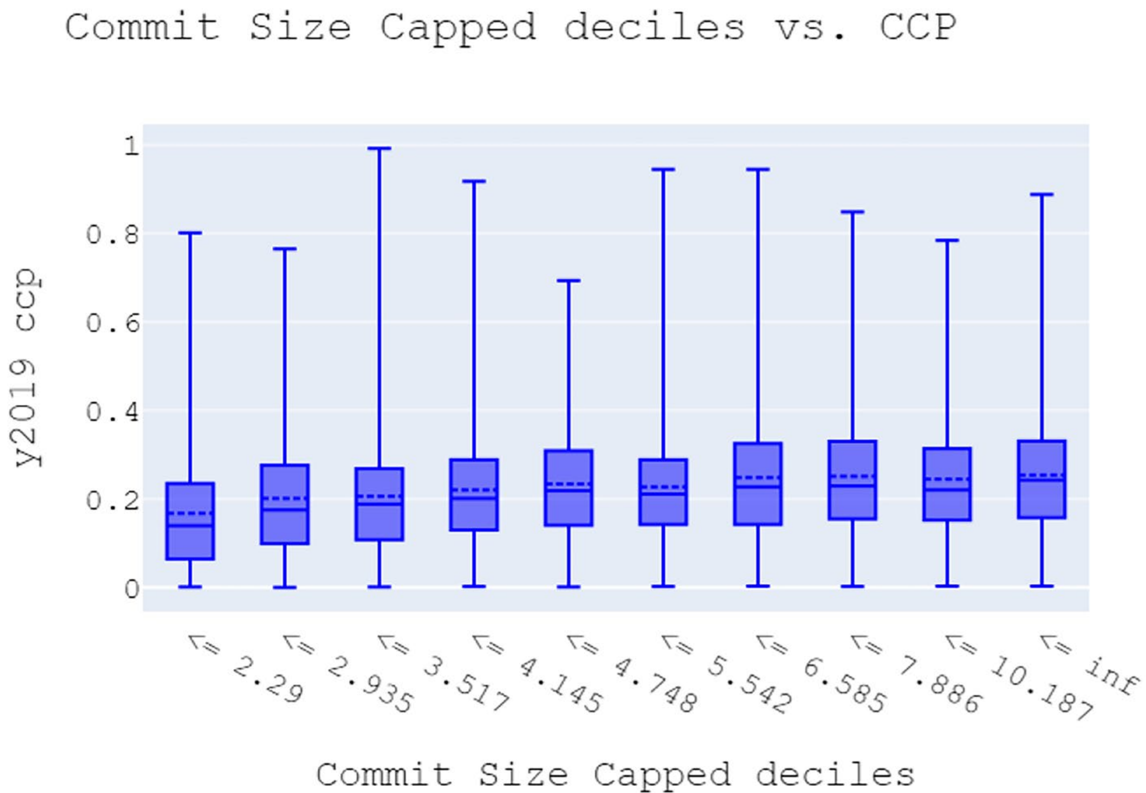


Fig. 7 CCP distribution for projects with different average commit sizes (number of files, capped, in non-corrective commits)

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.

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 the average number of

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

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

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 (Norick et al., 2010; Bird et al., 2011; Weyuker et al., 2008). The number of developers and CCP have

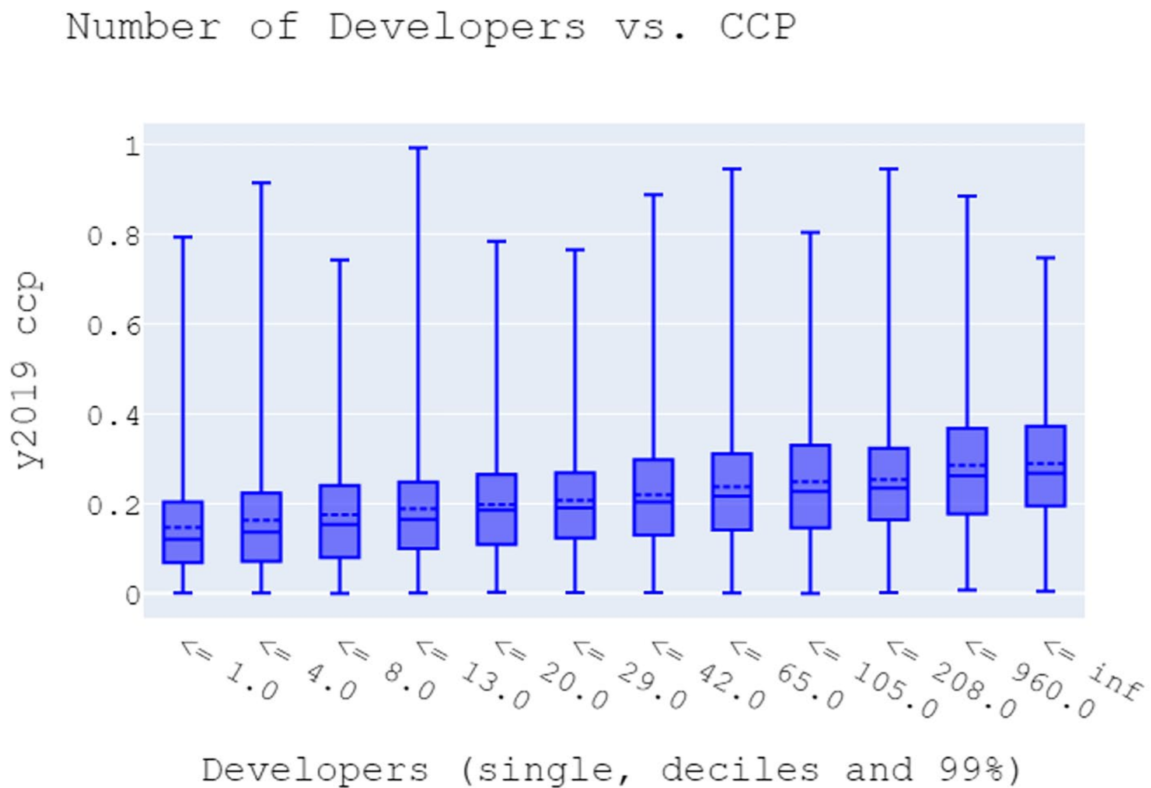


Fig. 8 CCP distribution for projects with different numbers of developers

Pearson correlation of 0.12. The number of developers can reach very high values and therefore be very influential.

Figure 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” (Brooks, 1975). 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 (Prechelt, 2000; Nanz & Furia, 2015; Ray et al., 2014; Berger et al., 2019; Bhattacharya & Neamtiu, 2011; Kochhar et al., 2016). 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.,

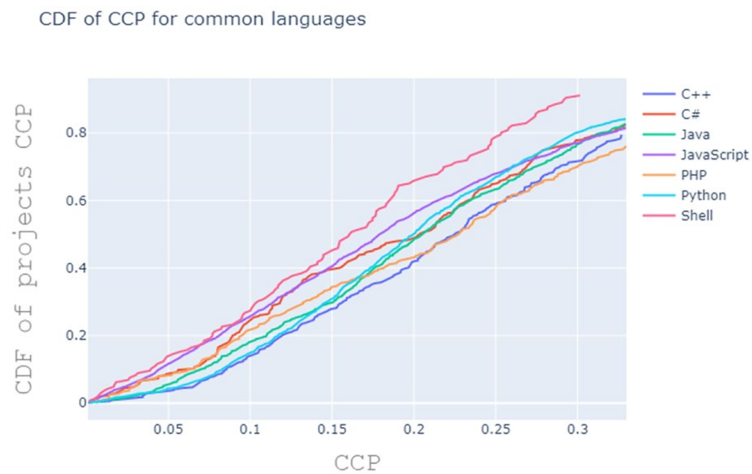


Fig. 9 Cumulative distribution of CCP by language. Distributions shifted to the right tend to have higher CCP

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.

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 (Fisher, 1919) led to an F-statistic of 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 Prechelt, 2000; Nanz & Furia, 2015; Ray et al., 2014; Berger et al., 2019) for comparisons

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

		Metric			
Language	Projects	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

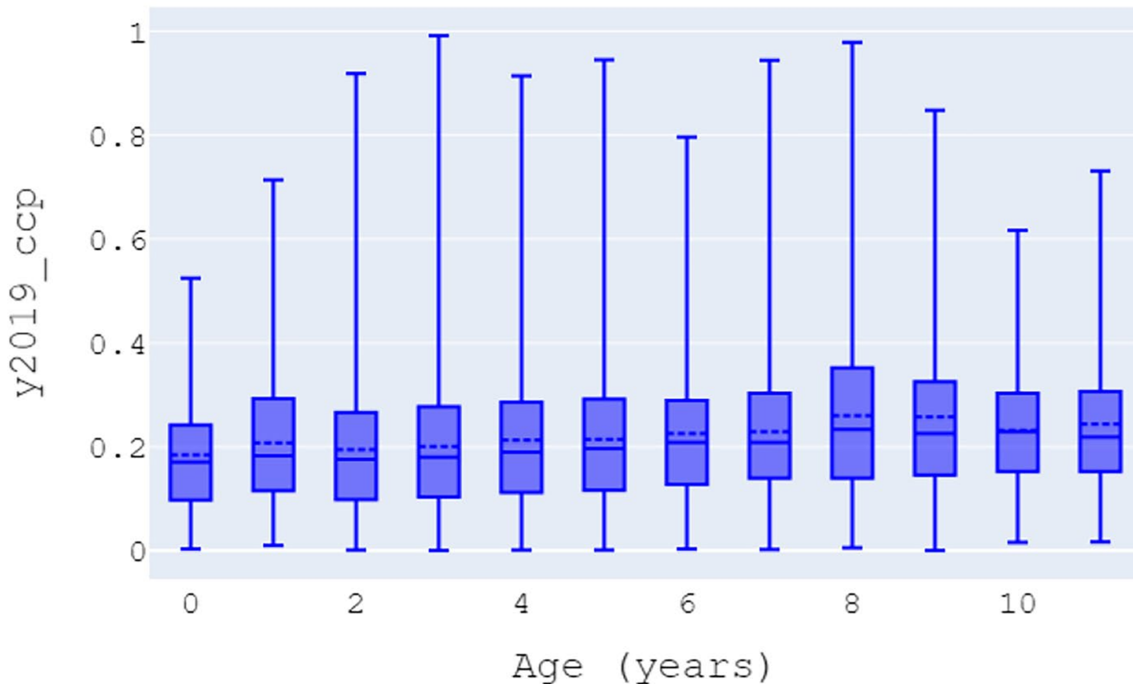


Fig. 10 CCP distribution (during 2019) in projects of different ages

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 Ray et al. (2014) and Berger et al. (2019), 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 (Lehman, 1980; Lehman et al., 1997). 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.

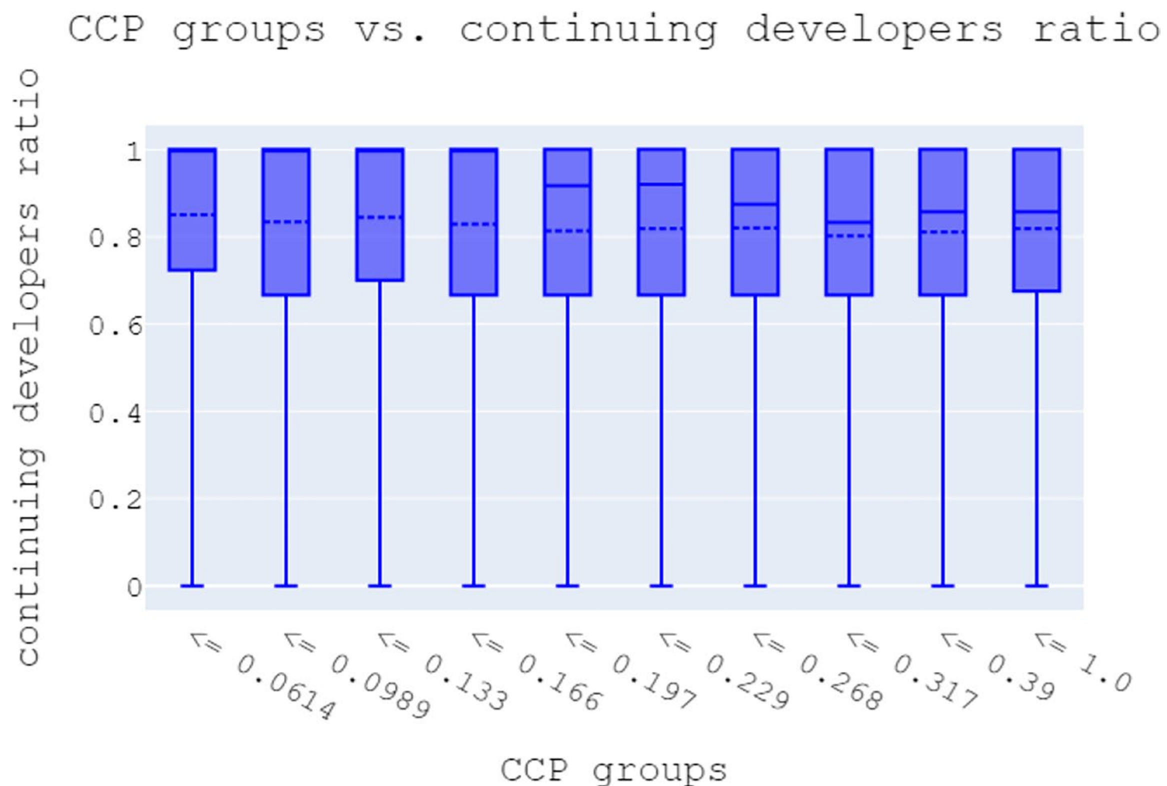


Fig. 11 Projects' developer retention per CCP decile. Note the change in the median

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 (Campbell et al., 1993; Wright & Cropanzano, 2000). Argyle investigated the relation between developers' happiness and their job satisfaction and work performance, showing "modestly positive correlations with productivity, absenteeism, and labour turnover" (Argyle, 1989). In the other direction, Ghayyur et al. conducted a survey in which 72% claimed that poor code quality is demotivating (Ghayyur et al., 2018). Hence, quality might be both the outcome and the cause of motivation.

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 (Fig. 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 ($1 - retention$), 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.

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.

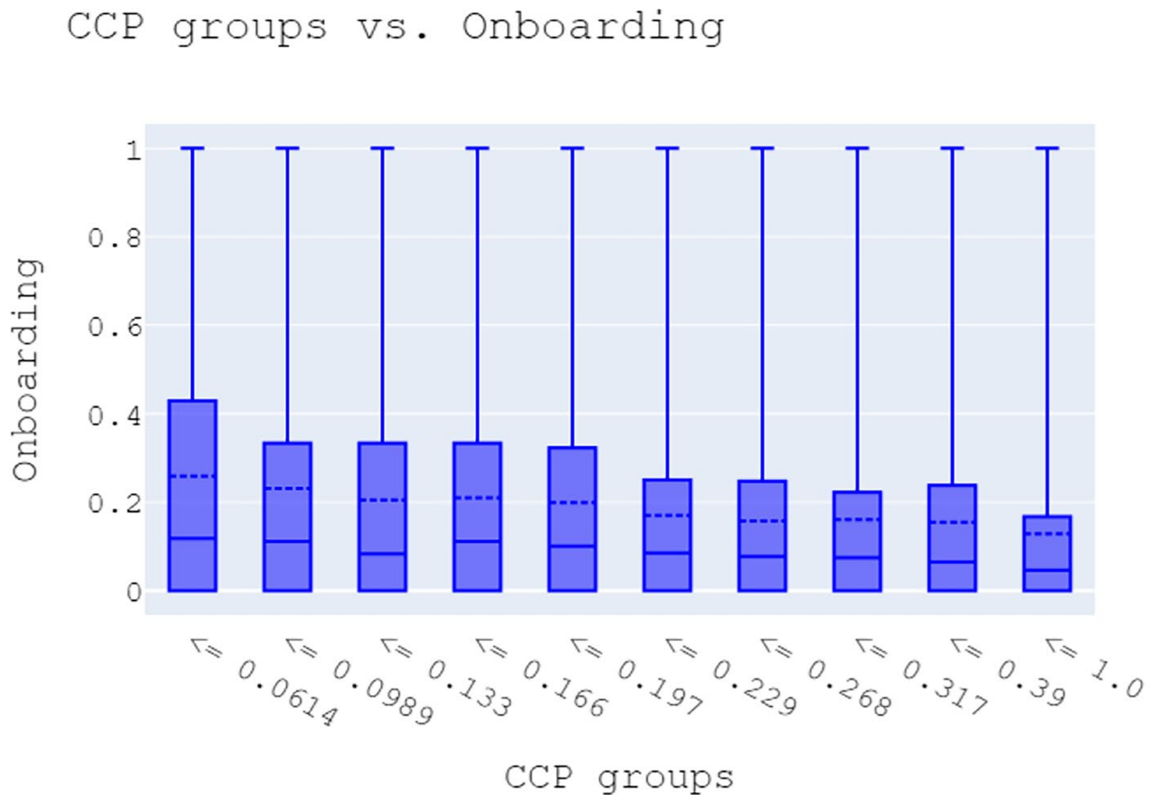


Fig. 12 On-boarding per CCP decile

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 (Maxwell et al., 1996), modules (Morasca & Russo, 2001), and function points (Maxwell & Forselius, 2000; Jiang et al., 2007) per time unit have been suggested and criticized (Kemerer & Porter, 1992; Kemerer, 1993). 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. (1968). The number of commits is correlated with self-rated productivity (Muphy-Hill et al., 2019) and team lead perception of productivity (Oliveira et al., 2020). 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.

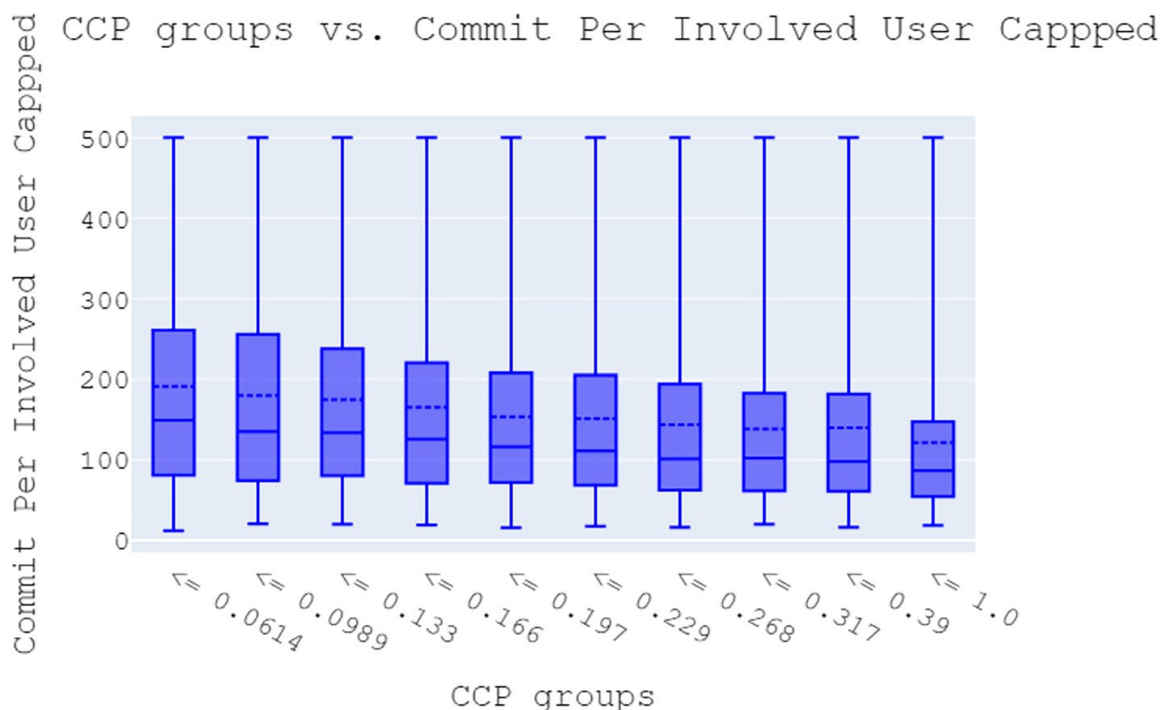


Fig. 13 Distribution of commits per year of involved developers (capped) per CCP decile

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 Fig. 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).

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

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

-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 (Gousios & Spinellis, 2012). Note that while our project selection represents large projects active during 2019, the selection criteria of Gousios and Spinellis (2012) 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.

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 (Oisen, 1971) 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 (Crosby, 1979). 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

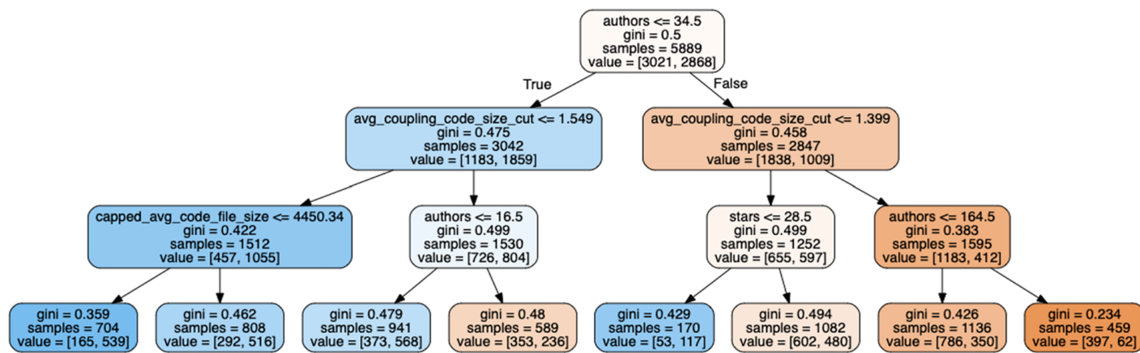


Fig. 14 Model Predicting High Quality Group

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 (Jones 2015, 2006) and developers performing the same tasks during "Personal Software Process" training (Shrikanth et al., 2020).

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 (Van Hulse et al., 2007; Krawczyk, 2016; Oak et al., 2019).

We present in Fig. 14 a small tree model generated using scikit-learn (Pedregosa et al., 2011) 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 (Quinlan, 1986) 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, Sect. 8.2), coupling (quality, Sect. 7.3), file length (quality, Sect. 7.1)

and stars (detection efficiency, Sect. 6) are informative. Yet, even when using them, the leaves are rather diverse as demonstrated by the Gini metric values (Dorfman, 1979) 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 (Arpit et al., 2017). 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 Sect. 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 (Blum & Mitchell, 1998; Lewis, 1998) 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 submodels 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 (Śliwerski et al., 2005)). 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" (Raymond, 1998) 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 Sect. 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 (Kim & Whitehead, 2006). Different policies and methods, such as the adoption of continuous integration (Bernardo et al., 2018), 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 (Muphy-Hill et al., 2019) and team lead perception of productivity (Oliveira et al., 2020), 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 (Herzig & Zeller, 2013; Herbold et al., 2020), 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 Sect. 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. (2018), 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 (Hindle et al., 2009; Amor et al., 2006; Levin & Yehudai, 2017; Amit & Feitelson, 2019), 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.

12 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 (Levin & Yehudai, 2017). We thank Guilherme Avelino for drawing our attention to the importance of Truck Factor Developers Detachment (TFDD) and providing a data set (Avelino et al., 2019). Many thanks to the reviewers whose comments were instrumental in improving the focus of the paper.

References

Al-Kilidar, H., Cox, K., & Kitchenham, B. (2005). The use and usefulness of the ISO/IEC 9126 quality standard. In *International Symposium Empirical Software Engineering*, pages 126–132.

Allamanis, M. (2019). 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. Association for Computing Machinery.

Amit, I., Ben Ezra, N., & Feitelson, D. G. (2021). Follow your nose – which code smells are worth chasing? arXiv:2103.01861 [cs.SE].

Amit., I, Feitelson, D. G. (2019). 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.

Amit, I., Firstenberg, E., & Meshi., Y. (2017). Framework for semi-supervised learning when no labeled data is given. U.S. patent application #US20190164086A1.

Amor, J. J., Robles, G., Gonzalez-Barahona, J. M., & Navarro, A. (2006). Discriminating development activities in versioning systems: A case study.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., & Guéhéneuc, Y. G. (2008). 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. Association for Computing Machinery.

Arcelli Fontana, F., Ferme, V., Marino, A., Walter, B., & Martenka, P. (2013). 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.

Argyle, M. (1989). Do happy workers work harder? the effect of job satisfaction on job performance. In R. Veenhoven (Ed.), *How harmful is happiness? Consequences of enjoying life or not*. Rotterdam, The Netherlands: Universitaire Pers.

Arpit, D., Jastrzębski, S., Ballas, N., Krueger, D., Bengio, E., Kanwal, M. S, Maharaj, T, Fischer, A., Courville, A., Bengio, Y., et al. (2017). A closer look at memorization in deep networks. arXiv preprint arXiv:1706.05394.

Avelino, G., Constantinou, E., Valente, M. T., & Serebrenik, A. (2019) On the abandonment and survival of open source projects: An empirical investigation. *CoRR*, abs/1906.08058. abs/1906.08058

Avelino, G., Passos, L. T., Hora, A. C., & Valente, M. T. (2016). A novel approach for estimating truck factors. *CoRR*, abs/1604.06766.

Baggen, R., Correia, J. P., Schill, K., & Visser, J. (2012). Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal, 20*(2), 287–307.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761.

Beizer, B. (2003). *Software testing techniques*. Dreamtech Press.

Berger, E. D, Hollenbeck, C., Maj, P., Vitek, O., & Vitek, J. (2019). On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems*, 41(4).

Bernardo, J. H, da Costa, D. A, & Kulesza, U. (2018). 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. Association for Computing Machinery.

Bhattacharya, P. & Neamtiu, I. (2011). 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.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009). 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. ACM.

Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011). 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.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., & Devanbu, P. (2009). The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10.

Blum, A., & Mitchell, T. (1998). 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. ACM.

Boehm, B., & Basili, V. R. (2001). Software defect reduction top 10 list. *Computer, 34*(1), 135–137.

Boehm, B. W. (1981). Software Engineering Economics. Prentice-Hall.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *International Conference Software Engineering, 2,* 592–605.

Boehm, B. W., & Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering, 14*(10), 1462–1477.

Box, G. (1979). 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.

Brooks Jr., F. P. (1975). The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley.

Campbell, J. P., McCloy, R. A., Oppler, S. H., & Sager, C. E. (1993). A theory of performance. In N. Schmitt, W. C. Borman, and Associates, editors, *Personnel Selection in Organizations*, pages 35–70. Jossey-Bass Pub.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement, 20*(1), 37–46.

Corral, L., & Fronza, I. (2015). 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.

Crosby, P. (1979). *Quality Is Free: The Art of Making Quality Certain*. McGrawHill.

Cunningham, W. (1992). 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. Association for Computing Machinery.

da Costa, D. A., McIntosh, S., Treude, C., Kulesza, U., & Hassan, A. E. (2018). The impact of rapid release cycles on the integration delay of fixed issues. *Empirical Software Engineering, 23*(2), 835–904.

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41.

Dawid, A. P. & A. M. Skene. (1979). 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.

Dawson, M., Burrell, D., Rahim, E., & Brewster, S. (2010). Integrating software assurance into the software development life cycle (SDLC). *Journal of Information Systems Technology and Planning, 3,* 49–53.

Dorfman, R. (1979). A formula for the gini coefficient. *The review of economics and statistics, 61*(1), 146–149.

Dromey, G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering, 21*(2), 146–162.

Efron, B. (1992). Bootstrap Methods: Another Look at the Jackknife. In S. Kotz & N. L. Johnson (Eds.), *Breakthroughs in Statistics: Methodology and Distribution*, pages 569–593. Springer New York, New York, NY.

Fisher, R. (1919). The correlation between relatives on the supposition of mendelian inheritance. *Transactions of the Royal Society of Edinburgh, 52*(2), 399–433.

Fowler, M., Beck, K., & Opdyke, W. R. (1997). Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*.

Gharehyazie, M., Ray, B., Keshani, M., Zavosht, M. S., Heydarnoori, A., & Filkov, V. (2019). Cross-project code clones in github. *Empirical Software Engineering, 24*(3), 1538–1573.

Ghayyur, S. A. K., Ahmed, S., Ullah, S., & Ahmed, W. (2018). The impact of motivator and demotivator factors on agile software development. *Intl J Adv Comp Sci Appl, 9*(7).

Gil, Y., & Lalouche, G. (2017). On the correlation between size and metric validity. *Empirical Software Engineering, 22*(5), 2585–2611.

Gousios, G., & Spinellis, D. (2012). Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering, 26*(7), 653–661.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897–910.

Hackbarth., R., Mockus, A., Palframan, J., & Sethi, R. (2016). Improving software quality as customers perceive it. *IEEE Software, 33*(4):40–45.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

Halstead, M. H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc.

Hastings, C., Mosteller, F., Tukey, J. W., & Winsor, C. P. (1947). Low moments for small samples: A comparative study of order statistics. *Annals of Mathematical Statistics, 18*(3):413–426.

Hattori, L. P., & Lanza, M. (2008). On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 63–71. IEEE.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of Chemical Information and Computer Sciences, 44*(1), 1–12.

Herbold, S., Trautsch, A., Ledel, B., Aghamohammadi, A., Ghaleb, T. A., Chahal, K. K., Bossenmaier, T., Nagaria, B., Makedonski, P., Ahmadabadi, M. N., Szabados, K., Spieker, H., Madeja, M., Hoy, N., Lenarduzzi, V., Wang, S., Rodriguez-Perez, G., Colomo-Palacios, R., Verdecchia, R., Singh, P., Qin, Y., Chakroborti, D., Davis, W., Walunj, V., Wu, H., Marcilio, D., Alam, O., Aldaeej, A., Amit, I., Turhan, B., Eismann, S., Wickert, A. K., Malavolta, I., Sulir, M., Fard, F., Henley, A. Z., Kourtzanidis, S., Tuzun, E., Treude, C., Shamasbi, S. M., Pashchenko, I., Wyrich, M., Davis, J., Serebrenik, A., Albrecht, E., Aktas, E. U., Strber, D., & Erbel, J. (2020). *Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling*. arXiv:2011.06244 [cs.SE].

Herzig, K., Just, S., & Zeller, A. (2013). 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. IEEE Press.

Herzig, K., Zeller, A. (2013). The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130.

Hindle, A., German, D. M., Godfrey, M. W., & Holt, R. C. (2009). Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39.

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Not, 39*(12), 92–106.

International Organization for Standardization. (2001). ISO/IEC 9126-1:2001 Software engineering - Product quality - Part 1: Quality model.

International Organization for Standardization. (2011). ISO/IEC 25010:2011 Systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - System software quality models.

Jiang, Z., Naud, P., & Comstock, C. (2007). An investigation on the variation of software development productivity. International Journal of Computer and Information Science and Engineering, *1*(2), 72–81.

Jones, C. (1991). *Applied Software Measurement: Assuring Productivity and Quality*. New York, NY, USA: McGraw-Hill Inc.

Jones, C. (2006). Social and technical reasons for software project failures. *Crosstalk, the Journal of Defense Software Engineering, 19*(6):4.

Jones, C. (2012). Software quality in 2012: A survey of the state of the art. [Online; accessed 24-September-2018].

Jones, C. (2015). Wastage: The impact of poor quality on software economics. *Software Quality Professional, 18*(1):23-32. retrieved from http://asq.org/pub/sqp/.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2016). An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering, 21*, 2035-2071.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering, 39*(6), 757–773.

Reliability of function points measurement. (1993). A field experiment. *Communication ACM, 36*(2), 85–97.

Kemerer, C. F., & Porter, B. S. (1992). Improving the reliability of function point measurement: An empirical study. *IEEE Transactions on Software Engineering, 18*(11), 1011–1024.

Khomh, F., Dhaliwal, T., Zou, Y., & Adams, B. (2012). 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. IEEE Press.

Khomh, F., Di Penta, M., & Gueheneuc, Y. G. (2009). 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.

Kim, S., & Whitehead Jr., E. J. (2006) 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. ACM.

Kim, S., Zimmermann, T., Whitehead Jr, E. J., & Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA. IEEE Computer Society.

Kochhar, P. S., Wijedasa, D., & Lo, D. (2016). 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.

Krawczyk, B. (2016). Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence, 5*(4), 221–232.

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software, 29*(6), 18–21.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–50.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE, 68*(9), 1060–1076.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. *International Software Metrics Symposium, 4,* 20–32.

Levin, S., & Yehudai, A. (2017). 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.

Levin, S., & Yehudai, A. (2017). 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.

Lewis, D. D. (1998). 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. Springer Berlin Heidelberg.

Lientz, B. P. (1983). Issues in software maintenance. *ACM Computing Surveys, 15*(3), 271–278.

Lientz, B. P., Swanson, E. B., & Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communication ACM, 21*(6), 466–471.

Lipow, M. (1982). Number of faults per line of code. *IEEE Transactions on Software Engineering, 4,* 437–439.

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., & Vitek, J. (2017). Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA).

Maxwell, K. D., & Forselius, P. (2000). Benchmarking software development productivity. *IEEE Software, 17*(1), 80–88.

Maxwell, K. D., Van Wassenhove, L., & Dutta, S. (1996). Software development productivity of european space, military, and industrial applications. *IEEE Transactions on Software Engineering, 22*(10), 706–718.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.

Mockus, A., Spinellis, D., Kotti, Z., & Dusing, G. J. (2020). A complete set of related git repositories identified via community detection approaches based on shared commits. In 17th *Mining Software Repositories*, pages 513-517.

Molnar, A. J., NeamŢu, A., & Motogna, S. (2020). Evaluation of software product quality metrics. pp. In E. Damiani, G. Spanoudakis, & L. A. Maciaszek (Eds.), *Evaluation of Novel Approaches to Software Engineering* (pp. 163–187). Cham: Springer International Publishing.

Morasca, S., & Russo, G. (2011). An empirical study of software productivity. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 317–322.

Moser, R., Pedrycz, W., & Succi, G. (2008). 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. ACM.

Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017). Curating github for engineered software projects. *Empirical Software Engineering, 22,* 04.

Murphy-Hill, W., Jaspan, C., Sadowski, C., Shepherd, D. C., Phillips, M., Winter, C., Dolan, A. K., Smith, E. K., & Jorde, M. A. (2021). What predicts software developers productivity? *Transaction on Software Engineering 47*, 582-594

Myers, G. J., Badgett, T., Thomas, T. M., & Sandler, C. (2004). *The art of software testing*, volume 2. Wiley Online Library.

Nanz, S., & Furia, C. A. (2015). 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.

Norick, B., Krohnm J., Howard, E., Welna, B., & Izurieta, C. (2010). 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–.

Oak, R., Du, M., Yan, D., Takawale, H., & Amit, I. (2019). 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. Association for Computing Machinery.

Oisen, R. (1971). Can project management be defined? *Project Management Quarterly, 2*(1), 12–14.

Oliveira, E., Fernandes, E., Steinmacher, I., Cristo, M., Conte, T., & Garcia, A. (2020). Code and commit metrics of developer productivity: a study on team leaders perceptions. *Empirical Software Engineering 25*, 2519-2549.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12,* 2825–2830.

Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100. IEEE.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer, 33*(10), 23–29.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*(1), 81–106.

Rahman, F., & Devanbu, P. (2013). How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441.

Rahman, F., Posnett, D., Hindle, A., Barr, E. T., & Devanbu, P. T. (2011). Bugcache for inspections: hit or miss? In *SIGSOFT FSE*.

Rantala, L., Mantyla, M., & Lo, D. (2020). Prevalence, contents and automatic detection of KL-SATD. 46th *Euromicro Conference on Software Engineering and Advanced Applications,* 385-388.

Ratner, A. J., De Sa, C. M., Wu, S., Selsam, D., & Ré, C. (2016). 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.

Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). 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, US. ACM.

Raymond, E. (1998). The cathedral and the bazaar. First Monday, 3(3).

Reddivari, S. & Raman, J. (2019). 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.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society, 74*(2), 358–366.

Romano, S., Caulo, M., Scanniello, G., Baldassarre, M. T., & Caivano, D. (2020). Sentiment polarity and bug introduction. In *International Conference on Product-Focused Software Process Improvement*, pages 347–363. Springer.

Rosenberg, J. (1997). Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*, pages 137–142. IEEE.
Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM, 11*(1), 3–11.
Schach, S. R., Jin, B., Yu, L., Heller, G. Z., & Offutt, J. (2003). Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Software Engineering, 8*(4), 351–365.
Schneidewind, N. F. (2002). Body of knowledge for software quality measurement. *Computer, 35*(2), 77–83.
Settles, B. (2010). Active learning literature survey. *Technical report*, University of Wisconsin Madison.
Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal, 3*(2), 30–36.
Shihab, E., Hassan, A. E., Adams, B., & Jiang, Z. M. (2012). 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. ACM.
Shrikanth, N. C., & Menzies, T. (2020). Assessing practitioner beliefs about software defect prediction. In *International Confernce on Softwware Engineering,* number 42.
Shrikanth, N. C., Nichols, W., Fahid, F. M., & Menzies, T. (2020). Assessing practitioner beliefs about software engineering. arXiv:2006.05060.
Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? *SIGSOFT Software Engineering Notes, 30*(4), 1–5.
Spinellis, D. (2006). *Software Quality: The Open Source Perspective*. Pearson Education Inc.
Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open source software development. *Information Systems Journal, 12*(1), 43–60.
Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA. IEEE Computer Society Press.
Taba, S. E. S., Khomh, F., Zou, Y., Hassan, A. E., & Nagappan, M. (2013). Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*, pages 270–279.
Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software, 86*(6), 1498–1516.
Van Emden, E., Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE.
Van Hulse, J., Khoshgoftaar, T. M., & Napolitano, A. (2007). Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942.
Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., & Filkov, V. (2015). 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. ACM.
Walkinshaw, N., & Minku, L. (2018). 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. ACM.
Weyuker, E., Ostrand, T., & Bell, R. (2008). Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering, 13*,539–559.
Williams, L., & Kessler, R. (2002). *Pair Programming Illuminated*. Inc, USA: Addison-Wesley Longman Publishing Co.
Wood, A. (1996). Predicting software reliability. *Computer, 29*(11), 69–77.
Wright, T. A., & Cropanzano, R. (2000). Psychological well-being and job satisfaction as predictors of job performance. *Journal of Occupational Health Psychology, 5,* 84–94.
Yamada, S. & Osaki, S. (1985). Software reliability growth modeling: Models and applications. *IEEE Transactions on Software Engineering*, SE-11(12):1431–1437.
Yamashita, A. & Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE.
Zaidman, A., Van Rompaey, B., van Deursen, A., & Demeyer, S. (2011). 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.
Zimmermann, T., Diehl, S., & Zeller, A. (2003). How history justifies system architecture (or not). In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 73–83.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Idan Amit is a PhD student at the Hebrew University. He also serves as the research manager of Acumen. Mr. Amit works in research and software development for many years.

Dror Feitelson has been on the faculty of the department of computer science at the Hebrew University since 1995. His research interests are in experimental computer science and empirical software engineering.

4.3 A Large Scale Survey of Motivation in Software Development and Analysis of its Validity

Unpublished: Amit, Idan, and Dror G. Feitelson. "A Large Scale Survey of Motivation in Software Development and Analysis of its Validity", 2023.

In this thesis, we developed a machine learning method to investigate motivation on a large scale, over years, in a quantitative and reproducible way. We based our work on labeling functions, classifiers that need to be just somewhat better than a guess [67, 62, 6]. Given such labeling functions, we could apply our relation identification methods to investigate the relation between motivation and productivity, etc.

However, part of the labeling functions validation should be done by comparing reported motivation and actual behavior. We therefore conducted a survey and asked the participants for their GitHub profile, allowing us to do the matching.

The survey led to interesting results besides the matching. We asked about eleven motivators from the literature (e.g., ownership, recognition). All eleven motivators had good performance when predicting high motivation. We also conducted a follow-up survey with the same developers with the same question on the same project. This allowed us to evaluate the performance of *motivator improvement* in predicting *motivation improvement*. Recognition, and specifically expressing interest were found to be an effective way to improve motivation, a way that is simple, widely applicable, and always within budget.

We also extended the co-change model, building a full model on all features, considering the context and aiming to predict a change. Finally, we also examined the reliability and validity of the answers and noted that they are limited. The methods that we used for the validity are a contribution of their own.

A Large Scale Survey of Motivation in Software Development and Analysis of its Validity

Idan Amit
idan.amit@mail.huji.ac.il
Dror G. Feitelson
feit@cs.huji.ac.il
Department of Computer Science
The Hebrew University, Jerusalem 91904, Israel

Abstract

Context: Motivation is known to improve performance. In software development in particular, there has been considerable interest in the motivation of contributors to open-source.
Objective: We would like to predict motivation, in various settings. We identify 11 motivators from the literature (enjoying programming, ownership of code, learning, self-use, etc.), and evaluate their relative effect on motivation using supervised learning. Since motivation is an internal subjective feeling, we also analyze the validity of the answers.
Method: We conducted a survey with 66 questions on motivation which was completed by 521 developers. Most of the questions used an 11-point scale. We evaluated the answers' validity by comparing related questions, comparing to actual behavior on GitHub, and comparison with the same developer in a follow-up survey.
Results: Validity problems include moderate correlations between answers to related questions, as well as self-promotion and mistakes in the answers. Despite these problems, predictive analysis—investigating how diverse motivators influence the probability of high motivation—provided valuable insights. The correlations between the different motivators are low, implying their independence. High values in all 11 motivators predict increased probability of high motivation. In addition, improvement analysis shows that an increase in most motivators predicts an increase in general motivation.
Conclusions: All 11 motivators indeed support motivation, but only moderately. No single motivator suffices to predict high motivation or motivation improvement, and each motivator sheds light on a different aspect of motivation. Therefore models based on multiple motivators predict *motivation improvement* with up to 94% accuracy, better than any single motivator.

Keywords: Motivation, Software engineering, Open-source development, Survey validity.

1 Introduction

Motivation has a high impact on human performance in many fields [38, 13, 60, 104, 54]. In the context of software development it is especially interesting [22, 89], due to the phenomenon of open-source development [81], where many of the developers are volunteers [88].

We conducted a large-scale survey asking developers about their motivation. Our survey contained questions related to eleven motivators which may affect motivation, taken from prior work. Some of these motivators relate to the culture of open source, such as adhering to its **ideology** [24] and developing software for **self-use** [81]. Other motivators are related to internal reasons, such as **enjoyment** from developing code [44, 12], **learning** from it [18], the feeling of overcoming a **challenge** [71, 67], or gaining **ownership** of a project [60, 16]. Additional motivators extend this to a wider context: being part of a **community** [15, 105], receiving **recognition** for one's work [41, 100] — or, in unpleasant situations, suffering from **hostility** [79, 3]. Finally, there are project-based motivators like a sense of **importance** [45], as well as receiving **payment** for participating in a project [25, 82]. The survey contained 66 questions, covering 11 motivators. We obtained answers from 1,724 developers, and 521 of them completed the whole survey. A year later we conducted a followup survey, answered by 124 of the original participants.

We wanted to investigate motivation in the framework of supervised learning, enabling us to predict motivation in many settings. Our main goal was to obtain answers to motivation questions by developers whose real activity is public on GitHub. This matching allows to validate labeling functions, heuristics that predict motivation better than a guess. A first example of the use of our data is given in Amit and Feitelson [10] which validated labeling functions like long commit messages and working diverse hours. The labeling functions that were validated on our data predicted well the retention of 151,775 developers in their real natural activity at GitHub.

Then we moved to predict motivation using the motivators. In prior work it was common to focus on hit rate, how common is a motivator [37, 30, 48, 41]. We wanted to extend this analysis and investigate questions like necessary and sufficient conditions for motivation.

The follow-up survey allowed us to further advance from mere motivation prediction to change prediction. If a motivator increased from the original survey to the follow-up, it improved. We modeled and investigated the predictive power of *motivator improvement* on *motivation improvement.*

Taken together, a motivator that is influential in all these separate ways is likely to have a true impact. In general, all the motivators were found to contribute to motivation in the predictive sense (knowing of a high motivator means higher probability of motivation). Other than challenge, ideology, and hostility, an improvement in the follow-up answers of the motivator also increases the probability of motivation improvement. On the other hand, none of the motivators is sufficient or necessary on its own for high motivation. Yet, when used together in a predictive model, one can predict well both high motivation and

motivation improvement.

Other than using the answers, we evaluated their reliability and validity. Motivation is an internal subjective feeling. When answering regarding our motivation, a valid answer depends on internal identification of the motivation, correct translation into a proper answer, and lack of personal biases [19, 58]. We examined the correlations between questions concerning the same motivator to evaluate coherence. We also checked the consistency of answers of the same person to the same questions in the original and follow-up surveys. Moreover, we investigated the validity of the answers, using identification of errors, differences between perception and actual behavior on GitHub, and biases.

We found that the attitudes toward motivators are only moderately stable. The validity also suffers from many kinds of problems. Simple mistakes, like typos, are rather rare. Biases are more common. For example, almost all developers agree that 'My code is of high quality'. This may indicate that our participants are indeed highly skilled developers. However, the GitHub profiles provided by some participants allowed us to compare their answers to their actual behavior. This showed that the participants overestimated their code quality, their documentation level, and their productivity. Despite these problems, questions belonging to the same motivator are usually coherent. Comparing answers of the same developer in the original and follow-up survey also shows a moderate stability. Therefore, the results that were supported by multiple analyses seem to be valid.

This study makes three main contributions:

- We conducted a large-scale survey on software developers' motivation, covering 11 motivators.
- We make several methodological innovations, including about assessing the validity of the results:
  - We asked participants for their GitHub profiles, which enabled comparing survey answers and actual behavior.
  - This allowed the validation of motivation labeling functions for motivation [10]. The labeling functions introduces a new methodology for motivation research complementing experiments, interviews, surveys, and case studies. This methodology facilitates quantified, reproducible, long-term investigation, based on large-scale data from real projects.
  - We conducted a follow-up survey, asking the same people the same questions again after more than a year. This allowed us to measure the answers' stability and the impact of changes in motivators on changes in motivation.
  - We framed the analysis as a supervised learning problem. We initially considered each single motivator as a classifier for high motivation, moved to full models, and then applied the same methods for motivation improvement.

  - We used the correlations of different questions in the same topic to measure absolute and relative coherence.
  - The large scale of the study allowed us to compare answers of different people in the same project, estimating subjectivity.
  - We used several types of analyses in tandem to investigate the relations between motivators and motivation. Most relations were consistent in most or all the methods, testifying to their validity.

- We analyze the results using a machine-learning framework and reach several conclusions regarding motivators and their influence:
  - We corroborate previous work showing that in general motivators from prior work are indeed correlated with motivation.
  - At the same time we find that none of them alone is enough to guarantee motivation, so developers usually need several reasons in order to have high motivation. Also, predictive performance is improved by taking multiple motivators into account.
  - Answers regarding motivation have moderate validity, shown by comparing to similar questions, comparing to a different date, or comparing to actual behavior.
  - We found that although hostility is rare, when it exists it has a negative influence on motivation. Yet, it tends to be unobserved by others in the same project.

2 Related Work

2.1 Motivation

Motivation, in general and in work context, has been extensively investigated due to its importance. Many theories were suggested. Skinner suggested operant conditioning, learning behavior due to reward and punishments [92]. Maslow's hierarchy of needs sees self-actualization as the top need [69]. McClelland argues that motivation comes from a mixture of affiliation (society based), authority (opportunities to gain it), and achievements (overcoming challenges) [71]. The equity theory claims that motivation might be hurt due to relative comparison and the feeling of not being fairly treated [3].

In the context of work, the Goal Setting Theory claims that challenging yet achievable goals benefit the motivation [67]. Close in spirit is Vroom's Expectancy Theory [101] that claims that one estimates the outcome, the outcome value, and the probability of the value. Given these, the motivation is determined, and one will have more motivation in tasks where an outcome that is valued is likely to be achieved.

Herzbereg et al. suggested the Motivation-Hygiene Theory [49]. According to it, positive motivation is usually due to intrinsic motivators. However, external hygiene factors might lead to the loss of motivation. Hackman and Oldman

suggested the Job Characteristics Theory [45]. They claim that the motivation might come from the job itself, due to the significance, autonomy, skill, identity, and feedback related to the job.

These theories are classical and were introduced long ago, and use different approaches (see comparison [52]). Though they were criticized, they are still beneficial [19].

Demarco and Lister [32], and also Frangos [38], claim that the important software problems are human and not technological. So, many have investigated motivation in software engineering [62, 22, 39, 42].

Open-source development is the collaborative development of software that is free to use and further modify. The best-known non-technical equivalent is Wikipedia. A seminal description of the phenomenon is given in Raymond's "The Cathedral and the Bazaar" [81]. Payment is probably the most common way to motivate people to perform a task, though it is an extrinsic motivation and therefore its influence is more complex [25, 82]. However, it is common to perform open-source software development as a volunteer, which means that salary is not the motivation, making it startling from an economical point of view at first sight [63]. Therefore, the motivation of open-source developers was investigated as a specific domain, in an effort to uncover other motivators [105, 100, 31, 65, 47, 83].

Empirical research also supports the benefit of motivation. Task significance, a motivation cause, was shown to increase productivity [43]. Campbell et al. see performance as a function of motivation but also of knowledge and skill [28].

2.2 Motivators

The research literature has not produced a canonical agreed list of factors that influence motivation. Mayer et al. reviewed 75 years of motivation measures [70]. This showed that many different factors have an effect, but the agreement between them is limited. We therefore needed to select which ones to include in our study.

We based our list of motivators mainly on Beecham et al.'s review of motivation in software engineering [22] and Gerosa et al.'s [41] work on motivation in open-source development. Our default was to include motivators in order to cover more aspects of motivation. The motivators that we chose have a long history going back to Herzberg's Motivation-Hygiene Theory [49], and therefore were thoroughly investigated over the years (e.g., in general [48] and in software development [30]). Note that we excluded some of the motivators which are less relevant to open-source development, like "Job security" and "Company policies". Conversely, we did include "hostility", which is a *de*motivator (it is common to refer to factors of positive influence as motivators and those of negative influence as demotivators) [3]. Since we have only a single demotivator, we use the term "motivator" to refer to both it and the positive motivators.

In Section 5.2 we list and discuss the 11 motivators used in our study. In Section 6.1 we compare our results and the results in the survey from which we selected our motivators.

2.3 Reliability of Motivation Reports

The limited reliability of motivation reports, a problem that we also cope with, was investigated in prior work. Using self-estimation in a survey might be a threat to the validity of the collected data. There might be biases due to ego defenses[19], the Dunning–Kruger effect [58], subjectivity, and different personal scales. Further, "research on self-esteem (Shavit & Shouval, 1980) [90] has demonstrated empirically that individuals resist lowering favorable self-perceptions" [27]. Previous work has tried to evaluate these difficulties.

Argyle [12] checked the reliability of self-estimation of happiness and showed it is related to peer and supervisory estimation. The Maslach Burnout Inventory validated self-estimation on burnout by comparison with the answers of a close person such as a spouse or a co-worker [68]. Judge et al. [55] also compared a person's and significant other's answers. For work answers "The average correlation between the self and significant-other reports, corrected for unreliability, was r = .68."

Wigert and Harter investigated performance reviews, an area close to motivation [103]. They mention methodological difficulties when one tries to rely on supervisory estimation instead of self-estimation: individual supervisory ratings are a much less reliable measure of performance than objective measures [99], and 62% of the variance in ratings can be attributed to rater bias, while actual performance accounts for just 21% of the variance [86]. Yet, Tsui reports that an employee and his manager's evaluation of effectiveness match [84].

Beatty et al. [20] also compare manager and employee's appraisals. They found that there is agreement on medium performance and some disagreements on high and low performance. In a second usage there was higher agreement, though it was not clear if it was due to clarification of requirements or just better communication.

As prior work shows, there is a moderate agreement between self-reports and a close person's report. This supports the self-reported answers validity yet warns that they are not perfectly accurate. In this study, we compare the same person's answers to related questions, and the same person's answers in the original and follow-up surveys, reaching a similar agreement level. We also note that despite all the above concerns, Scott et al. report that Facebook found that surveys are twice more accurate than predictive analytics in employee churn [85].

3 The Survey Instrument

3.1 Design

Our design aimed to serve some goals. This led to the construction of a relatively long survey with 66 questions (see Appendix A). The first goal was to obtain labels about quality and productivity in development. The first section of the survey was "Questions regarding yourself" (18 questions). This part included general questions about motivation and verifiable questions about conduct (e.g.,

the writing of detailed commit messages). We also asked questions about self-rating of skill. After collecting the answers we found out that developers tend to have a very good opinion about their performance so we ended using these questions only to investigate answer validity problems (Section 6.2). Few of the questions (e.g., "I enjoy software development very much") were used to indicate the existence of a motivator, later analyzed for influence on general motivation.

The second goal was to obtain information about motivators. This served us for predictive analysis of answers, co-change analysis of motivation improvement, and labeling functions based on behaviour cues in real software development [10]. Due to that we aimed to represent many motivators and use few questions for the main ones. The questions were in the second section "Questions regarding activity in a repository" (28 questions), in which we asked about a specific project and its related behavior. This included our motivation ground truth question: "I regularly have a high level of motivation to contribute to the repository" (based on [73]).

We planned to validate our results with respect to the popular Job Satisfaction Scale questionnaire [50] (10 questions) which was included as is. Analysis and direct feedback indicated confusion, so we ended up using it just for validity. Hence, it served as a source of attention-check items [87]. In questions about the community, we asked people in a single person project to skip, also serving as attention checking. These cases are discussed in Section 6.2. Since the mistakes were due to misunderstanding, as some of the participants directly commented, we included participants that answered so and just measured the how common these mistakes were.

We also added a "Demography" section (8 questions). Demography is of interest on its own and enables us to compare it to the Stack Overflow survey. Last, we ask an open question requesting comments whose goal is to ensure that we did not miss a significant factor in the structured questions.

By using questions from prior work, we benefit from a previous validation. In the discussion below we note prior work on each motivator (Section 5.2), and in the replication package we identify the specific source of each survey question [9].

An important goal of the survey was to enable us to compare answers regarding motivation and actual behavior. For example, we could compare answers to "I write detailed commit messages" and actual commit message length. Therefore, we asked the participants to choose a specific project and provide its name, preferably a public GitHub project. We asked for their GitHub profile for investigating the developer behavior. We also asked for the email from participants who were interested in the research results. Email and GitHub profile are personally identifying information, which is usually not collected. We needed them to match the answers to other data related to the same person, but do not include them with the other experimental materials. This was approved by our IRB (study 09032020).

The survey was designed to take about 10 minutes. Most questions used a Likert scale [66, 53]. Values ranged from 1 to 11, providing closest-to-normal distribution [64]. All participants saw the sections in the same order, yet the

questions order within the sections was randomized.

3.2 Execution

The target population in software engineering is hard to define in many cases [17]. We started with a well-defined population - active developers in large active GitHub projects. We approached only developers that published there email as public, and we obtained a list describing the population by name. This population was essential in order to match answers about motivation and actual behavior in GitHub. However, this population is small compared to all developers. We therefore also reached developers in social media, which led to 80% of the participants. We used machine learning to differ between the population and did not find a large difference (see supplementary materials).

The survey was conducted using the Qualtrics platform from December 2019 to March 2021. We obtained 1,724 participants, 521 of them completed the survey.

We conducted a pilot until February 2020, including 16 people from our social circles. We received feedback on the focus of the survey and redundant questions, which were deliberate decisions. Few participants alerted us about the principal position of GitHub, feedback we failed to use at this stage. We received wording suggestions that we applied. We checked that the long survey can be completed in 15 minutes, which was OK. After the entire survey was conducted, we found out that 84% of those that finished it did it in this duration. Since the content of the questions was not changed, we included the pilot participants in the survey.

GitHub is a platform for source control and code development used by millions of users [1]. We initially focused on 1,530 active public GitHub projects with 500+ commits during 2018, described in [7, 8, 5]. About 40,000 developers contributed to these projects that year, of which 9,000 contributed more than 12 commits. We extracted developers' email addresses using the GitHub public email API, fetching the emails of the developers that chose to share them publicly. We sent emails to 3,255 developers with a public email that had enough commits. We also had a gift card lottery, offering $50 to three of the participants. This channel led to 339 participants, which is 20% of the total.

We also recruited participants in social networks by convenience sampling [33, 2], which led to the remaining 80% of the participants. We used Reddit, an online discussions site, as an important source of participants. Reddit has numerous subreddits, channels dedicated to discussions on specific topics. It has many channels relevant to programmers such as programming language based, operating system based, tools, etc. We posted slowly in different subreddits, to get familiar with the community, as posting at a high rate might be considered as spamming. Each subreddit has different formal and informal rules that should be respected. We found the people showed interest in the survey, which led to discussions and upvotes. These in turn led to more attention to the survey.

As noted above, we asked participants for the name of their project and their GitHub profile. They provided the names of 484 projects and 303 personal

GitHub profiles. But after posting the survey in social media, we noted that many participants stop at the "Questions regarding activity in a **repository**" section since they do not contribute to a GitHub repository. This was also accompanied by direct feedback saying that. Since we were interested in answers about motivation in general and needed the GitHub profile only to link to actual behavior, we changed the questions about "GitHub repository" to "any project", avoiding this drop.

The original survey ended in March 2021. A year after the last response, in April 2022, we sent a follow-up survey to the 341 participants that provided their emails in the original survey. We sent them the name of the project on which they initially answered, and asked to answer on the same project in case that they are still active there and on a different project otherwise. The questionnaire was the same as the first one, with the additional validation question "Is it the same project on which you answered last time?". In the follow-up survey, 124 out of the 341 participants we reached out to answered (36.3%).

3.3 Answers Treatment and Validation

Our emails to GitHub developers had 4.4% response rate, close to the 5% reported by [51]. Due to a mistake, we sent multiple emails to developers that worked on multiple suitable projects, and we apologize for that. This was both annoying and mis-estimated the number of developers. After fixing the mistake the response rate was 8.1%, close to the 9% reported by Feitelson which used a similar dataset [35]. We believe that the response rate benefited from the use of the GitHub public email API. Email is used by GitHub in the development, and therefore they are usually updated (our bounce rate was 1%). The API returns the emails of developers that choose to share them publicly, hence more agreeing to cold communication. We believe that many people found the topic, motivation in software development, exciting. We anecdotally noticed it in up-votes and comments in social media, responses to the open questions in the survey, and the 19.8% of the participants that left email wanting to learn the research results.

We checked if participants answer all questions with the same answer (e.g., due to maliciousness or boredom). There was a single such participant, who answered a few "Questions about yourself" and dropped, not included in analysis anyway. 91% of the participants that did not finish the survey, abandoned in less than 5 minutes. Only 7% answered the survey in less than 5 minutes, a quick yet possible duration.

Some participants do not answer all questions. In these cases, we used the relevant answers and did not attempt to fill the missing ones.

Section 6.2 presents validity problems in the answers. Section 6.3 presents the internal coherence of motivators and their construction. Section 6.4 uses the follow-up survey and treats the survey as a longitudinal study and shows that their agreement is better than independent.

However, the strongest validation comes from the ability to predict the behaviour of 151,775 developers in 18,958 GitHub projects. Amit and Feitelson

used our dataset to define labeling functions [26, 80, 11] for motivation [10]. The labeling functions are weak classifiers, validated heuristics that predict motivation better than a guess, which are based on behavioural cues. For example, working in more diverse hours and writing longer commit messages are labeling functions. These labeling functions predicted the high answers of our survey participants. When used on the entire GitHub datasets, they predicted retention, and higher activity and output of 151,775 developers. Hence the answers agree with the labeling function, which agree with the behaviour of a mass of new unseen developers.

All data and code are in the supplementary materials. Personal identifiers were hashed to preserve anonymity yet allow matching.

4 Analysis

Supervised learning is a powerful framework, suitable to our needs. In supervised learning we try to predict a concept (e.g., high motivation) using a classifier (e.g., decision tree) based on features (e.g., motivators). The versatility of supervised learning allows us to investigate a single motivator, all of them, or temporal changes. One can apply supervised learning to a single motivator for direct evaluation or to all of them, leverage their combined power.

The concept, which we try to predict, is high motivation. The classifiers, which provide us with predictions, are the motivators (e.g., high ownership). We evaluate how well motivators predict high motivation, using metrics that compare the prediction to the actual motivation. Interesting metrics are the fraction of those with, say, high ownership who indeed are highly motivated (precision), the improvement over just the prevalence in the population (precision lift), and what fraction of the highly motivated who have high ownership (recall).

The analysis of each individual motivator with respect to general motivation provides simple basic results yet ignores more complex relations like confounders. We therefore also performed additional, more complex analyses. We analyzed the relations between motivators to see that this risk is small. We used the follow-up survey to analyze motivation improvement of each motivator alone. Last, we built combined models utilizing all motivators to avoid risk of confounding and leverage the power of all motivators.

5 Results

5.1 General Motivation

Motivation might derive from many motivators, from payment to enjoyment. The concept that we would like to investigate is high motivation to contribute to a project, and its relations to these various factors.

We measure general motivation using the question "I **regularly** have a **high** level of motivation to contribute **to the repository**" (pattern is based on [73]).

The results show that developers are generally motivated (Figure 1). High motivation (at least 9 = ‘somewhat agree’ on a scale of 1 to 11) was reported by 52.4% of the participants.

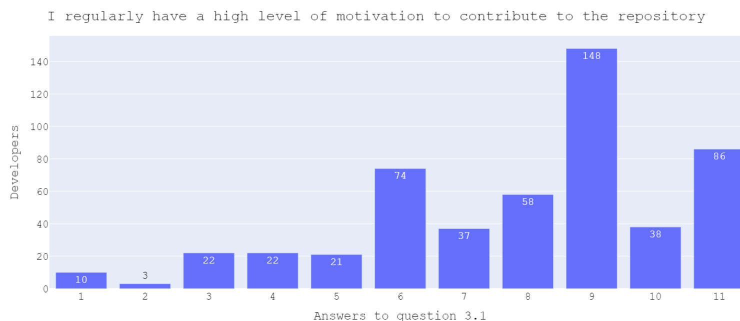


Figure 1: Distribution of answers to general motivation question.

We identified paid people by their answer to a specific question about it (question 3.c). 41% of the participants that answered this question said that they are paid (284 participants). 24% of the participants (377) that had GitHub in their project name were identified as contributing to it (the question encouraged providing a GitHub project when possible). Note that being paid and using GitHub are not mutually exclusive. 38% of participants of GitHub projects that answered the payment question, said that they are paid.

It is generally accepted that motivated workers work longer hours [21]. Showing that our measurement of general motivation exhibits the same relation provides supporting evidence to its validity. And indeed, participants reporting high motivation (at least 9) reported an average of working 19.3 hours a week on the project, compared to 3.4 hours for those reporting low motivation (below 9). This result may be tainted by mixing data about paid developers with data about volunteers, who are common in open-source projects. We checked this by looking at paid workers separately, out of the participants that reported payment, and the influence of payment is indeed large. For unpaid workers the reported average working hours were 10.8 (high motivation) and 4.5 (low), while for paid workers they were 27.9 (high) and 25.8 (low). Thus, both paid and unpaid participants work more hours when motivated yet the average of unmotivated paid developers is higher than that of motivated unpaid developers and therefore we do not mix them.

We planned to also validate the measurement with the similar question from the Job Satisfaction Survey: “Taking everything into consideration, how do you feel about your work?” [50]. However, as discussed in Section 6.2, some of the participants were confused and answered most survey questions on an open-source project to which they contribute, yet answered the Job Satisfaction Survey on their regular job. Despite this confusion, the Pearson correlation between the questions is 0.32. When focusing on paid developers, for which

the probability of confusion is lower, the correlation is 0.36. This correlation is similar to the inner correlation in the other motivators as shown below.

5.2 Motivators

Table 1 summarizes the predictive performance of all the motivators. We discuss each of the motivators in the following subsections. Note that in each row we analyzed participants that answered the motivation question and at least one question about the motivator, so populations are not necessarily identical.

We use common metrics used in machine learning and information retrieval. The concept that we want to predict is high general motivation. This was operationalized by the answer to the question "I regularly have a high level of motivation to contribute to the repository" being 9 'somewhat agree' or above.

Usually in machine learning a classification algorithm (e.g., decision tree) is used to create a model, which is a specific rule providing predictions (e.g., if A & *not* B). In contrast, our models are the motivators (e.g., ownership, challenge), also binarized into high and low using 9 as the threshold. Note that in this part the models are pre-defined, and not learnt by a classification algorithm, and we only evaluate their predictive performance.

The cases in which the concept is true are called 'positives' and the positive rate is denoted $P(positive)$ (in our case this is 0.52 as noted above). Cases in which the model is true are called 'hits' and the hit rate is $P(hit)$. For example, a high hit rate for ownership means that many participants report ownership, and we want to see whether they are also generally motivated.

Ideally, hits correspond to positives, but usually some of them differ. Precision, defined as $P(positive|hit)$, measures a model's tendency to avoid false positives (FP). But precision might be high simply since the positive rate is high. Precision lift, defined as $\frac{precision}{P(positive)} - 1 = \frac{P(positive|hit) - P(positive)}{P(positive)}$, copes with this difficulty and measures the *additional* probability of having a true positive relative to the base positive rate. Thus, a useless random model 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 highly motivated participants also report high ownership.

We now present our analysis of each individual motivator, from the most to the least prevalent. We show how common high answers (9 and above) are to each motivator, in general, for paid developers, and for open-source developers.

5.2.1 Enjoyment

We measure enjoyment [41, 44, 12] by the following questions (numbered by their location in the survey):

2.9 I enjoy software development very much

2.15 I enjoy trying to solve complex problems

Table 1: High Motivation Predictability by Motivator

Motivator	Hit rate	Performance as predictor of motivation			
	(Fraction ≥ 9)	Accuracy	Precision	Prec. lift	Recall
Enjoyment	0.74	0.64	0.62	0.18	0.86
Ownership	0.73	0.59	0.57	0.10	0.81
Learning	0.72	0.59	0.58	0.10	0.80
Importance	0.63	0.61	0.61	0.16	0.73
Challenge	0.62	0.63	0.62	0.20	0.74
Self-use	0.56	0.60	0.61	0.17	0.65
Ideology	0.53	0.57	0.59	0.13	0.60
Recognition	0.48	0.58	0.60	0.18	0.56
Payment	0.45	0.55	0.58	0.10	0.49
Community	0.41	0.63	0.67	0.35	0.53
Hostility	0.07	0.52	0.65	0.30	0.08

3.8 My work on the repository is creative

3.10 I derive satisfaction from working on this repository

An example of the results is shown in Figure 2.

We calculated the average answer to all these questions per participant, and then the average of these averages. This led to an overall average of 9.07. 74% of the participants reported high enjoyment (at least 9 - 'somewhat agree'), more than all other motivators.

76% of the GitHub participants reported high enjoyment and 75% of the paid participants. The correlation of enjoyment with motivation is 0.51, the highest of all motivators.

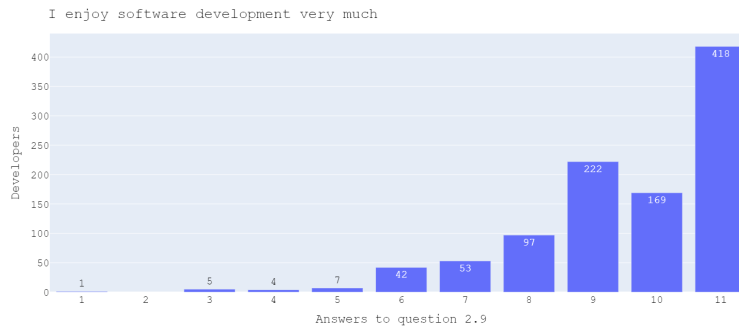


Figure 2: Answers distribution of enjoyment question.

The recall of enjoyment is 0.83, making it very common among people of high motivation, more than both the hit rate and the positive rate. The precision is

0.62 and the precision lift is 0.18, which is positive and significant yet not very high.

Note that the hit rate of 74% for enjoyment is significantly higher than the positive rate of 52%. This has a large influence on the predictive metrics. True positives are the intersection of the hits and the positives, so they are bounded by both. But once the hit rate is higher than the positive rate, the precision is bounded from above. In our case, the model hit rate is 74% and the positive rate is 52%, so the model precision can be at most $\frac{52}{74} = 70\%$. The actual precision is 62% which is not high yet is 89% of the bound created from the positive rate and the hit rate.

5.2.2 Ownership

We measure ownership [22, 41, 60, 16, 72] by the following questions:

3.2 - I have complete autonomy in contributing to the repository

3.3 - I have significant influence on the repository

3.4 - I feel responsible to the repository success

3.16 - I am a core member of the repository

The average answer for ownership was 9.02. 73% of the participants reported high ownership (9 or above, second highest of all motivators). This was also the percentage for paid participants; with GitHub participants it was 75%. The correlation of ownership with motivation is 0.24. The recall when predicting high motivation based on high ownership was 0.81, higher than the hit rate. However, the precision is 0.57 and the precision lift is only 0.10, partly since ownership is so common.

5.2.3 Learning

Learning [22, 41, 18] is based on the question:

3.17 - I learn from my contributions

The average of the answers was 9.15, the highest among all motivators. 72% of the participants reported high levels of learning, 70% of the GitHub participants and 77% of the paid ones. The correlation of learning with motivation is 0.23. Learning has a recall of 0.80, indicating that it is another very common characteristic of people with high motivation. Its precision is 0.58 and its precision lift is 0.10, which is relatively low.

5.2.4 Importance

Importance [22, 45, 43] is based on the question:

3.11 - The repository is important

The average was 8.62. The correlation of importance with motivation is 0.35. 63% of the participants report high feeling of importance. The same occurred among GitHub participants, compared to 74% among paid participants. This is rather surprising since we assume that one will have more considerations and constraints in the context of a paid job than in volunteering to open-source projects. So, we would assume that one would have higher freedom to choose by importance when volunteering, leading to a higher rate among GitHub participants, but the data shows the opposite. Importance has precision of 0.61 and precision lift of 0.16. It has a recall of 0.73, which is high in absolute terms and relative to its hit rate.

5.2.5 Challenge

Challenge [22, 71, 67, 94, 78, 89] is based on the question:

3.9 - Working on this repository is challenging

The average of challenge answers was 8.41. 62% of participants reported a high sense of challenge, 60% of the GitHub participants and 66% of paid ones. The correlation of challenge with motivation is 0.30. Challenge has precision of 0.62 and precision lift of 0.20. Its recall is 0.74, which is high in absolute terms and relative to its hit rate.

5.2.6 Self-use

Self-use [41, 81] is based on the question:

3.5 - I'm interested in the repository for my own needs

The average of self-use answers was 7.86. 56% of the participants reported high self-use motivation, 61% of GitHub participants and 43% of paid ones. Note that while usually the probabilities in the entire population, in GitHub, and among paid participants are rather similar, in this case the probabilities are quite different. 'Scratching your own itch' is a well known motivation in open source [81] so one would expect a higher probability in GitHub. On the other hand, many companies produce organizational software that does not have personal uses, so 43% of paid participants which self-use may sound rather high. The correlation of self-use with motivation is 0.16. Self-use has a recall of almost two thirds, 0.65. This seems to be a unique attribute of open source, enabling people to develop the software that they need. The precision is 0.61 and the precision lift is 0.17. This might be since people see satisfying their need as a task to complete and not an enjoyable activity. Indeed, self-use and enjoyment have a Pearson correlation of just 0.16.

5.2.7 Ideology

Ideology [41, 24] is based on the question:

2.18 - I contribute to open source due to ideology

53% of the participants reported high ideology-based motivation, rising to 61% of GitHub participants, aligned with the ideological roots of open-source development [24]. 49% of paid participants also gave high answers regarding contribution due to ideology. That can be either due to many people being paid to contribute to open source, or a common habit of paid developers to contribute to open source in their free time. Regardless of the reason, the popularity is surprisingly high. But we note that in the answers to the open question participants said that different ideologies (e.g., 'Software should be free' [93], 'Social good') can lead to contribution to open-source software, and that a finer distinction is needed. The average of ideology answers was 7.34. The correlation of ideology with motivation is 0.14, the lowest other than for hostility. The precision of ideology is 0.59, the precision list is 0.13, and the recall 0.60.

5.2.8 Recognition

We measure recognition [22, 41, 82, 81, 100] by the following questions:

2.13 I contribute to open source in order to have an online portfolio

2.14 I try to write high quality code because others will see it

3.15 I get recognition due to my contribution to the repository

3.24 In the past year, members of my GitHub community asked questions that show their understanding of my contributions (based on [57])

3.25 In the past year, members of my GitHub community expressed interest in my contributions (based on [57])

The average of all the questions was 7.33, lower than all positive motivators. 48% of the participants reported high recognition-based motivation, 49% of the GitHub ones, and 52% of paid ones. The correlation of recognition with motivation is 0.27. The precision is 0.60 and the precision lift 0.18, indicating a boost to motivation. The recall is 0.56 while the recognition hit rate is 0.48.

5.2.9 Payment

Payment [22, 41, 25, 82] is based on the yes/no question:

3.c - I'm being paid for my work in this repository

We note that remuneration in open-source projects may have many facets. Developers may accrue income from donations or lectures. Their work on the project may help them secure future positions or gain access to future consulting contracts. In the interest of simplicity and precision we use the objective criterion of specifically being paid a salary to define payment.

41% of the participants that answered the payment question said that they are paid. 38% of participants of GitHub projects that answered the payment question said that they are paid.

Payment has precision of 0.58 and precision lift of 0.10, making it one of the weakest motivators in general and the weakest for its hit rate. Payment is also the only motivator that had a negative precision lift when we run the same analysis on the follow-up survey.

Its recall is 0.49, less than the positive rate of high motivation which is 0.52. The correlation of payment with motivation is 0.15, lowest than all but ideology and hostility.

5.2.10 Community

We measure community [22, 41, 71, 23, 15, 105, 36] by the following questions (which we asked to answer only if you are not the only developer in the project):

3.13 Belonging to the community is motivating my work on the repository

3.14 The community is very professional

3.20 The repository's community of developers is more motivated than that of other repositories

3.24 In the past year, members of my GitHub community asked questions that show their understanding of my contributions (based on [57])

3.25 In the past year, members of my GitHub community expressed interest in my contributions (based on [57])

Note the questions 3.24 and 3.25 are about recognition from the community and therefore appear in both motivators.

The average of community answers was 7.36. 40% of the participants reported high community-based motivation, lower than all positive motivators. This is based on 40% of the GitHub participants and 44% of the paid ones. The correlation of community with motivation is 0.42, the second highest. The precision is 0.67 and the precision lift is 0.35, higher than all other motivators. Hence, though the community motivator is not common, when it exists, the probability of high motivation is higher. The recall is 53%, not very high yet 29% higher than the hit rate.

5.2.11 Hostility

Hostility can be viewed as a community with negative influence. Hostility hurts motivation hence it is a demotivator and not a positive motivator. We measure hostility [79, 3, 49, 29] by the following questions (which we asked to answer only if you are not the only developer in the project):

3.6 We have many heated arguments in the community

3.7 I wish that certain developers in the repository will leave

3.22 In the past year, members of my GitHub community put me down or were condescending to me (based on [29])

3.23 In the past year, members of my GitHub community made demeaning or derogatory remarks about me (based on [29], Figure 3)

The average of hostility answers was 2.80. Only 7% of the participants reported high hostility, 4% of the GitHub participants and 7% of the paid ones. The recall is just 8%, yet it is higher than the hit rate. The correlation of hostility with motivation is 0.01. This is the lowest correlation, close to zero yet not the large negative correlation which is expected.

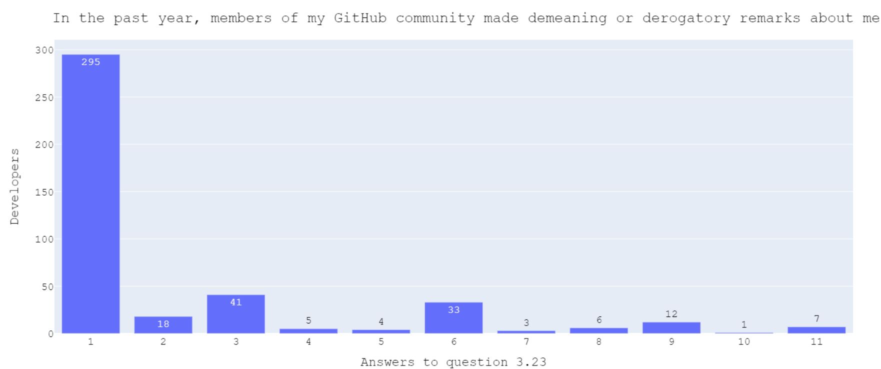


Figure 3: Answers distribution of a hostility question.

Surprisingly, hostility also has a relatively high precision of 0.65 and a high precision lift of 0.30. One would expect that knowing that someone suffers from hostility will *reduce* the probability of high motivation, instead of the increase that we see. But the participants who reported high hostility also reported higher averages for all motivators besides payment. A possible explanation is that those participants kept contributing to the project due to the other motivators; those who suffered from hostility and did not have other reasons to stay probably left. Note, however, that we had only 9 people that reported both high hostility and high motivation, so the analysis is not robust.

We identified 10 pairs of developers which contribute to the same project. This allowed us to evaluate their agreement on hostility. Surprisingly, when a person reports heated arguments (question 3.6), the probability that the other participant will agree is just 50%. For the rest of the hostility questions, the other participants never claimed high hostility too. For comparison, in importance and challenge, which also describe aspects of the project, if one participant provided a high half answer (6 or above), the other always agreed. This provides an important indication that hostility might go unnoticed.

5.3 Motivation Improvement Analysis

An important goal in many data analyses is to uncover causal relations. But causality is hard to define rigorously because it is hard to ascertain that motivator A caused outcome B. The usual approach is to look at correlations between

motivators and outcomes in a given dataset, as we did in Section 5.2. We now extend this to look at the *dynamics across time*: the possible correlation between *a change* in a motivator and *a change* in the outcome. For example, we want to see whether an increase in the sense of ownership of a project predicts an increase in motivation.

Such "co-change" analysis [8] is important for the following reason. If causality exists, meaning that in certain contexts motivator A causes outcome B, then a change in A will cause a change in B. But co-change of two motivators does not necessarily imply causality. By identifying instances of co-change, where a change in A correlates with a change in B, we therefore identify cases where causality may be at work.

Note, however, that the actual relationship between motivators and general motivation may be conditioned on other motivators. In this subsection we look at the co-change of motivators and general motivation alone, regardless of context. In the next subsection we consider all the motivators together, to handle cases where the effect of A on B is conditioned on another motivator C.

The change data comes from comparing the original survey and the follow-up survey. In the original survey 341 developers provided their emails. A year after the last response, we reached out and asked them to answer the survey again. We asked them to answer on the same project if they are still active in it. This allowed us to compare the answers of the same person over time. We had 124 follow-up participants in total. 60 of them continued in the same project, and these are the ones we analyze here. For each of them, we look at increases in the motivators and general motivation from one year to the next. Note that if a person reported 3 for ownership in the first survey, and 4 in the follow-up, this is an increase regardless of the values being low.

Table 2: Motivation Improvement Over Time Predictability by Motivator

Motivator	Improvement rate	Prediction of improved motivation			
		Accuracy	Precision	Prec. lift	Recall
Challenge	0.33	0.53	0.10	-0.50	0.17
Ideology	0.30	0.60	0.17	-0.17	0.25
Importance	0.30	0.70	0.33	0.67	0.50
Learning	0.30	0.70	0.33	0.67	0.50
Enjoyment	0.28	0.71	0.34	0.71	0.48
Recognition	0.27	0.72	0.39	0.95	0.47
Self-use	0.22	0.78	0.46	1.31	0.50
Ownership	0.17	0.77	0.45	1.27	0.35
Hostility	0.16	0.70	0.20	-0.01	0.15
Community	0.16	0.75	0.39	0.93	0.28

The probability of improvement in general motivation, i.e. the positive rate, was 20%. Since we included only developers that stayed in the same project for at least a year, there is probably survivorship bias, and the probability in the whole population is probably even lower.

The probability of improvement in the different motivators is at most 33% (Table 2). When the precision lift is positive, it tends to be very high. We could not find out why the lift is negative for challenge and ideology. One could expect a larger negative lift for hostility, which did not materialize. This may be explained by developers having other motivators that offset hostility as explained in Section 5.2.11.

The recall is up to 50% for many motivators and higher than their improvement rate (hit rate). This shows that improvement in importance, learning, enjoyment, recognition, and self-use are common when motivation improves.

Only a single person that was not originally paid received a payment in the follow-up, therefore we did not apply co-change analysis to this motivator.

Co-change analysis can be performed in the downward direction too: given a decrease in a motivator, how common is a decrease in general motivation. Results are quite similar to the upward direction and given in the supplementary materials.

The follow-up survey can also be considered a replication of the original survey. We used all 124 participants that answered the follow-up survey to run the analysis of predicting motivation (as in Table 1). We found a positive precision lift for all motivators besides payment. The agreement supports the results in general. The disagreement in payment indicates that the result is not robust.

5.4 A Combined Motivation Model

All the motivators have a positive precision lift of at least 10% (Table 1). This is aligned with the prior work claiming their positive influence. However, the highest precision lift is just 35%, with 67% precision, for the 'Community' motivator. This means that none of the motivators is a sufficient condition for high motivation or close to it. We analyzed the Pearson correlation between motivators (in supplementary materials) and noted that none of the correlations is very high. Hence, each motivator describes a different aspect, and a combination of motivators might help to reach high motivation.

The inputs of machine learning models are named "features". So far, we investigated each motivator as a single feature, ignoring all other motivators. This type of analysis suffers from the threat of confounding variables on one hand and does not leverage the full power of the data on the other hand. We therefore built combined models to predict motivation, based on all the motivators as features.

Since exact values tend to change (see Section 6.4) and we are indifferent to the level of low motivation, we prefer classification framework over regression. The predicted concept was high motivation, operationalized as before by answers of 9 ('somewhat agree') or above to "I regularly have a high level of motivation to contribute to the repository". We had 345 participants that answered this question. The positive rate is 52%.

We used the scikit-learn package for classification algorithms [75]. We used low-capacity small models such as decision trees and logistic regression in order

to obtain simple interpretable models which are also rather robust to overfitting [14]. We also used models of medium capacity such as random forests, boosting, and neural networks to build models of better representation ability and performance. When splitting samples into train and test, we would like to have enough samples for reliable estimation in the test, and use the rest for train since models perform better with more data. The number of samples we have is very low. Repeating with 10 different seeds we noted that the standard deviation on both the accuracy and the positive rate (which is not influenced from learning) is 0.03. We therefore used 30% for reasonable test estimation and used the rest for training models that are suitable for small datasets.

The performance of all the models was rather close, with accuracy ranging from 62% to 77%. Amusingly, the highest accuracy on the test set (77%) was reached by a single node tree, checking high enjoyment. As Table 1 showed, the accuracy when using enjoyment on the whole dataset is just 64%, so this result is accidental. The model with the second highest accuracy (76%) was a neural network [61], whose capacity is relatively high. The simple model of highest accuracy was a logistic regression model [98] which reached accuracy of 72%. Its intercept was -2.04, indicating a general tendency for low motivation. Hostility had a strong negative coefficient of -0.46. All the other motivators had positive coefficients. The highest were enjoyment with 1.13, self-use with 0.61, and importance with 0.59.

Note that models can assign different weights to false positives and false negatives, and trade off precision and recall. Using this, we could build a precision-favoring decision tree model [77] with precision of 81% and recall of 41%. Conversely, a recall-favoring stochastic gradient descent (SGD) model [106] reached recall of 96% with precision of 62%.

We also modeled the co-change dataset of Section 5.3, to predict a change in the motivation based on changes in the motivators as features. Such a model is of interest since assuming that motivation is a function, a co-change model can predict the result of a change.

Two properties of such modeling deserve special attention: accuracy and minimality. We first discuss accuracy. A co-change model allows us to predict *the motivation change* given any *motivator's change*. Perfect accuracy assures us that there are no other external causal variables influencing the samples in our dataset. Assume by contradiction such a variable c, other than the model variables. Hence there is a behavior function g and an assignment of values such that $g(c_1, v_1, ...v_n) \neq g(c_2, v_1, ...v_n)$ where $v_1, ...v_n$ are the values of the model variables. However, since we have perfect prediction given the model variables, it should be that $g(c_1, v_1, ...v_n) = m(v_1, ...v_n) = g(c_2, v_1, ...v_n)$ — a contradiction. Hence such a variable cannot exist. Perfect accuracy is rare, and mostly indicates a problem in the analysis and not capturing all causal variables. However, the accuracy bounds the influence of such external variables.

As for minimality, consider decision trees [77] as models. For each leaf, variables that do not appear in the path to this leaf do not influence the prediction. On the other hand, each variable along the path is necessary, and a change in its value will change the prediction. In this sense, the model is minimal and

every variable along the path is required. All the variables that we use here are mutable (can change, in contrast for example to the project creation year). With both perfect accuracy and minimality, the model predicts each change. More than that, the removal of any variable will hurt the prediction of some changes.

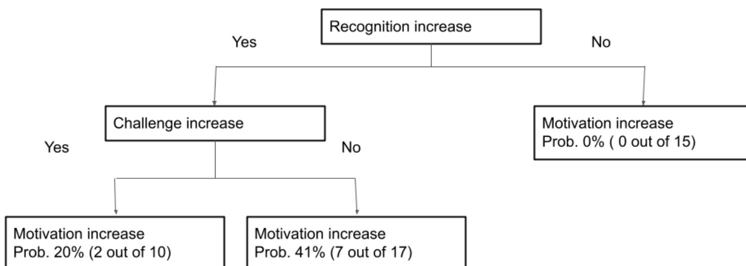


Figure 4: A decision tree predicting motivation improvement

When aiming for high accuracy, we built models based on AdaBoost [40] and Neural Networks [61] which reached accuracy of 94%. Aiming for either precision or recall, 100% were reached. The size of these models is high and far from minimality, which is the price for achieving the high accuracy. Alternatively, we found a small model, presented in Figure 4, whose "recognition increase and challenge decrease" leaf reaches 78% recall with 41% precision. Note, however, that the dataset is a very small dataset with a high VC dimension [97, 96] (due to having many questions and wide scale), and therefore the threat of noise is very high. Also, the dataset probably does not fully represent motivation complexity. A larger dataset will probably better represent human motivation behavior but will require a larger model and have lower performance.

6 Analysis of Validity and Reliability

In the previous analysis we analyzed the data as if it is completely reliable. However, the reliability might be limited in many ways. Since the data is given, what we do in this section is to evaluate its reliability from various aspects.

When using the answers of participants, one should check which population they represent. We compare our survey demographics to the demographics of the Stack Overflow survey, answered by around 80 thousand developers worldwide. Distributions are not similar yet close. We used two channels in order to reach participants: direct emails to developers contributing at GitHub and social media. We build supervised learning trying to differ them, whose performance was no better than the positive rate, indicating no big obvious difference. These are detailed in the supplementary materials.

We compare our results to other surveys, the investigate their agreement

(Section 6.1). We grouped questions into motivators based on their content, regardless of the answers to them. In Section 6.3 we examine the coherence of the motivators and compare them to grouping based on the answers. The follow-up survey allows us to evaluate the stability of answers, comparing a person's answer in two different dates (Section 6.4). It also provides an additional dataset on which we can check the degree in which our results reproduce. Last but not least, we investigate reliability in the answers themselves, from typos to mistakes and biases (Section 6.2).

6.1 Surveys' Hit Rate Comparison

In Table 3 we compare our survey to prior work on motivation. Most prior work provides hit rates of motivators, forcing us to use the same metric for the comparison. Note that hit rate indicates how common motivators are and not their relation to general motivation. Moreover, some works provide only the ranking of motivators. In case the hit rate itself was provided, we present it in parentheses. Gap is the difference between the minimal and maximal rank of a motivator in the different surveys, serving as a diversity metric. Also note that in some cases different names or even just overlapping concepts were used in the prior work, detailed in the replication package.

Note that the surveys are from different years, from 1978 to 2022. Herzberg investigated the general population, Fitz and Couger investigated IT personnel, Gerosa investigate open-source developers, and we investigated developers in general, many of which were open-source developers. The instruments were also different in most cases. However, if the surveys generalize to describe a universal behavior, they should agree.

We computed the Pearson correlation between the average ranks in all surveys and the ranks in each individual survey. The result was above 0.8 for Fitz and for our survey, about 0.7 for Couger and Herzberg, and 0.64 for Gerosa. Gerosa is closer to us in time and population so a higher correlation is expected. Our ranking and our followup ranks had a correlation of 0.96, Herzberg and Fitz had a correlation of 0.87 and the rest were much lower. For example, Couger that reproduced the work of Fitz 10 years later, had a correlation of 0.67 with it.

When looking at the hit rates themselves, even just browsing the table shows that in general the distributions are quite different. We get a 0.97 correlation between our survey and our follow-up. Gerosa has 0.65 correlation with ours, 0.51 with the follow-up. Herzberg has 0.24 correlation with ours, 0.38 with the follow-up, and -0.21 with Gerosa.

Enjoyment is ranked high in all surveys, with a small gap between the maximal and minimal rank, *indicating a stable high result.* Community and payment are rather low, and not ranked high in any survey. Ownership, Recognition and Learning have large gaps, hence results regrading them in one survey do not generalize well to the others. It might be possible achieve higher agreement by applying transformations considering different times (e.g., importance of payment is reduced) or different populations (e.g., payment is less important

Table 3: Comparison of Motivators Hit Rates and their Ranks in Different Surveys

Motivator	**Avg. Rank**	**Fitz [37]**	**Couger [30]**	**Herzberg [48]**	**Gerosa [41]**	**Our**	**Follow-up**	**Gap**
Enjoyment	2	2	1	3 (22)	3 (18.9)	1 (74)	2 (71)	2
Challenge	2.60	1	2	1 (40)		5 (62)	4 (64)	4
Ownership	3.80	6	6	4 (20)		2 (73)	1 (76)	5
Learning	4	5	7	5 (5)	1 (22.6)	3 (72)	3 (69)	6
Importance	5					4 (63)	6 (59)	2
Self-use	5				2 (21)	6 (53)	7 (53)	5
Recognition	5.17	3	5	2 (30)	5 (10.6)	8 (48)	8 (46)	6
Ideology	5.67				6 (8.8)	6 (53)	5 (61)	1
Payment	6.50	8	4	5 (5)	4 (16.6)	9 (45)	9 (37)	5
Community	8	7	8	7 (4)	6 (8.8)	10 (41)	10 (32)	4
Hostility	11					11 (7)	11 (6)	

in open source). However, the disagreement is not surprising since Couger [30] already indicated the people in different position report different motivators, and so do people from different countries Herzbereg [48].

6.2 Face Validity of Answers

To check the validity of the answers in our survey, we looked for mistakes, insincere answers, and biases.

The answer to the gender question was a free text field, in which the participant could write any answer. Only 4 (0.8%) of the answers had a typo (e.g., ‘mail’, ‘boi’). Three of the answers were variants of ‘Attack Helicopter’, a term “used to disparage transgender people”[1]. Hence, these answers were probably not sincere. In the country question, 1.2% of the answers had a typo.

1.3% of the developers said they had 15 years of experience with GitHub, established in 2008, which was impossible when the survey ended in 2021. A single answer (0.2%) of age of 100 years is probably insincere. Note that these error rates are much better than the 8.5% who seemed to have given a wrong answer to a single simple question in [46], and the 10% failure to identify negatively worded (reverse-coded) items discussed in [76].

The job satisfaction questions were taken from a survey of 9,900 Australian clinical medical workers published in 2011 [50]. Amusingly, software developers

[1] https://en.wikipedia.org/wiki/I_Sexually_Identify_as_an_Attack_Helicopter

were on average less satisfied in all questions. More importantly, questions about payment are irrelevant to volunteers and questions about community are irrelevant to people working alone. We explicitly asked to skip these questions if they are irrelevant. However, 57.7% of the people that answered that they are not paid, answered the payment satisfaction question. Therefore, it seems they answered regarding their salary from a different job, not related to the discussed project. Some participants made comments in the open question that support this.

Some developers answered that they work on a public GitHub project. For these projects, we checked the number of developers who committed code. Of five developers who were found to work on single person projects, one answered most of the community-related questions, which are irrelevant to such projects.

There were questions in which the change in the follow-up survey is known in advance: age and experience should grow linearly with time. The follow-up question was about a year after the first one. In order to avoid rounding mistakes (e.g., a 20.5 years old participant might answer either 20 or 21), we consider answers as "unreasonable" only if the follow-up answer was more than a year lower, or at least three years higher. 26% of the answers about experience exhibited such unreasonable differences. Two participants lost 5 years of experience each, somehow compensated by a participant that gained 11 years of experience in about a year. 16% of the answers regarding GitHub experience were unreasonable. However, for age, which has a higher presence in daily life, there were no unreasonable differences.

It seems that the biggest reliability problem comes from human failings [76], bias due to ego defenses[19], or the Dunning–Kruger effect (that people with lower capabilities tend to have higher self-esteem) [58]. Only 5.6% of the participants gave a low answer to 'My code is of high quality', going up to 20.8% when including neutral answers (6 on the 11-point scale). The Pearson correlation with years of experience, a common method to estimate skill [34], was a very low 0.06. Moreover, first degree holders gave answers averaging 9.05, higher than all others. People trained in computer science gave answers averaging 9.3, lower than the 10.5 average in math, yet a bit higher than arts (9.0), science (8.7), technology (8.6), and business (7.5).

Using the participants' GitHub profile, we can compare their actual activity to their self-perception. People that answered that they write detailed commit messages (at least 9 - 'somewhat agree'), had average commit message length of 89 characters, placing them in the 61 percentile of GitHub developers, not very far from the median. Participants saying that they write high quality code have corrective commit probability (CCP) [8] of 0.36 (investing more than a third of their work in bug fixing), worse than 81% of the GitHub developers.

It seems that there is also a bias leading to higher answers about the participant than about the community. The average answer for questions about themselves is 9.1, 24% more than the average answer to questions about the project. A somewhat smaller difference of 4.5% to 17.1% was found when the questions were essentially paired (e.g, 'My code is of high quality' and 'The quality of the code in this repository is better than others').

We cannot accurately aggregate the probability of mistakes. The probability of identifying an insincere answer is low, around 0.2%. Mistakes typically occur in few percent of the answers, yet in specific questions (the job satisfaction in our case) might be around 50%. Biases are the largest threat to validity, as demonstrated by the 79% participants that consider their code to be of high quality. Only 5.6% of the participants gave a low answer to 'My code is of high quality', and 20.8% when including neutral. The 5.6% that think that their code is of low quality seem to be either more modest or more realistic.

6.3 Internal Coherence of Motivators

The motivators were represented in the survey by one or more questions each. The use of multiple questions (e.g. for 'community') allows us to treat them as labeling functions of the same concept and evaluate their agreement [80, 11]. The agreement, measured by the average Pearson correlation of the related questions, reflects the internal coherence of these motivators. Low coherence might be due to our subjective grouping of questions or due to human nature.

As a reference of the level of correlation that we can expect, we focus on closely related question pairs. For example, 'I am skilled in software development' has a correlation of just 0.62 with 'My code is of high quality'. 'I regularly reach a high level of productivity' and 'I am a relatively productive programmer' have correlation of just 0.57. Table 5 shows that the correlation of the same person answers to the motivation question in the original and follow-up survey is 0.52. Note that a correlation of 0.5 is even higher than the correlation between LOC count and step functions on it [6]. Therefore, coherence of about 0.6 is high.

Table 4: Motivator Coherence

Motivator	Coherence	Follow-up Coherence
All Questions	0.11	0.07
Community	0.36	0.15
Enjoyment	0.32	0.25
Hostility	0.49	0.60
Ownership	0.58	0.57
Recognition	0.24	0.17

Table 4 presents the coherence of the motivators. 'Coherence' is defined as the average Pearson correlation between all pairs of questions related to the same motivator. The motivators 'Challenge', 'Ideology', 'Importance', 'Learning', 'Payment', and 'Self-use' do not appear in Table 4 since they are based on a single question each, hence our method is not applicable to them. 'Follow-up Coherence' is the same metric as 'Coherence' computed on the follow-up survey. Note that this provides additional support, yet the support is not totally independent since the participants in the follow-up survey also participated in the original one.

The ‘All Questions’ row represents all the questions together (basically related to motivation) and has rather low coherence. The following motivators all have much higher coherence, indicating that they indeed reflect a meaningful grouping of questions related to specific concepts. The coherence of ‘Hostility’ and ‘Ownership’ is relatively high in both surveys, and close to the highest coherence we can expect. The coherence of ‘Community’, ‘Enjoyment’, and ‘Recognition’ is between 0.15 to 0.36 in both surveys.

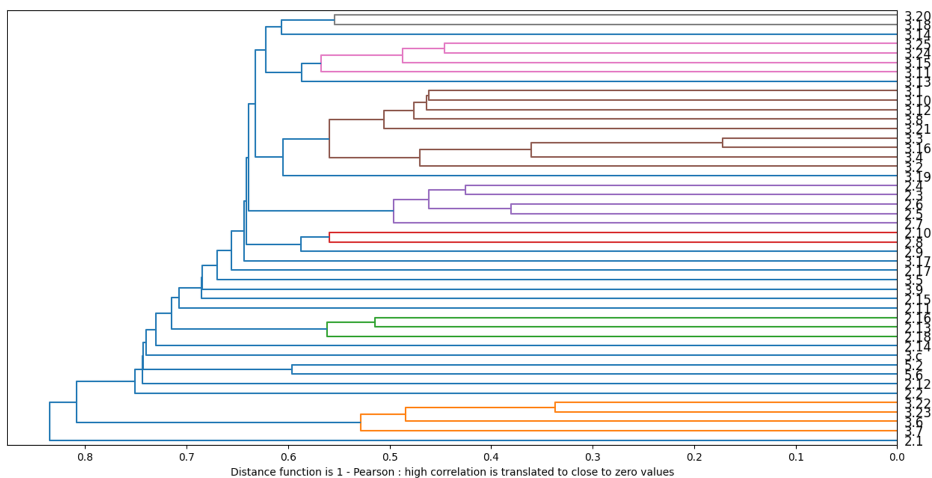


Figure 5: Dendrogram of questions based on their Pearson correlation. See questions text in [9]

The creation, selection and grouping of questions to motivators was done by the authors. In case of questions from prior work, we had indications of the intended motivators. Disagreements were discussed and resolved. Though the taxonomy is justifiable and fits our needs, we are aware of the justification of other ones, some considered by us. As a benchmark for our taxonomy we compare it to one *created using the answers, based on actual relations between them.* We build this taxonomy using automatic clustering based on their answer correlations. This will provide additional evidence on whether our grouping was indeed meaningful.

The dendrogram in Figure 5 represents a hierarchical clustering [74] of questions based on the correlations between them. The questions ‘I have significant influence on the repository’ (3.3) and ‘I am a core member of the repository’ (3.16) are the most correlated questions with Pearson of 0.83 (transformed to $1 - 0.83 = 0.17$ to represent distance in the figure). As we allow weaker correlations, more questions are clustered together, and when we allow Pearson correlation of only 0.3 most questions are already grouped into one big cluster,

which is uninteresting. Some of the clusters match our content-based motivators: The orange cluster matches hostility and the lower brown sub-cluster matches ownership. The purple cluster shows a group that we did not consider: productivity and possible productivity improving elements such as good colleagues, physical conditions, and opportunities to use your abilities. Other clusters may overlap our definitions, but also mix in unrelated questions. For example, the pink cluster contains 3 questions about recognition and one about importance, which we feel is not really related. Since our manually built factors are more coherent with respect to content, we use them and not the hierarchical clusters.

6.4 Answers Stability Between Original and Follow-up Surveys

The follow-up survey, conducted one year after the original survey, allowed us to compare the answers of the same person over time. Table 5 shows stability of questions by motivator.

To compare the answers in the two surveys we first compute the Pearson correlation between them. We also compute the differences between them, both the average absolute average difference (column 'Avg. Abs. Diff') and the average relative difference (difference divided by the question value, column 'Avg. Rel. Diff'). 'Pred(25)' [102] is the probability that the follow-up answer is in the range of 25% of the initial answer.

Note that the distributions of answers are far from uniform, and some answers are much more popular than others. As a result, there is a high probability for getting the same answer even when the answers are independent. 'Pred(25) Lift' computes the lift, i.e. the extra probability above the expected Pred(25) from two independent answers from the answers distribution.

Pearson correlation, Pred(25), Pred(25) lift, and relative difference indicate stability for almost all motivators. Hostility has a near zero lift and not a large positive one, indicating less stability than expected. Note that the hostility distribution (e.g. Figure 3) has a strong mode in the lowest value, making the independent distribution benchmark very high. Note also that the lift is close to zero hence more likely to be influenced by noise.

Payment is a binary feature hence its stability should be analyzed with different metrics. The initial and follow-up payment agree in 85% of the cases. 70% of those that were paid in the initial survey were also paid a year later. Only a single person out of the 27 that were not paid in the initial survey got payment in the follow-up.

The Pearson correlations are between 0.42 to 0.68. Though, over time the project, the people, and their motivations change [41], which might result in different answers.

Table 5: Similarity of Motivation Type Answers of Same Person in Two Dates

Motivator	Pearson	Avg. Abs. Diff	Avg. Rel. Diff	Pred(25)	Pred(25) Lift
Learning	0.68	0.91	0.04	0.81	0.22
Ownership	0.66	1.02	-0.01	0.83	0.42
Hostility	0.63	1.10	0.38	0.38	-0.02
Enjoyment	0.60	1.06	0.00	0.84	0.29
Ideology	0.57	1.61	0.02	0.74	0.99
Importance	0.54	1.28	0.02	0.74	0.38
Motivation	0.52	1.83	-0.03	0.60	0.30
Challenge	0.51	1.46	0.08	0.70	0.23
Community	0.48	1.46	0.04	0.44	0.04
Recognition	0.45	1.70	0.19	0.54	0.36
Self-use	0.43	2.35	0.04	0.51	0.20

7 Threats to Validity

Usually, construct validity raises due to the measurement method. In the case of motivation, and motivators, there is a problem since the concept themselves are not well defined [70, 95] or over-defined, with 102 definitions [56]. Therefore, it is hard to measure them or evaluate how well a measurement method performs. We cope with this threat using several methods like using questions from prior work, which were already considered to be useful.

The selection of motivators and questions has subjective aspects, and others could be chosen. We based our selection on motivators with massive prior work in motivation in general, in software development, and open source. We compared our taxonomy to an automated objective taxonomy, derived from the answers.

However, our strongest calming evidence for both construct and external validity, comes from the use of our data to validate motivation labeling functions [10]. Our data agrees with the labeling functions, which agree with the retention of 151 thousands developers working on 18,958 real projects. Hence, the answers agree with the retention of a large number of developers in their natural real work.

Investigating internal validity, some questions have systematic problems. The job satisfaction questions were answered by many participants on their day job. In self-assessment questions developers have a very high perception of themselves, not aligned with their actual performance (Section 6.2). We therefore avoided using these answers for motivation analysis.

In order to further reduce the influence of individual questions, we grouped questions by motivators. While we still do the analysis at the question level too (available in the supplementary materials), the aggregation reduces the weight of a specific answer and makes the concepts more robust. However, answers to different questions on the same concepts are only moderately correlated (Section

5.2), so one can argue that our grouping is not correct. Indeed, we grouped questions by subjective judgment of their contents, and in principle a different taxonomy could be used. We compared our content based grouping to the one inferred from correlation (Section 5.2). The match is only partial hence our grouping is supported yet there are also different justifiable groupings.

Surveys are answered by people. Answers of the same person change over time (Section 6.4) and therefore analysis based on the original survey might not agree with the same analysis on the follow-up survey. Concerning the concept of hostility, different people provided widely different answers about the same project (Section 5.2.11). In this case this does not represent a data-quality problem, since the answers actually represent different experiences, and the difference itself is an important result.

We measured the relations between motivators and motivation in multiple ways: correlation, predictive performance, and co-change. A similar result in all methods (e.g., community increases motivation by 20%) would have been very reliable. However, there are many quantitative and even several qualitative differences in the results. For example, Table 1 shows that all motivators have positive precision lift in high motivation prediction, hence knowing of a positive motivator increases the probability of high motivation. On the other hand, in the follow-up analysis presented in Table 2, three of the eleven motivators have negative precision lift, hence knowing of their increase from the original survey indicates higher probability of motivation reduction. While negative lift is expected for hostility, the results for challenge and ideology disagree with the high motivation prediction.

Reliability was evaluated over time, using the follow-up survey, and with respect to other questions. We also investigated the validity of questions using simple mistakes, biases, and comparison to actual behavior.

Throughout this research we obtained many results. While our number of participants is very high for a survey, we analyzed the answers in many ways. In some scenarios (e.g., developers in the same project, developers answering the follow-up), the numbers are quite small. Statistical learning theory [97, 96] tells us that in such cases several of the empirical results will probably be different from the actual ones. This is an inherited threat from the dataset size and analysis type, which should be resolved by replication studies obtaining more data and supporting the results in different analyses.

Our survey population came from two sources - GitHub developers accessed by email and developers accessed using social media. We used machine learning (details in the supplementary materials) to see how different these populations are and could not find a model differing them better than the positive rate. This means that there is no obvious big difference between the populations.

Similarity between the participants group and the desired population increases the probability of generalization. We provide demographics analysis and a comparison to the Stack Overflow survey (in the supplementary materials). Though our participants resemble the Stack Overflow survey participants (while being somewhat more professional), it is not clear what the general developers group is.

8 Conclusions

We conducted a large survey of software developers regarding motivation. We grouped the questions by motivators and analyzed their relation to motivation.

Our supervised-learning-based analysis of motivators ranged from using each motivator as a classifier for motivation, through using motivator improvement as a classifier for motivation improvement, to constructing models that use the combined power of multiple motivators for improved prediction. We confirmed that previously suggested motivators do indeed contribute to motivation. At the same time, the influence of each individual motivator is limited, as also noted in prior work [48, 30].

Apparently, the motivation of different developers, working in different contexts, may be influenced by different motivators. No single motivator by itself is sufficient for inducing high motivation. At the same time, none of the motivators is strictly necessary. An analysis of the relations between them indicated that motivators tend to have low correlation. This indicates that one should not look at motivators from the prism of which is the "most important" one; a better description is that each one of them captures a different aspect of motivation [70], and multiple aspects should be satisfied in order to have high motivation.

All motivators have coherence higher than the set of all questions together, but only hostility and ownership have rather high coherence. In general, all motivators are at least moderately coherent and predictive, in all analyses. However, out of the eleven motivators, ten motivators (excluding ownership) did not meet all three criteria of high coherence, stability, and predictive power. This indicates that the bar that we set is high.

It is also interesting to notice the relative position of payment. Trying to predict high motivation based on a single motivator, payment has precision lift of 10%, the lowest value of all positive motivators, and the only negative lift on the follow-up survey. Since payment is the common way to promote motivation in businesses, it is important to note that other motivators might lead to a larger effect.

Hostility is a very coherent demotivator. However, different people in the same project disagree on hostility, implying that it is not noticed by others. Hence, not noticing hostility is not enough to assure lack of hostility in a project, and therefore actively looking for it might be needed.

Our survey also provided another motivation to investigate motivation. 73% of the participants answered that motivation has more influence on their productivity (answers higher than neutral), and only 9% answered that their skill is more influential. This agrees with prior work on the benefit of motivation for productivity [91].

Participants who reported an improvement in the interest expressed in them had a large tendency for improvement in motivation. Recognition, and specifically expressing interest, is free, applicable in all situations, and influential. Considerate behavior and looking for practical benefits coincide here. Be kind and give recognition, it is likely to pay off.

Data Availability

All experimental materials (except for identifying data such as emails and GitHub profiles) is available at [9].

Acknowledgments

First, we would like to thank our participants. Other than just answering the survey, they alerted on problems and suggested many ideas. Many of them left their email to receive the research results which is heartwarming. We could not conduct this research without you. You are our partners. Thank you!

We thank the IRB committee for their feedback and approval of the study (09032020). This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

We thank David Amit for the support, interest, and many wise ideas and questions. We thank Avraham Kluger who introduced to us the importance of listening and helped in many ways, and to Dan Ariely who guided us into the domain of motivation. We also thank Tali Kleiman, Itamar Gati, Zafrir Buchler, Doody Parizada, Eyal Zaidman, Yaniv Mama, Asaf Korem, and Nili Ben Ezra for the discussions, insights, and help.

References

[1] Github user search showing 37,446,292 available users. URL `https://github.com/search?q=type:user&type=Users`. Accessed: 2019-05-23.

[2] A. S. Acharya, A. Prakash, P. Saxena, and A. Nigam. Sampling: Why and how of it. *Indian Journal of Medical Specialties*, 4(2):330–333, 2013. doi:10.7713/ijms.2013.0032.

[3] J. S. Adams. Toward an understanding of inequity. *Journal of Abnormal Psychology*, 67(5):422–436, 1963. doi:10.1037/h0040968.

[4] T. M. Amabile, K. G. Hill, B. A. Hennessey, and E. M. Tighe. The work preference inventory: Assessing intrinsic and extrinsic motivational orientations. *Journal of Personality and Social Psychology*, 66(5):950–967, 1994. doi:10.1037/0022-3514.66.5.950.

[5] I. Amit. End to end software engineering research. 2021, arXiv:2112.11858 [cs.SE].

[6] I. Amit, N. B. Ezra, and D. G. Feitelson. Follow your nose – which code smells are worth chasing? 2021, arXiv:2103.01861 [cs.SE].

[7] I. Amit and D. G. Feitelson. Which refactoring reduces bug rate? In *Proceedings of the 15th International Conference on Predictive Models*

and Data Analytics in Software Engineering, PROMISE'19, pages 12–15, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3345629.3345631.

[8] I. Amit and D. G. Feitelson. Corrective commit probability: A measure of the effort invested in bug fixing. *Software Quality Journal*, 29(4):817–861, Aug 2021. doi:10.1007/s11219-021-09564-z.

[9] I. Amit and D. G. Feitelson. Replication package https://github.com/evidencebp/motivation-survey, Dec 2023.

[10] I. Amit and D. G. Feitelson. Motivation research using labeling functions. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 222–231, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661167.3661224.

[11] I. Amit, E. Firstenberg, and Y. Meshi. Framework for semi-supervised learning when no labeled data is given. U.S. patent #US11468358B2, 2017. URL `https://patents.google.com/patent/US11468358B2`.

[12] 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*, pages 94–105. Universitaire Pers, Rotterdam, The Netherlands, 1989.

[13] D. Ariely, E. Kamenica, and D. Prelec. Man's search for meaning: The case of legos. *Journal of Economic Behavior & Organization*, 67(3-4):671–677, 2008. doi:10.1016/j.jebo.2008.01.004.

[14] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, and S. Lacoste-Julien. A closer look at memorization in deep networks. In *34th International Conference on Machine Learning*, pages 233–242. PMLR, 2017.

[15] G. Asproni. Motivation, teamwork, and agile development. *Agile Times*, 4, Mar 2004.

[16] N. Baddoo and T. Hall. Motivators of software process improvement: an analysis of practitioners' views. *Journal of Systems and Software*, 62(2):85–96, 2002. doi:10.1016/S0164-1212(01)00125-X.

[17] S. Baltes and P. Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022.

[18] K. M. Bartol and D. C. Martin. Managing information systems personnel: A review of the literature and managerial implications. *MIS Quarterly*, 6(4):49–70, Dec 1982. URL `http://dl.acm.org/citation.cfm?id=2025521.2025525`.

[19] N. Bassett-Jones and G. C. Lloyd. Does herzberg's motivation theory have staying power? *Journal of Management Development*, 24:929–943, Dec 2005. doi:10.1108/02621710510627064.

[20] R. W. Beatty, C. E. Schneier, and J. R. Beatty. An empirical investigation of perceptions of ratee behavior frequency and ratee behavior change using behavioral expectation scales (BES). *Personnel Psychology*, 30(4):647–658, 1977. doi:10.1111/j.1744-6570.1977.tb02333.x.

[21] D. G. Beckers, D. van der Linden, P. G. Smulders, M. A. Kompier, M. J. van Veldhoven, and N. W. van Yperen. Working overtime hours: relations with fatigue, work motivation, and the quality of work. *Journal of Occupational and Environmental Medicine*, pages 1282–1289, 2004.

[22] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp. Motivation in software engineering: A systematic literature review. *Information & Software Technology*, 50(9-10):860–878, 2008. doi:10.1016/j.infsof.2007.09.004.

[23] J. K. Beggan. On the social nature of nonsocial perception: The mere ownership effect. *Journal of Personality and Social Psychology*, 62(2):229, 1992. doi:10.1037/0022-3514.62.2.229.

[24] S. Belenzon and M. Schankerman. Motivation and Sorting in Open Source Software Innovation. CEP Discussion Papers dp0893, Centre for Economic Performance, LSE, Oct 2008. URL `https://ideas.repec.org/p/cep/cepdps/dp0893.html`.

[25] R. Bénabou and J. Tirole. Intrinsic and extrinsic motivation. *The Review of Economic Studies*, 70(3):489–520, 2003.

[26] 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. doi:10.1145/279943.279962.

[27] D. J. Campbell and C. Lee. Self-appraisal in performance evaluation: Development versus evaluation. *The Academy of Management Review*, 13(2):302–314, 1988. doi:10.2307/258579.

[28] 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.

[29] L. Cortina, V. Magley, J. Williams, and R. Langhout. Incivility in the workplace: Incidence and impact. *Journal of Occupational Health Psychology*, 6:64–80, Feb 2001. doi:10.1037/1076-8998.6.1.64.

[30] J. D. Couger. Motivators vs. demotivators in the is environment. *Journal of Systems Management*, 39(6):36, 1988.

[31] A. B. Cristina and C. Rossi. Altruistic individuals, selfish firms? the structure of motivation in open source software. *First Monday*, 9:9, 2004. doi:10.5210/fm.v9i1.1113.

[32] T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams*. Dorset House, 1987.

[33] I. Etikan, S. A. Musa, R. S. Alkassim, et al. Comparison of convenience sampling and purposive sampling. *American Journal of Theoretical and Applied Statistics*, 5(1):1–4, 2016. doi:10.11648/j.ajtas.20160501.11.

[34] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *20th IEEE International Conference on Program Comprehension*, pages 73–82, 2012. doi:10.1109/ICPC.2012.6240511.

[35] D. G. Feitelson. "we do not appreciate being experimented on": Developer and researcher views on the ethics of experiments on open-source projects. arXiv 2112.13217 [cs.SE], 2021, arXiv:2112.13217 [cs.SE].

[36] F. Ferreira, L. L. Silva, and M. T. Valente. Turnover in open-source projects: The case of core developers. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 447–456, 2020. doi:10.1145/3422392.3422433.

[37] J. Fitz-Enz. Who is the dp professional. *Datamation*, 24(9):125–128, 1978.

[38] S. A. Frangos. Motivated humans for reliable software products. In D. Gritzalis, editor, *Reliability, Quality and Safety of Software-Intensive Systems: IFIP TC5 WG5.4 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, pages 83–91. Springer US, Boston, MA, May 1997. doi:10.1007/978-0-387-35097-4_7.

[39] C. França, F. Q. B. da Silva, and H. Sharp. Motivation and satisfaction of software engineers. *IEEE Transactions on Software Engineering*, 46(2):118–140, Feb 2020. doi:10.1109/TSE.2018.2842201.

[40] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. doi:10.1006/jcss.1997.1504.

[41] M. Gerosa, I. Wiese, B. Trinkenreich, G. Link, G. Robles, C. Treude, I. Steinmacher, and A. Sarma. The shifting sands of motivation: Revisiting what drives contributors in open source. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 1046–1058, 2021. doi:10.1109/ICSE43902.2021.00098.

[42] 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. doi:10.14569/IJACSA.2018.090712.

[43] A. Grant. The significance of task significance: Job performance effects, relational mechanisms, and boundary conditions. *The Journal of Applied Psychology*, 93:108–24, Feb 2008. doi:10.1037/0021-9010.93.1.108.

[44] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson. What happens when software developers are (un)happy. *Journal of Systems and Software*, 140:32–47, 2018. doi:10.1016/j.jss.2018.02.041.

[45] J. R. Hackman and G. R. Oldman. *Motivation Through the Design of Work: Test of a Theory*. Academic Press, New York, 1976.

[46] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, et al. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, 27(6):1–49, 2022. doi:10.1007/s10664-021-10083-5.

[47] G. Hertel, S. Niedner, and S. Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research Policy*, 32(7):1159–1177, 2003. doi:10.1016/S0048-7333(03)00047-7. Open Source Software Development.

[48] F. Herzberg. One more time: How do you motivate employees? *New York: The Leader Manager*, pages 433–448, 1986.

[49] F. Herzberg, B. Mausner, and B. B. Snyderman. *Motivation to Work*. Wiley, New York, 1959.

[50] D. Hills, C. Joyce, and J. Humphreys. Validation of a job satisfaction scale in the australian clinical medical workforce. *Evaluation & the Health Professions*, 35:47–76, Mar 2011. doi:10.1177/0163278710397339.

[51] Y. Huang, D. Ford, and T. Zimmermann. Leaving my fingerprints: Motivations and challenges of contributing to oss for social good.

[52] J. Irvine. A framework for comparing theories related to motivation in education. *Research in Higher Education Journal*, 35, 2018.

[53] A. Joshi, S. Kale, S. Chandel, and D. K. Pal. Likert scale: Explored and explained. *British Journal of Applied Science & Technology*, 7(4):396, 2015. doi:10.9734/BJAST/2015/14975.

[54] T. Judge, C. Thoresen, J. Bono, and G. Patton. The job satisfaction-job performance relationship: a qualitative and quantitative review. *Psychological Bulletin*, 127:376–407, Jan 2001. doi:10.1037/0033-2909.127.3.376.

[55] T. A. Judge, E. A. Locke, C. C. Durham, and A. N. Kluger. Dispositional effects on job and life satisfaction: The role of core evaluations. *Journal of Applied Psychology*, 83:17–34, 1998. doi:10.1037/0021-9010.83.1.17.

[56] P. R. Kleinginna Jr and A. M. Kleinginna. A categorized list of motivation definitions, with a suggestion for a consensual definition. *Motivation and emotion*, 5(3):263–291, 1981.

[57] A. Kluger and O. Bouskila-Yam. Facilitating listening scale (FLS). In D. L. Worthington and G. D. Bodie, editors, *The Sourcebook of Listening Research: Methodology and Measures*, pages 272–280. Wiley Online Library, Aug 2017. doi:10.1002/9781119102991.ch25.

[58] J. Kruger and D. Dunning. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77:1121–1134, 1999. doi:10.1037/0022-3514.77.6.1121.

[59] K. Kuusinen, H. Petrie, F. Fagerholm, and T. Mikkonen. Flow, intrinsic motivation, and developer experience in software engineering. In H. Sharp and T. Hall, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 104–117. Springer International Publishing, 2016. doi:10.1007/978-3-319-33515-5_9.

[60] C. F. Lam and S. T. Gurland. Self-determined work motivation predicts job outcomes, but what predicts self-determined work motivation? *Journal of Research in Personality*, 42(4):1109–1115, 2008. doi:10.1016/j.jrp.2008.02.002.

[61] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi:10.1038/nature14539.

[62] P. Lenberg, R. Feldt, and L. G. Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, 2015. doi:10.1016/j.jss.2015.04.084.

[63] J. Lerner and J. Tirole. Some simple economics of open-source. *Journal of Industrial Economics*, 50:197–234, Feb 2002. URL `http://www.people.hbs.edu/jlerner/simple.pdf`.

[64] S.-O. Leung. A comparison of psychometric properties and normality in 4-, 5-, 6-, and 11-point likert scales. *Journal of social service research*, 37(4):412–421, 2011.

[65] Y. Li, C.-H. Tan, H. Teo, and T. Mattar. Motivating open source software developers: Influence of transformational and transactional leaderships. In *Proceedings of the 2006 ACM SIGMIS CPR Conference on Computer Personnel Research: Forty Four Years of Computer Personnel Research: Achievements, Challenges & the Future*, pages 34–43, Jan 2006. doi:10.1145/1125170.1125182.

[66] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22:55, 1932.

[67] E. A. Locke. Toward a theory of task motivation and incentives. *Organizational Behavior and Human Performance*, 3(2):157–189, 1968. doi:10.1016/0030-5073(68)90004-4.

[68] C. Maslach, S. Jackson, and M. Leiter. Maslach burnout inventory third edition. In C. P. Zalaquett and R. J. Wood, editors, *Evaluating Stress: A Book of Resources*, pages 191–218. Scarecrow Education, 1997.

[69] A. H. Maslow. A theory of human motivation. *Psychological Review*, 50:370–396, 1943.

[70] J. D. Mayer, M. A. Faber, and X. Xu. Seventy-five years of motivation measures (1930–2005): A descriptive analysis. *Motivation and Emotion*, 31:83–103, 2007.

[71] D. C. McClelland. *The Achieving Society.* Van Nostrand, Princeton, NJ, 1961.

[72] A. N. Meyer, E. T. Barr, C. Bird, and T. Zimmermann. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering*, 47(5):863–880, 2021. doi:10.1109/TSE.2019.2904957.

[73] E. Murphy-Hill, C. Jaspan, C. Sadowski, D. Shepherd, M. Phillips, C. Winter, A. Knight, E. Smith, and M. Jorde. What predicts software developers' productivity? *IEEE Transactions on Software Engineering*, 47(3):582–594, 2021. doi:10.1109/TSE.2019.2900308.

[74] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview. *WIREs Data Mining and Knowledge Discovery*, 2(1):86–97, 2012. doi:10.1002/widm.53.

[75] 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. URL `http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf`.

[76] P. M. Podsakoff, S. B. MacKenzie, J.-Y. Lee, and N. P. Podsakoff. Common method biases in behavioral research: a critical review of the literature and recommended remedies. *Journal of Applied Psychology*, 88(5):879, 2003. doi:10.1037/0021-9010.88.5.879.

[77] J. R. Quinlan. *C4. 5: programs for machine learning.* Elsevier, 2014.

[78] S. Ramachandran and S. V. Rao. An effort towards identifying occupational culture among information systems professionals. In *Proceedings of the 2006 ACM SIGMIS CPR Conference on Computer Personnel Research: Forty Four Years of Computer Personnel Research: Achievements, Challenges & the Future*, pages 198–204, 2006. doi:10.1145/1125170.1125221.

[79] N. Raman, M. Cao, Y. Tsvetkov, C. Kästner, and B. Vasilescu. Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 57–60, 2020. doi:10.1145/3377816.3381732.

[80] 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. URL http://papers.nips.cc/paper/6523-data-programming-creating-large-training-sets-quickly.pdf.

[81] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[82] D. Riehle. The economic motivation of open source software: Stakeholder perspectives. *Computer*, 40(4):25–32, 2007. doi:10.1109/MC.2007.147.

[83] J. A. Roberts, I.-H. Hann, and S. A. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Manage. Sci.*, 52(7):984–999, Jul 2006. doi:10.1287/mnsc.1060.0554.

[84] A. S. Tsui. A role set analysis of managerial reputation. *Academy of Management Proceedings*, 1982:265–269, Aug 1982. doi:10.5465/AMBPP.1982.4976642.

[85] J. Scott, E. O'Rourke, and A. Grant. Employee surveys are still one of the best ways to measure engagement. *Harvard Business Review*, March 2018. URL https://hbr.org/2018/03/employee-surveys-are-still-one-of-the-best-ways-to-measure-engagement.

[86] S. Scullen, M. Mount, and M. Goff. Understanding the latent structure of job performance ratings. *The Journal of Applied Psychology*, 85:956–70, Jan 2001. doi:10.1037//0021-9010.85.6.956.

[87] H. Shamon and C. Berning. Attention check items and instructions in online surveys with incentivized and non-incentivized samples: Boon or bane for data quality? *Shamon, H., & Berning, CC (2020). Attention Check Items and Instructions in Online Surveys with Incentivized and Non-Incentivized Samples: Boon or Bane for Data Quality*, pages 55–77, 2019.

[88] H. Sharp, N. Baddoo, S. Beecham, T. Hall, and H. Robinson. Models of motivation in software engineering. *Information & Software Technology*, 51(1):219–233, 2009. doi:10.1016/j.infsof.2008.05.009.

[89] H. Sharp and T. Hall. An initial investigation of software practitioners' motivation. In *Proceedings of the ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pages 84–91, 2009. doi:10.1109/CHASE.2009.5071418.

[90] H. Shavit and R. Shouval. Self-esteem and cognitive consistency effects on self-other evaluation. *Journal of Experimental Social Psychology*, 16(5):417–425, 1980. doi:10.1016/0022-1031(80)90048-7.

[91] J. A. Shepperd. Productivity loss in performance groups: A motivation analysis. *Psychological Bulletin*, 113(1):67, 1993. doi:10.1037/0033-2909.113.1.67.

[92] B. F. Skinner. *The Behavior of Organisms: An Experimental Analysis*. New York: Appleton-Century-Crofts, 1938.

[93] R. Stallman. Why software should be free. 2007. URL `https://www.gnu.org/philosophy/shouldbefree.en.html`.

[94] F. R. Tanner. On motivating engineers. In *IEMC '03 Proceedings. Managing Technologically Driven Organizations: The Human Side of Innovation and Change*, pages 214–218, Nov 2003. doi:10.1109/IEMC.2003.1252263.

[95] K. Toode, P. Routasalo, and T. Suominen. Work motivation of nurses: A literature review. *International Journal of Nursing Studies*, 48(2):246–257, 2011. doi:https://doi.org/10.1016/j.ijnurstu.2010.09.013.

[96] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer science & business media, 2013.

[97] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971. doi:10.1137/1116025.

[98] P. F. Verhulst. Resherches mathematiques sur la loi d'accroissement de la population. *Nouveaux Memoires de l'Academie Royale des Sciences*, 18:1–41, 1845.

[99] C. V. Viswesvaran, D. Ones, and F. Schmidt. Comparative analysis of reliability of job performance ratings. *Journal of Applied Psychology*, 81:557–574, Oct 1996. doi:10.1037/0021-9010.81.5.557.

[100] G. von Krogh, S. Haefliger, S. Spaeth, and M. W. Wallin. Carrots and rainbows: Motivation and social practice in open source software development. *MIS Quarterly*, 36(2):649–676, 2012. URL `http://www.jstor.org/stable/41703471`.

[101] V. H. Vroom. *Work and Motivation*. Wiley, New York, 1964.

[102] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology*, 54(1):41–59, 2012. doi:10.1016/j.infsof.2011.09.002.

[103] B. Wigert and J. Harter. Re-engineering performance management. *Gallup.com*, 2019. URL http://news.gallup.com/reports/208811/re-engineering-performance-management.aspx.

[104] 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. doi:10.1037/1076-8998.5.1.84.

[105] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *Proceedings of the 25th International Conference on Software Engineering*, pages 419–429, 2003. doi:10.1109/ICSE.2003.1201220.

[106] B. Zadrozny and C. Elkan. Transforming classifier scores into accurate multiclass probability estimates. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 694–699, 2002. doi:10.1145/775047.775151.

A Survey Questions

To facilitate the review, the full questionnaire used in the survey is reproduced herewith.

A.1 Survey introduction

Dear participant,

We are a team of researchers interested in improving software development (see for example https://www.cse.huji.ac.il/~feit/papers/Refactor19PROMISE.pdf).

If you contributed to a GitHub repository as a developer in the last 12 months, we ask for your help by answering questions about your contribution and motivation. Answering these questions is estimated to take 10–15 minutes of your time.

Based on the experience of respondents to this questionnaire in the past, you may gain new insights about your priorities in software development and areas of importance to you. Your answers, with the answers of others, will allow researchers in the future to investigate motivation, quality and productivity in software development and hopefully improve them.

We would appreciate a link to your GitHub profile in order to match your answers and GitHub activity (e.g., number of commits, years in the repository). We are aware that the profile is a personal identifier and we will keep it private and use it for research purposes only. The results of analysis of the profile data

will be reported in aggregated form only. Of course, in case that you are not interested, you can leave the field empty.

If you are willing to participate in this study, move to the next page. By moving to the next page you agree to participate in this study. The only inconvenience that this study may cause you is the need to concentrate on the questions for about 10-15 minutes. Yet, you may quit this survey at any time without answering all the questions, with no consequences for you. We will be grateful if you complete ALL the questions. No personally identifying information will be collected, except your GitHub profile if you choose to share it.

Thank you so much for your help.

Prof. Dror Feitelson, Prof. Avi Kluger, Ph.D. candidate Idan Amit

If you have any question you can contact Idan Amit at idan.amit@mail.huji.ac.il

A.2 Questions regarding yourself

The questions in this section are in Likert scale where 1 is 'Strongly disagree' and 11 is 'Strongly agree'.

1. Productivity is more important to me than quality
2. My motivation has more influence on my productivity, than my skill
3. I regularly reach a high level of productivity (based on [73])
4. I am a relatively productive programmer
5. I am skilled in software development (based on [59])
6. My code is of high quality
7. I am satisfied with my performance in software development [59]
8. I want my code to be beautiful
9. I enjoy software development very much
10. It is important for me to program well (based on [59])
11. I write tests for my code
12. I write detailed commit messages
13. I contribute to open source in order to have an online portfolio
14. I try to write high quality code because others will see it
15. I enjoy trying to solve complex problems [4]
16. I contribute to open source in order to become a better programmer
17. I improved as a programmer since a year ago
18. I contribute to open source due to ideology

A.3 Questions regarding activity in a repository

Please choose *one specific* GitHub repository that you work on. Answer the following questions with respect to this repository. (These questions

- What is the link of the GitHub repository that you answer on? (Free ext)
- How many hours a week do you work on the repository (average)? (Free text)
- I'm being paid for my work in this repository (Yes/No)

The questions in this section are in Likert scale where 1 is 'Strongly disagree' and 11 is 'Strongly agree'.

1. I regularly have a high level of motivation to contribute to the repository (based on [73])
2. I have complete autonomy in contributing to the repository
3. I have significant influence on the repository
4. I feel responsible for the repository's success
5. I'm interested in the repository for my own needs
6. We have many heated arguments in the community. If you are the only developer in the project, please skip.
7. I wish that certain developers in the project will leave. If you are the only developer in the project, please skip.
8. My work on the repository is creative
9. Working on this repository is challenging
10. I derive satisfaction from working on this repository
11. The repository is important
12. When I look at what we accomplish, I feel a sense of pride.
13. Belonging to the community is motivating my work on the project. If you are the only developer in the project, please skip.
14. The community is very professional. If you are the only developer in the project, please skip.
15. I get recognition due to my contribution to the repository
16. I am a core member of the repository

17. I learn from my contributions
18. The quality of the code in this repository is better than others
19. Code quality in the repository improved since a year ago
20. The project's community of developers is more motivated than that of other projects. If you are the only developer in the project, please skip.
21. My personal motivation in this repository has increased since a year ago
22. In the past year, members of my project community put me down or were condescending to me. If you are the only developer in the project, please skip. (based on [29])
23. In the past year, members of my GitHub community made demeaning or derogatory remarks about me. If you are the only developer in the project, please skip. (based on [29])
24. In the past year, members of my project community asked questions that show their understanding of my contributions. If you are the only developer in the project, please skip. (based on [57])
25. In the past year, members of my project community expressed interest in my contributions. If you are the only developer in the project, please skip. (based on [57])

A.4 Job Satisfaction

The following questions are from Job Satisfaction Scale questionnaire [50]. We present the questionnaire as is in order to compare to previous results. In case that you find some questions irrelevant, please skip them.

The questions in this section are in Likert scale where 1 is 'Extremely dissatisfied' and 7 is 'Extremely satisfied', as in the original survey [50].

The questions indicate level of satisfaction with the following:

1. Freedom to choose your own method of working
2. Amount of variety in your work
3. Physical working conditions
4. Opportunities to use your abilities
5. Your colleagues and fellow workers
6. Recognition you get for good work
7. Your hours of work
8. Your remuneration (payment)
9. Amount of responsibility you are given
10. Taking everything into consideration, how do you feel about your work?

A.5 Demography

1. Country (Free text)
2. Age (0-100 selection)
3. Gender (Free text)
4. I work as a professional programmer (Yes/No)
5. Years of work experience (not including studies) (Free text)
6. Years of contribution to GitHub(0-15 selection)
7. Academic background (degree and graduation year) (Free text)
8. Git profile link (Free text) We would appreciate a link to your GitHub profile in order to match your answers and GitHub activity (e.g., number of commits, years in the repository). We are aware that the profile is a personal identifier and we will keep it private and use it for research purposes only. The results of analysis of the profile data will be reported in aggregated form only. Of course, in case that you are not interested, you can leave the field empty.

A.6 Open questions

1. Do you have any comments on the questionnaire or research? Are you motivated due to a cause that we didn't consider? Do you have a method that increases your code quality? (Free text)
2. Thank you for answering our survey. If you would like to be informed in the results of the research or to participate in the gift card lottery, please enter your email and we will send it to you once completed. The email will not be used for profile identification. (Free text)

4.4 Motivation Research Using Labeling Functions

Published: Amit, Idan, and Dror G. Feitelson. "Motivation Research Using Labeling Functions." Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. 2024.

In this work we used labeling functions [62, 6] in order to investigate motivation and its relation to behavior in real world activity. This adds a new method to the toolbox of abstract concept investigation, common in social sciences. It allows large-scale investigation over long periods, identification of rare patterns, quantification, and reproducibility.

We used the answers to the motivation survey to show that our labeling functions are weak classifiers for motivation. We further validated them at large scale on the GitHub data using co-change, twins, and more, to show that they measure the same concept.

Applying this showed the impact of motivation of people in their natural behavior on a massive scale, with data about more than 150 thousand developers, over years of activity. Here are some examples of the results. The activity period in a project is longer, by up to 70%, given higher motivation as identified by any of the functions. Commit duration is also higher, possibly indicating higher attention. But despite the extra time investment in a single commit, the number of commits might be 4 times higher.

This completes the path of measuring software development, identifying relations between concepts, and estimating the impact on performance.

Motivation Research Using Labeling Functions

Idan Amit
idan.amit@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Dror G. Feitelson
feit@cs.huji.ac.il
The Hebrew University
Jerusalem, Israel

ABSTRACT

Motivation is an important factor in software development. However, it is a subjective concept that is hard to quantify and study empirically. In order to use the wealth of data available about real software development projects in GitHub, we represent the motivation of developers using labeling functions. These are validated heuristics that need only be better than a guess, computable on a dataset. We define four labeling functions for motivation based on behavioral cues like working in diverse hours of the day. We validated the functions by agreement with respect to a developers survey, per person behavior, and temporal changes. We then apply them to 150 thousand developers working on GitHub projects. Using the identification of motivated developers, we measure developer performance gaps. We show that motivated developers have up to 70% longer activity period, produce up to 300% more commits, and invest up to 44% more time per commit.

CCS CONCEPTS

• **Applied computing → Psychology**; • **Computing methodologies → Machine learning**.

KEYWORDS

methodology, weak supervision, software engineering, motivation

ACM Reference Format:
Idan Amit and Dror G. Feitelson. 2024. Motivation Research Using Labeling Functions. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661224

1 INTRODUCTION

Human aspects of software engineering are usually studied using tools like experiments, case studies, interviews, and surveys. These can be costly in money and effort to apply, limiting the data to only a small number of samples. Machine learning on large datasets can complement the research done using such methods and leverage the data available in open-source code repositories. However, to investigate a concept using such datasets one needs to identify it. This is difficult to do when the concept is abstract and subjective, such as motivation.

Machine learning copes with this problem by using labeled samples instead of a precise definition. A model that can predict the labels of unseen samples demonstrates that it has captured the concept that they represent. In supervised learning, one builds the model using labeled train samples. But we do not have enough labeled samples to build a robust model.

The solution we propose is to use labeling functions as models. Labeling functions are heuristics for generating labels, correlated with motivation, and validated to predict it better than a guess [50]. In our case the labeling functions use developers' behaviors to predict whether they are motivated. Even a single labeling function is helpful, but it might be biased. By using multiple labeling functions that capture diverse perspectives of motivation, we increase our confidence in results that reproduce for all of them.

Our labeling functions include refactoring (investing in design improvement reflects motivation) and working diverse hours (a developer that occasionally works in late hours is assumed to be motivated). Using them we quantify the behavior of developers with and without motivation, reproduce prior work regarding its benefits [13, 33, 38], and predict future developer retention.

Our main contributions are the following:

- We provide a new methodology, complementing surveys and experiments, to investigate motivation.
- The method enables large scale, long term, quantitative, and reproducible investigation of motivation in actual behavior in a natural setting.
- We provide and validate four labeling functions for motivation, two new and two from prior work.
- We show that our labeling functions can be used to predict churn in advance, allowing intervention.
- We show that motivation is correlated with more activity, more output, and investing more time in each task.

2 THE VISION: LABELING FUNCTIONS AS A RESEARCH FRAMEWORK

The current toolbox of motivation researchers contains experiments [13], interviews, surveys [44], and case studies [52]. While these methods allow control, the cost of each sample is high, and therefore there is a validity threat due to the small datasets [58]. This small dataset problem can be solved by mining software repositories, leveraging the millions of activities performed by many thousands of developers.

However, to apply supervised learning and investigate a concept (such as motivation), we need to label the data and distinguish cases where the concept applies from those where it does not. Usually, one can obtain labels for the concept using manual human work (e.g., asking all the developers about their motivation). Manual labeling is limited in capacity, preventing leveraging the power of big data.

This work is licensed under a Creative Commons Attribution International 4.0 License.

EASE 2024, June 18–21, 2024, Salerno, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661224

Labeling functions, providing predictions that need be only slightly better than a guess [50], provide an alternative to manually labeling the concept. The use of multiple labeling functions allows us to raise our confidence in the labeling when all or most of them agree. Given a good enough representation of the concept, one can investigate it at scale. This can provide supporting evidence for relations of interest (e.g., activity period is longer given all motivation functions).

Labeling functions are especially beneficial in concepts that are not well defined. **The essence of the difficulty** is motivation being an internal, subjective, hard to measure concept. Literature surveys — including one spanning 75 years of motivation research — found that there is no canonical method to measure motivation, and the suggested methods have limited agreement [44, 54].

Hence, we do not aim to provide a new definition of motivation or measure our work with respect to an existing one. Instead, we evaluate our functions with respect to answers about motivation, specifically self-reporting on motivation and working hours [16].

2.1 Labeling Functions

A labeling function [19] is a validated computable weak classifier, a heuristic that one can apply computationally to a dataset and get predictions that are better than a guess.

The concept of weak learnability [37], learning slightly better than a guess, was suggested as a way to relax the high requirements of PAC learning [56]. Surprisingly, it was shown that the concepts are equivalent, and one can boost weak learners to regular PAC learners [50]. Since then boosting became an important learning method [29, 50]. Weak classifiers were also found powerful in coping with the lack of labeled data. They were part of both theoretical and practical work like co-training [19] and weakly-supervised learning [9, 11, 23].

Labeling functions can be either learned from a dataset or just a fixed rule. An example of such a rule is “people that participate in popular projects tend to be motivated by recognition”. This rule is not perfectly accurate, yet it encapsulates knowledge which improves our prediction. Given a single labeling function, one can use it as a proxy for the concept. For example, the labeling function of retention in a project can be used to investigate the concept of motivation. Of course, the same investigation can also be framed as the investigation of retention as an object of interest on its own.

Behavioral cues [25] and even labeling functions have been used previously in motivation research, though not formally. For example, coming to work in a snowstorm is predictive of high satisfaction [52]. In open-source the projects’ license openness level is predictive of the developer ideology-based motivation [18].

Our goal in this paper is not to reach the best predictive power, but to find relations between motivation and performance. For this it is beneficial to have different and diverse functions capturing different aspects of motivation. An increase of a behavior of interest, given an increase in a motivation labeling function, is a hint of the relation between them. If the result reproduces for several different motivation labeling functions, it increases the likelihood that motivation indeed increases this behavior.

3 MOTIVATION LABELING FUNCTIONS AND THE INTUITION BEHIND THEM

We looked for motivation labeling functions that represent activities which are not mandatory and require some cost. We looked for actions for which usually there is no external enforcement. For example, writing tests requires extra investment, but in many projects they are enforced by cultural norms or technological means. Therefore, tests may not be a good labeling function for motivation. Yet, the length of the commit message documenting development can be a good labeling function, since colleagues and managers usually are not aware of the message length and therefore do not enforce it.

We also wanted the functions to fit open-source development, where many of the participants volunteer as a part-time hobby. Thus, counting working days could be a labeling function for full time employees, but it might underestimate the motivation of a volunteer working on weekends.

We deliberately choose simple, one-variable binary functions. We need a binary function to distinguish motivated from unmotivated developers. Hence, we need to choose a cut-off value for the continuous labeling functions. For simplicity, we uniformly use the mean of each metric. In Section 6.1 we investigate the benefit of using the raw metrics without a cut-off and show that there is no major difference.

Last, we wanted the functions to be diverse and capture motivation in different ways. By doing so we increase the robustness of estimation and validity of relations. The labeling functions we selected are retention in a project, working diverse hours, performing refactoring, and writing detailed commit messages. The first two are based on prior work and the last two are new.

3.1 Retention

Out of a project’s developers in a given year, we define the retained ones as those that continue in the next year. Given the prior work supporting the relation between motivation and retention [12, 42, 49], we label retained developers to be motivated in the current year and developers who did not continue as not motivated.

Argyle found modest correlation between job satisfaction and retention, tending to be stronger among white collar workers. [12]

Note, though, that ending activity in a project due to reasons not related to motivation poses a threat. Reasons to take a break are personal in 78.3% of the cases, of which 36.2% are due to a life event or a financial issue, unrelated to the developer’s desires or the project [20]. To reduce external influence, we examine only active projects, hence the reason for leaving the project is not project termination.

Another problem is that retention is a binary function, so it lacks the fidelity of a continuous function. For a developer working for years in a project we can only claim a similar level of motivation in the first years, and lower motivation in the last year. Continuous labeling functions do not have this limitation and allow us to better quantify the motivation level. Another limitation of retention as a labeling function is that we can know it only in retrospect. In research we can indeed use past data for investigation, but this labeling function cannot help estimating the current motivation.

Other than the developer, retention is influenced by the project characteristics. The retention in large projects is 65% of the retention in medium ones and only 18% of that in single person projects. Retention in new projects is 25% higher than in old ones. The retention in projects belonging to companies is 79% that in projects owned by other organizations. Retention in extraordinarily popular projects is 21% of that in projects of low popularity. All these effects are not surprising, as they are associated with feelings of community and ownership, known motivators — and thus retention [14, 38, 45].

Nevertheless, such influences of the project characteristics might lead to systematic biases of the labeling function. To cope with this, we first identify these biases. We use control variables to verify that a behavior is not due to the control. We verify that the function is predictive despite the biases, and we use several functions to have diversity and reduce the impact of each specific bias.

3.2 Refactoring

While quitting a project can be a strong indication of lack of motivation, it is a binary indicator that usually happens only once. We wanted the other functions to allow continuous monitoring of motivation and its levels. Therefore, we base them on recurring activities and measure the level of activity.

Refactoring is improvement of the design of code while keeping the same functionality [26]. Since refactoring does not add functionality, its value is not always seen from an external point of view (e.g., of the developers' managers or customers). Investing in refactoring therefore reflects motivation on the part of the developer. As far as we know, we are the first to use refactoring as an indication of motivation. However, it is known that improvement activities in other contexts are hard and require motivation [39].

We identify refactoring activity using a linguistic-analysis-based classifier applied to the commit messages documenting the change done [6, 21]. In order not to be dependent on the number of commits, we use the refactoring probability, namely the ratio of refactoring commits to total commits [4].

In some of the use cases we need a binary function instead of a continuous one. We turned all our activity-based labeling functions into binary ones by comparing each developer's activity to the mean activity in our survey dataset (See Section 5.1). For refactoring this cut-off is at 20% refactoring probability, which is the 69th percentile of cases in GitHub. This threshold is chosen for its simplicity, and we show in Sections 5.2 and 6.1 that the influence of the specific threshold is limited.

Refactoring is also influenced by the project characteristics. The refactoring probability in old projects is 37% higher than in new ones. In large projects (with many developers), the refactoring probability is 13% higher than in a single person project, and 37% higher than in small ones. In extraordinarily popular projects, the refactoring rate is 60% higher than in low popularity projects. We use control variables to avoid false impact attribution.

3.3 Diverse Working Hours

A motivated worker might start working earlier or stay later when needed. The correlation between motivation and overtime [16] also supports this metric.

Using the commits' timestamp, we identified each developer's working hours in a whole calendar year. We used the number of distinct hours of the day in which commits were performed as our metric. Hence the maximal value is 24 hours, and a single sleepless working day should be enough to reach it. On the other hand, a person working 9 to 5 will have the value of 8. We did not use the sum of working hours since an unmotivated full-time employee will probably still work more hours than a week-end motivated hobbyist. The cut-off value for binarization (the mean) is working at least 18 hours, reached at the 71st percentile.

The longer the period a person contributes to a project, the more likely the person is to contribute in diverse hours, regardless of motivation. This is supported by the Pearson correlation between activity days and distinct hours, which is 0.59. The threat due to this correlation increases since activity days also have 0.19 correlation with retention. However, diverse hours have a higher 0.26 correlation with retention, so at least part of it is not due to activity days.

Moreover, the activity in a specific hour of the day, per developer, year, and project, is due to a single commit in 43% of the hours, and at most two in 59%. Hence, the contribution in these hours is very sensitive and a single commit might change the number of distinct hours. Indeed, a single sleepless night is enough to reach 24 distinct hours. However, deciding to devote your sleepless night to programming might be a strong indicator of motivation.

3.4 Long Commit Messages

Commit messages are used to document the change done when committing code. The content of the message might contain the change, the reason to perform it, administrative details, etc. [21]. As far as we know, we are the first to use message length as an indicator for motivation. Yet, documentation is considered to be a tedious task and requires motivation [51]. We use the average length of the messages as indication to the motivation and investment in writing them. The cut-off value for high average message length is above 84 characters, reached at the 59th percentile.

Note that messages can be very long. The 99th percentile is 1,204 characters and there are also messages of millions of characters. These are probably the result of mechanisms that automatically generate very long messages. In "Squash commits" the work in several commits is aggregated into one and their messages are combined. Some tools automatically add the "git diff", the summary of the modifications to the code. 9% of the messages above 10k characters mention squash, compared to only 0.1% in shorter messages; diff is mentioned in 37% compared to 1.2%. In such cases the long message is not an indication of motivation and investing effort in writing it. One could take it further and claim this is an indication of lack of motivation to remove a "git log polluting" too long message.

Also, the average message length in single-developer projects is 47 characters, compared to the almost 3 times more (140 characters) in others. This is probably since single developers write messages for 'future me' while in larger projects the community needs and enforces this documentation. In twins experiment, comparing the same developer in a single developer project and larger ones, the average length is shorter in the single developer project in 59% of the cases.

4 METHODOLOGY

4.1 Analysis

The goal of our methodology is to validate that we represent motivation well and capture the relations between motivation and developer performance. We start by using 4 labeling functions, to reduce the influence of a specific function's artifacts. As ground truth we use developers self-reporting on two motivation questions. We verify that the functions weakly predict them. Once being weak classifiers is established, we validate the functions on a large scale, with respect to each other, using the following methods:

- We measure monotonicity with respect to retention.
- We validate agreement at the developer level using twin experiments, showing that a higher value in one function tends to agree with higher values in the others.
- We validate temporal agreement, that an improvement in one function is predictive of improvements in the others, as expected when measuring improvement in the same concept.
- We use control variables to verify that the relations are not due to co-founding, both with a single control and with all in a supervised model.

Once we validate the representation of motivation, we investigate its relation to developer performance. We measure activity in project, output, and process motivation [55], each with two metrics. We evaluate the relation between each of the metrics and the functions. Other than predictive analysis, we used co-change and twin analysis and the control variables. The redundancy in the representation, and investigation in the population level, developer level, and temporal level, reduce the threat of misidentification.

We use the following control variables. Age groups, divided into projects before GitHub creation (before 2008), the oldest 25% (before 2014), the youngest 25% (since 2017), and the middle 50% [7]. Developers were grouped into 'Single', 'Small' (at most 10), 'Medium' (at most 100) and 'Large'. Popularity groups were divided into 'Low' (lowest 25%, at most 8 stars), 'Medium' (next 50%, at most 422 stars), 'High' (next 20%, at most 5027) and 'Extraordinary' for the top 5%. Projects belonging to a company were identified by manually labeling the 100 users with most projects. For programming languages, we control for: Python, JavaScript, Java, C++, PHP, and 'other'.

4.2 Motivation Survey Dataset

As a first validation of the labeling functions, we wanted to compare them to answers regarding motivation. We used a survey by Amit and Feitelson asking various questions regarding motivation [8]. To match the answers with the actual behavior the survey also asked for the GitHub profile and projects.

The survey included 66 questions about motivation and software development, covering 11 motivators as learning, recognition, etc. Our goal here is to establish first the ability to measure motivation and its influence in general. Therefore, we use only a few relevant questions here and leave labeling functions for motivators (e.g., people in popular projects report high recognition motivation) to future work.

The questions used in the labeling functions validation are:

- I regularly have a high level of motivation to contribute to the repository (based on [46])
- How many hours a week do you work on the repository (average)?
- I'm being paid for my work in this repository

The survey was conducted from December 2019 to March 2021. It obtained 1,724 responses, 521 of them finished the survey. The participants provided the names of 484 projects and 303 personal GitHub profiles.

After a year, a follow-up survey was sent to the participants that provided their emails in the original survey. In the follow up survey, 124 out of the 341 participants answered (36.3%).

4.3 GitHub Dataset

GitHub is a platform for source control and code development projects, used by millions of users. Our dataset is based on the BigQuery GitHub schema, which includes the commit history of select projects. We start with all projects with 50 or more commits during 2021. We excluded forks, redundant projects, and non-software projects [7], ending with 18,958 projects.

Many of the developers contributing to a project make only occasional, sporadic contributions, sometimes a single commit. For example, they may fix a bug found while working with the project or add a small functionality for self-use. These developers do not represent well the typical motivations of involved developers. Our focus is on the developers who make significant contributions to the project and are in some way invested in it. We choose to use the threshold of 12 commits per year, an average of one commit per month, as a lower bar for involvement [7]. While this omits 62% of the developers, they are responsible for only 6% of the commits.

To reduce the threat of bots [31], we also filtered out developers with 1,000+ commits per year, 0.04% of the developers. Note that since we started with projects active during 2021 and examined their history, none of the developers stopped working on a project because the project was terminated.

5 VALIDATION OF LABELING FUNCTIONS

5.1 Labeling Functions Validation by the Survey

In this section we validate the labeling functions by comparison to answers in the survey regarding motivation. We asked survey respondents for their GitHub profile, which allows us to match their actual behavior with their answers.

We perform the validation in a supervised learning manner, using a classifier to predict a concept. The 'Concept' column in Table 1 is the question for which we try to predict a high answer. We used both the motivation question and the working hours question. 'Classifier' is the way we predict the concept — by using a high value in either a labeling function, or in the other motivation question (That is, we used the motivation question to predict the working hour question and vice versa). Also, we compared high answers to motivation questions in the original and the follow-up surveys of the same developer contributing to the same project.

It is generally accepted that motivated workers work longer hours [16]. This result may be tainted by mixing data about paid developers with data about volunteers, both common in open-source projects. We checked this by separating the groups using the survey question about payment. For unpaid workers the reported average working hours were 10.8 (high motivation) and 4.5 (low), while for

Table 1: Validation of labeling functions using survey answers

Concept	Classifier	Cases	Accuracy	Accuracy Lift	Precision	Precision Lift
Motivation Answer	Retention	28	0.57	0.19	0.85	0.13
	High Refactoring	28	0.54	0.15	0.83	0.11
	High Hours	28	0.68	0.27	0.88	0.17
	Long Messages	28	0.43	0.04	0.78	0.04
	Working Hours Answer	245	0.65	0.30	0.70	0.35
Motivation Answer Follow-Up	Motivation Answer	46	0.72	0.47	0.63	0.45
Working Hours Answer	Retention	21	0.52	0.05	0.56	0.06
	High Refactoring	21	0.57	0.15	0.60	0.15
	High Hours	21	0.52	0.05	0.55	0.04
	Long Messages	21	0.52	0.07	0.60	0.15
	Motivation Answer	245	0.65	0.30	0.56	0.35
Working Hours Answer Follow-Up	Working Hours Answer	47	0.81	0.53	0.65	1.04

paid workers they were 27.9 (high) and 25.8 (low). Therefore in Table 1 we used only the behavior of unpaid developers to compare to the working hours question. In the rest of the analysis we do not do this filtering and use all developers.

'Cases' is the number of developers whose data was used in each row. When comparing questions, we have nearly 250 cases, and when comparing to the follow-up we have nearly 50 cases. However, when comparing answers to actual behavior the numbers are lower, since this requires a combination that occurs for only a small fraction of the survey respondents: they need to both provide their profile, and we need to have their project in our dataset. Note that we reached these numbers from a relatively big survey completed by 521 people with partial replies from 1,724 (a big drop due to not contributing to GitHub). Hence, it will be hard to enlarge this dataset significantly. Instead, to increase validity, we analyze in the next sections the full GitHub dataset, having years of activity in 18,958 projects by 151,775 developers.

The analysis is in the supervised framework, and we present accuracy, precision, and their lifts. The lift, $\frac{Accuracy - Independent\ Prob}{Independent\ Prob}$ (and likewise for precision with respect to positive rate), measures how much larger the result is relative to the base probability. Note that all labeling functions have positive accuracy lift and precision lift with respect to both questions, thereby validating them.

As expected, comparing with questions leads to higher performance than comparing with labeling functions; In particular, comparing with the same question in the follow-up survey leads to the highest scores. This provides a benchmark with which to compare the results for the labeling functions. While the performance achieved by the labeling functions is lower, it is not too low, and it is sufficient for the requirement from weak classifiers that can be applied at scale.

5.2 Labeling Functions Validation by Monotonicity

Correlation between two binary variables (e.g. high distinct hours and retention) is indicative of agreement, since the probability of accidental agreement decreases with the number of samples and level of agreement. We use retention as a **proxy** for motivation and compare the continuous labeling function to it. Retention has Pearson correlation of 0.26 with hours, 0.006 with message length, and 0.005 with refactoring probability. The first value is medium-low and the last two are almost zero.

Monotonicity [35, 53] adds to correlation by requiring a step by step increase and not only a general increase in both variables together. This is since even when there is some correlation between the variables, the probability of having an increase in every step by mere chance is low.

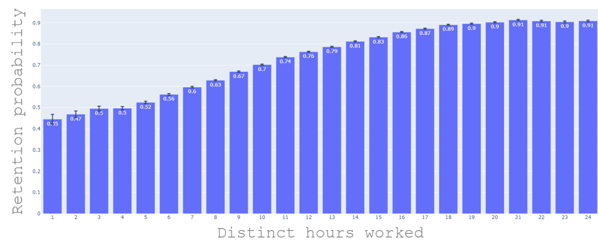


Figure 1: Retention in next year given distinct hours worked in the current one.

Figure 1 presents a "text-book graph" of the monotonicity of working hours and retention probability. The probability of retention also increases with average message length, but only in the lower part and then it is rather flat.

With refactoring there is no positive monotonicity. The differences in retention between deciles are small, sometimes decreasing, and inconsistent.

Hence, we see monotonicity of retention with respect to diverse working hours, partially with message length, but not with refactoring.

5.3 Labeling Functions Validation by Twins Experiments

In Figure 1 we show correlation between working hours and retention, in the population. Twin experiments allow us to investigate the correlation at the individual person level.

When a person is motivated in a project, it might be due to the person, the project, or the interaction between them. The ability to factor out possible influencing variables, by equating them, helps focusing on the other variables. A popular method for that in psychology is "twin experiments" [57]. Identical twins have the same genetic background, so a difference in their behavior is attributed to another variable (e.g., being raised differently). This idea was used in software to profile malware of the same developer [11], and to investigate software quality [5, 7].

We analyze the results of two types of twin experiments, where the twins differ in their retention. In the first one we observe the same developer in the same year in two projects. We choose pairs of projects such that in one the developer continued and in the other the developer left. This setting factors out the developer, yet the projects are different.

In the second type we observe the same developer in the same project in two consecutive years: the developer's last year in the project, and the one before it. Here we factor out the developer and the project, but not completely, because people and projects change over time.

Table 2: Validation of labeling functions by same developer twins experiments, using function agreement with retention being more than 0.5

Twins Type	Devs.	Pairs	Refactor Prob.	Distinct Hours	Msg. Length
Same year, different proj.	7,856	1,314,536	0.95	0.59	0.56
Same proj., consecutive years	42,087	51,549	0.58	0.68	0.49

Table 2 presents the twin experiments results. The columns show the probability that the labeling function is at least as good in the continuing case (indicating retention and implying higher motivation). Taking no influence as the null hypothesis, we expect a probability of 50%. Note that the probability is always higher, other than for message length in consecutive years, which is close to 50% from below. The very high probability of 95% for refactoring in the same year is due to developers not doing refactoring in both cases. When we ignore cases where both are zero the probability is 58%. The probability of equality was small in all other cases.

We also analyzed the twin experiments subject to the control variables. In the 'Same year' case controlling for company, number of developers, popularity, and programming language lead to the same behavior as without controlling. In the 'Consecutive years' case we got the same behavior when controlling for age, number of developers, and popularity. We sometimes got better message length when controlling a company and a programming language. Overall, this shows that the results are rather robust to the controls.

We also compared advantages in one continuous labeling function given advantage in another. In the first analysis we used twins which were the same developer, in the same year, in different projects, regardless of retention. For example, we check the probability of higher distinct hours in project A relative to project B, given a higher refactoring probability in project A relative to project B. In all cases there was a positive lift. Using controls, there was a positive lift in 200 out of 210 cases. In a similar way, we analyzed the same developer, in the same project, in consecutive years regardless of retention. All cases had a positive precision lift too. 118 out of 126 controls had positive lift.

5.4 Labeling Functions Validation by Co-Change

If a person becomes more motivated, we expect an improvement in all our labeling functions. However, we cannot measure the person's motivation directly but only using our labeling functions. If all the labeling functions reflect the person's motivation, an increase in one function is expected to correlate with an increase in the other functions. In co-change analysis [5, 7], we check this expected correlation between the labeling functions, omitting the person's motivation which is hidden from us.

We compare the change of two metrics on the same developer in the same project in two consecutive years. We use an improvement in one metric as a classifier predicting an improvement in the other concept and measure the precision and precision lift. Metric improvement is defined as an increase of our continuous labeling functions.

Table 3 presents the results of the co-change analysis. 'Classifier' is the metric that improved. 'Concept' is the metric that we checked its probability of improvement given an improvement in the classifier. In each cell we present the precision and in parenthesis the precision lift. Note that precision lift is a symmetric function and stays the same when replacing the classifier with the concept. The diagonal is empty since it represents the comparison of a metric with itself.

Table 3: Validation of labeling functions using co-change precision (precision lift in parenthesis)

Predicted concept	Classifier		
	Messages	**Refactoring**	**Hours**
Messages		0.20 (0.16)	0.46 (0.04)
Refactoring	0.65 (0.16)		0.45 (0.02)
Hours	0.58 (0.04)	0.18 (0.02)	

In all cases the precision lift is positive, indicating an increased probability of improvement in one metric, given an improvement in the other. Using controls, out of the 186 control cases, only in 18 cases there was a negative lift. Hence, the co-change analysis also shows high robustness with respect to controls.

5.5 Labeling Functions Reliability

We would like our functions to be reliable, returning similar results when measuring the same entity again. However, we cannot measure the same labeling function on the same developer twice at the same time.

As a second best, we compare the value of the labeling functions for the same developer, in the same project, in consecutive years [7]. Consecutive years are not the same time yet not too far from it. Given the value in one year and the consecutive year, we compute two metrics. Self-Pearson is the Pearson correlation of the pairs

of metrics for all developers in all projects. Hence, it measures the relative change in the developer ranking as time goes by. Relative difference is the average of the difference between the metric values in the consecutive years, divided by the value in the first year. This metric ignores the rest of the population and measures the average difference between two measurements. Hours had self Pearson of 0.59, and average relative difference of 5%. Refactoring had self Pearson of 0.63, and average relative difference of -13%. Message length had self Pearson of 0.11, rather low, and average relative difference of 15%. Note that average message length is very sensitive and even a single long message can change it dramatically.

6 RESULTS

Given the labeling functions for motivation, we now turn to seeing how they can be used with the GitHub dataset.

6.1 Predicting Retention

The first use is to predict the retention using the other labeling functions. Early prediction of abandonment is important because it allows intervention and possibly avoiding churn. Table 4 presents retention probabilities given the labeling functions. The 'Classifier' column is a labeling function. The 'Retention' column is the precision of predicting the retention concept, given that the classifier is high. In the 'Two Years Retention', we extend the co-change analysis and the classifier is **an improvement** in the metric from one year of a developer in a project to the next year; the column shows the retention rate in the next year.

Table 4: Labeling functions' predictions of retention

Classifier	Retention	Retention Lift	Two Years Retention	Two Years Lift
None	71	-0.058	75	-0.073
High Refactoring	75	0.003	81	-0.001
Long Messages	77	0.025	81	0.009
High Hours	90	0.196	86	0.061
All	90	0.204	86	0.063
Positive Rate	75	0.0	81	0.0

The baseline for each analysis is the positive rate, the probability of retention. In both cases, when none of the labeling functions is high, the retention rate is lower, and when all are high it is the highest. The lift of high hours is almost as high as for all labeling functions together. Long messages have 2.5% lift in retention and the lift in the other cases is close to zero.

To learn if the predictive power is due to the labeling functions or the controls, we compared the performance of models built on the functions, the controls, and both. Aiming for precision, logistic regression models reached precision of 91% and recall of 21% on both the labeling functions alone and when adding the controls. Using the controls alone it reached precision of 90% yet with recall of only 7%, hence the labeling functions have better predictive power, above the contribution from the controls.

We also investigated if the prediction can be improved by using the raw metrics and not the labeling functions (e.g., knowing of 24 distinct hours and not just a Boolean value). Logistic regression had a precision of 89% (2 percentage points less) yet recall of 37% (16 percentage points higher), higher Jaccard, and higher mutual information. This indicates that their use might be beneficial in some settings yet without a dramatic change.

Note that logistic regression is a low-capacity model and our dataset is large, reducing the threat of over-fitting. We also checked high-capacity models such as random forests, boosting, and neural networks to build models of higher representation ability and performance. They had lower performance, indicating that representation power is not the limiting constraint.

We used the 'Two Years Retention' to try to predict retention in the second year. Our co-change analysis (Section 5.4) used a single metric. We now apply the full power of supervised learning to predict changes. We allow more inputs and provide the functions in one year, the year afterwards, the difference (to ease representation), and controls. On this dataset we can apply any supervised learning classifier, learning complex and powerful representations. We were able to reach precision of 92% with recall of 33%, higher than the precision of 86% when all metrics improve. Hence, the application of the more powerful method is beneficial, yet since the baseline result is high it is not dramatic.

6.2 Developer Performance by Labeling Functions

We next use the labeling functions to estimate the relation between motivation and various metrics of developer performance. Table 5 has a 'Metric' column and a 'Description' column, explaining the metrics. There is an additional column per labeling function. The cell intersecting a labeling function column and a metric row represents $\frac{Avg(metric|function=High)}{Avg(metric|function=Low)}$, where 1.0 means similar averages. Note that the labeling functions are weak classifiers, labeling some motivated people as unmotivated and vice versa. Therefore the results are not accurate yet indicate the nature of the relations with motivation.

The first four rows match the labeling functions with each other. When a column labeling function is high, the average of the other labeling functions raw metrics is expected to be high, if they all capture motivation. This indeed happens in all cases, except the disagreement between hours and message length, providing an additional validation of the functions.

We are interested in three performance aspects: activity, output, and process motivation. We measure each aspect with two metrics to reduce the influence of a single metric's artifacts. Activity is measured by activity days, and, to better fit work by both full-time employees and volunteers, activity period. Output is measured by commits and files as units of work. Commits and other output units in software engineering are not of a single size (e.g., a commit might represent different amounts of work). However, commits are a common way to measure output [1]. We did not used issues or pull requests, which are not available in our dataset, yet measuring by either commits, issues, or pull requests tend to agree [7].

Developers might be driven by output motivation (wanting to produce more) or process motivation (produce better) [55]. Corrective commit probability is influenced by both the existence of bugs and their detection effectiveness. Process motivation might lead to investing more in detection effectiveness (e.g., by writing tests)

Table 5: Developer performance by labeling functions

Metric	Description	Reten.	Hours	Msgs.	Refact.
Retention	Probability of continuing in the next year		1.29	1.05	1.01
Commit Hours	Number of distinct hours of the day during a year	1.25	1.85	0.99	1.04
Message Length	Average number of characters in a commit message	1.13	0.91	5.75	2.41
Refactor Prob.	Attempt to improve the software [6, 7]	1.04	1.12	2.02	6.70
Activity Period	Days between the first and last commits in a year	1.70	1.41	1.11	1.01
Activity Days	Number of distinct days in which a commit was made	2.09	4.06	0.79	1.03
Commits	The number of commits, modifications of the code	2.08	4.03	0.80	1.03
Files Edited	Number of Files modified (or created)	1.62	2.99	0.84	0.99
Commit Duration	Average gross duration of a commit [3, 5, 7]	1.07	1.26	1.44	1.19
Corrective Commit Prob.	The ratio of bug fixing commits, Measures bug fixing effort [7]	1.08	1.03	1.98	1.57

which can lead to finding more bugs. Core developers abandoning the project (possibly lacking motivation) reduce effectiveness [7]. Similarly, commit duration is influenced by productivity and process, e.g., writing tests increase commit duration by 18% [3].

We expect that motivation will lead to higher involvement, as reflected by a longer activity period, more distinct activity days, more commits, and more files edited [55]. Our results indicate that in general this indeed happens. Activity period is longer for all functions. The increase in activity days is higher than the one of activity period in all cases other than messages where it is even lower than one. The ratios for commits are almost identical to those of activity days. However, since commit duration is longer, more time was invested to perform these commits. The ratios of files edited are lower than for commits but follow a similar pattern.

When we look at the table with respect to the labeling functions, metrics are always higher for retention and hours. For refactoring it holds other than the 0.99 for files edited. Long messages have a significant drop to 0.79 in activity days. Part of this is due to confounding variables like the tendency to short messages and long activity periods in projects of few developers. When controlling by developer group, the activity period is higher given long messages. Commits and files edited also have a drop yet per active day they improve.

The results can also be used to address cases where motivation can be hypothesized to have opposite effects [55]. For example, it may be claimed that motivation compensates for the tedious effort of fixing bugs, and therefore motivated individuals will perform more bug fixing. Alternatively, motivated individuals might apply more attention to their work, increasing quality, and therefore will have less bugs and require less bug fixes. The results indicate that the first hypothesis dominates: using all four labeling functions, higher motivation seems to go with higher corrective commit probability.

The same goes for commit duration, which is longer for all functions. One could expect a decrease due to output motivation and higher productivity. An increase can be explained by process motivation, higher attention and standards. Hence, our results better fit process motivation, aiming to produce better, than they fit output motivation, aiming to produce more [55].

As an additional validation, we compare the metrics in same-year twins experiments, comparing a project in which the developer continued to one abandoned. Commits are higher in the continued project in 65% of the twins pairs, files edited in 56%, activity period in 74%, activity days in 65%, and commit duration is higher in 55% (at random 50% is expected). Opposed to the table, CCP is lower in 86%. Results hold when controlled by any of our control variables.

Co-change analysis showed the improvements in the labeling functions lead to higher metric values for all the metrics. These results are the same as in the table.

7 RELATED WORK

Demarco and Lister [24], and also Frangos [27], claim that the important software problems are human and not technological. So there has been intensive investigation of motivation in software engineering [17, 28, 40].

Our work, and specifically the labeling functions, were designed to align with psychological motivation theories. Commits, refactoring, and hours are aligned with McClelland's [45] affiliation and achievement, and in certain contexts authority. Vroom's Expectancy Theory [59] predicts higher outcome from refactoring and documentation for motivated developers planning to stay. All our functions are aligned with ownership (e.g., Motivation-Hygiene Theory [34], others [14, 38]) and more motivators.

Open-source development is the collaborative development of software that is free to use and further modify [48]. It is common to develop open source software as a volunteer, which means that salary is not a motivator [41]. Therefore, the motivation of open source developers was investigated as a specific domain, in an effort to uncover other motivators [22, 60].

Ownership and autonomy are important motivators [14, 38]. We saw in Section 3.1 that retention is significantly higher in smaller projects, in which ownership and autonomy are high. Recognition [30] is another motivator. Recognition is stronger in popular projects that have lower retention. This is aligned with external motivators like recognition being weaker than internal ones like ownership [2].

Touré-Tillery and Fishbach distinguish between output motivation (producing more) and process motivation (producing well) [55]. We saw that commit duration increases, by all labeling functions, indicating process motivation — giving more attention to each commit instead of trying to finish them faster. They also claim that motivation is demonstrated in choice, speed, and performance.

8 THREATS TO VALIDITY AND LIMITATIONS

We discussed the limitations and biases of each labeling function when presenting it.

We base our work on machine learning frameworks [10, 47, 50]. However, the application of these frameworks to motivation and software engineering is new and therefore there are no benchmarks or other labeling functions to which we can compare. Instead, we validated the functions in many ways to reduce the risk of error.

Studying motivation poses a challenge since motivation is an internal abstract concept.

For example, survey answers might have reliability problems due to ego defenses [15], subjectivity, different personal scales, etc. Comparison of self-rating to those of a related person (e.g., supervisor, co-worker, or a spouse) showed only moderate correlation [12, 36, 43]. Therefore, we also independently evaluated the actual behavior working at GitHub on 151,775 developers.

The number of survey answers that were matched with the behavior in GitHub is low. Note that this is despite the survey itself being completed by 521 developers. Hence, there was a low probability of being able to match a person. This led to threats of both response bias [32] and incorrect statistical estimation problems [58]. Since the survey is already large it seems it will be hard to do a larger survey. Cooperation with companies, which have behavior information and might agree to conduct surveys is a possible option.

Motivation might be due to many motivators, like enjoyment, self-use, community, etc. We did not consider all these motivators and possible relations between them but only the outcome as motivation. For example, it is possible that people motivated by self-use will contribute only a single modification that helps them and therefore retention is irrelevant to their motivation. The survey we used included questions on 11 motivators and in future work we can apply the same methodology and validate labeling functions for specific motivators, getting a finer-grained picture of people's motivations.

A hard to notice threat is due to the motivation level. All the developers that we analyze contributed at GitHub hence are somewhat motivated in the first place. Hence, instead of comparing motivated and unmotivated people, we might have compared motivated and highly motivated people. This might turn out to be a benefit since members of organizations, and communities also have minimal motivation, as in our scenario.

9 CONCLUSIONS

GitHub contains years of data on thousands of developers in their natural every-day software development. We suggest four labeling functions based on behavioral cues [25] which enable using this data to study motivation and its effects in software development.

We first validated that the labeling functions are weak classifiers by predicting developers' answers to two motivation questions from a survey. We then checked agreement between the functions, monotonicity, agreement per person using twin experiments, and temporal agreement using co-change. We used control variables, alone and combined in a supervised learning model, to verify that the labeling functions add predictive power beyond these variables.

Our results reproduce prior work on the positive impact of motivation: the activity period is up to 70% longer, up to 44% higher time investment in commit, and up to 300% more commits. We also built models for developer retention. A high precision retention model can be used to identify dedicated developers on which a project can rely. A high recall model can be used to identify developers lacking motivation (those not identified by the model), allowing intervention that might increase it.

Our application of the methodology to motivation is just one example. Additional labeling functions can be used to obtain even better characterizations. Specifically, our survey included questions about 11 motivators, for which labeling functions can be built allowing a drill down into motivation details.

More importantly, the same methodology can be used for studying other concepts, especially when one cannot obtain a precise labeling of the concept due to its ambiguity, the cost of labeling, or noise. Using our methodology facilitates quantified, reproducible, long-term investigation, based on large-scale data from real projects.

EXPERIMENTAL MATERIALS

The replication package (DOI 10.5281/zenodo.10519880) can be found at https://zenodo.org/records/10519880. Most up-to-date version is available at https://github.com/evidencebp/motivation-labeling-functions.

ACKNOWLEDGMENTS

We thank David Amit, Daniel Shir, Assa Bentzur, Yaniv Mama, Yinnon Meshi, Aviad Baron, and Gil Shabtai for the discussions and their insights.

REFERENCES

[1] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. 2008. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *2008 16th IEEE International Conference on Program Comprehension*. 182–191. https://doi.org/10.1109/ICPC.2008.24

[2] Teresa M. Amabile, Karl G. Hill, Beth A. Hennessey, and Elizabeth M. Tighe. 1994. The work preference inventory: Assessing intrinsic and extrinsic motivational orientations. *Journal of Personality and Social Psychology* 66, 5 (1994), 950–967. https://doi.org/10.1037/0022-3514.66.5.950

[3] Idan Amit. 2020. Software development task effort estimation. U.S. patent application #US20220122025A1. https://patents.google.com/patent/US20220122025A1/

[4] Idan Amit. 2021. End to End Software Engineering Research. *arXiv preprint arXiv:2112.11858* (2021). arXiv:2112.11858 [cs.SE]

[5] Idan Amit, Nili Ben Ezra, and Dror G. Feitelson. 2021. Follow Your Nose – Which Code Smells are Worth Chasing? *arXiv preprint arXiv:2103.01861* (2021). arXiv:2103.01861 [cs.SE]

[6] Idan Amit and Dror G. Feitelson. 2019. Which Refactoring Reduces Bug Rate?. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering* (Recife, Brazil) *(PROMISE'19)*. Association for Computing Machinery, New York, NY, USA, 12–15. https://doi.org/10.1145/3345629.3345631

[7] Idan Amit and Dror G. Feitelson. 2021. Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal* 29, 4 (Aug 2021), 817–861. https://doi.org/10.1007/s11219-021-09564-z

[8] Idan Amit and Dror G. Feitelson. 2024. A Large Scale Survey of Motivation in Software Development and Analysis of its Validity. *arXiv preprint arXiv:2404.08303* (2024). arXiv:2404.08303 [cs.SE]

[9] Idan Amit, Eyal Firstenberg, Jonathan Allon, and Yaron Neuman. 2020. Identifying changes in use of user credentials. US Patent 10,686,829.

[10] Idan Amit, Eyal Firstenberg, and Yinnon Meshi. 2017. Framework for semi-supervised learning when no labeled data is given. U.S. patent #US11468358B2. https://patents.google.com/patent/US11468358B2

[11] Idan Amit, John Matherly, William Hewlett, Zhi Xu, Yinnon Meshi, and Yigal Weinberger. 2019. Machine Learning in Cyber-Security - Problems, Challenges and Data Sets. *arXiv preprint arXiv:1812.07858* (2019). arXiv:1812.07858 [cs.LG]

[12] M. Argyle. 1989. Do happy workers work harder? The effect of job satisfaction on job performance. In *How harmful is happiness? Consequences of enjoying life or not*, Ruut Veenhoven (Ed.). Universitaire Pers, Rotterdam, The Netherlands.

[13] Dan Ariely, Emir Kamenica, and Dražen Prelec. 2008. Man's search for meaning: The case of Legos. *Journal of Economic Behavior & Organization* 67, 3-4 (2008), 671–677. https://doi.org/10.1016/j.jebo.2008.01.004

[14] Nathan Baddoo and Tracy Hall. 2002. Motivators of Software Process Improvement: an analysis of practitioners' views. *Journal of Systems and Software* 62, 2 (2002), 85 – 96. https://doi.org/10.1016/S0164-1212(01)00125-X

[15] Nigel Bassett-Jones and Geoffrey C. Lloyd. 2005. Does Herzberg's motivation theory have staying power? *Journal of Management Development* 24 (12 2005), 929–943. https://doi.org/10.1108/02621710510627064

[16] Debby GJ Beckers, Dimitri van der Linden, Peter GW Smulders, Michiel AJ Kompier, Marc JPM van Veldhoven, and Nico W van Yperen. 2004. Working overtime hours: relations with fatigue, work motivation, and the quality of work. *Journal of Occupational and Environmental Medicine* (2004), 1282–1289.

[17] Sarah Beecham, Nathan Baddoo, Tracy Hall, Hugh Robinson, and Helen Sharp. 2008. Motivation in Software Engineering: A systematic literature review. *Information & Software Technology* 50, 9-10 (2008), 860–878. https://doi.org/10.1016/j.infsof.2007.09.004

[18] Sharon Belenzon and Mark A Schankerman. 2008. Motivation and sorting in open source software innovation. *Available at SSRN 1311136* (2008).

[19] Avrim Blum and Tom Mitchell. 1998. Combining Labeled and Unlabeled Data with Co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory* (Madison, Wisconsin, USA) *(COLT' 98)*. ACM, New York, NY, USA, 92–100. https://doi.org/10.1145/279943.279962

[20] Fabio Calefato, Marco Aurelio Gerosa, Giuseppe Iaffaldano, Filippo Lanubile, and Igor Steinmacher. 2021. Will You Come Back to Contribute? Investigating the Inactivity of OSS Core Developers in GitHub. arXiv:2103.04656 [cs.SE]

[21] Leshem Choshen and Idan Amit. 2021. ComSum: Commit Messages Summarization and Meaning Preservation. *arXiv preprint arXiv:2108.10763* (2021). arXiv:2108.10763 [cs.CL]

[22] Andrea Bonaccorsi Cristina and Cristina Rossi. 2004. Altruistic individuals, selfish firms? The structure of motivation in Open Source software. *First Monday* 9 (2004), 9. https://doi.org/10.5210/fm.v9i1.1113

[23] Hendrik Dahlkamp, Adrian Kaehler, David Stavens, Sebastian Thrun, and Gary R Bradski. 2006. Self-supervised monocular road detection in desert terrain.. In *Robotics: science and systems*, Vol. 38. Philadelphia.

[24] Tom DeMarco and Tim Lister. 2013. *Peopleware: productive projects and teams*. Addison-Wesley.

[25] Giel Dik and Henk Aarts. 2007. Behavioral cues to others' motivation and goal pursuits: The perception of effort facilitates goal inference and contagion. *Journal of Experimental Social Psychology* 43, 5 (2007), 727–737. https://doi.org/10.1016/j.jesp.2006.09.002

[26] M. Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Pearson Education. https://books.google.de/books?id=2H1_DwAAQBAJ

[27] S. A. Frangos. 1997. Motivated Humans for Reliable Software Products. In *Reliability, Quality and Safety of Software-Intensive Systems*, Dimitris Gritzalis (Ed.). Springer US, Boston, MA, 83–91. https://doi.org/10.1007/978-0-387-35097-4_7

[28] C. França, F. Q. B. da Silva, and H. Sharp. 2020. Motivation and Satisfaction of Software Engineers. *IEEE Transactions on Software Engineering* 46, 2 (Feb 2020), 118–140. https://doi.org/10.1109/TSE.2018.2842201

[29] Yoav Freund and Robert E Schapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. Comput. System Sci.* 55, 1 (1997), 119–139. https://doi.org/10.1006/jcss.1997.1504

[30] Marco Gerosa, Igor Wiese, Bianca Trinkenreich, Georg Link, Gregorio Robles, Christoph Treude, Igor Steinmacher, and Anita Sarma. 2021. The Shifting Sands of Motivation: Revisiting What Drives Contributors in Open Source. arXiv:2101.10291 [cs.SE]

[31] Mehdi Golzadeh, Alexandre Decan, Damien Legay, and Tom Mens. 2021. A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments. *Journal of Systems and Software* 175 (May 2021), 110911. https://doi.org/10.1016/j.jss.2021.110911

[32] Walter R Gove and Michael R Geerken. 1977. Response bias in surveys of mental health: An empirical investigation. *American journal of Sociology* 82, 6 (1977), 1289–1317.

[33] Adam Grant. 2008. The Significance of Task Significance: Job Performance Effects, Relational Mechanisms, and Boundary Conditions. *The Journal of Applied Psychology* 93 (Feb 2008), 108–24. https://doi.org/10.1037/0021-9010.93.1.108

[34] F. Herzberg, B. Mausner, and B. B. Snyderman. 1959. *Motivation to Work*. Wiley, New York.

[35] Austin Bradford Hill. 1965. The environment and disease: association or causation?

[36] Timothy A. Judge, Edwin A. Locke, Cathy C. Durham, and Avraham N. Kluger. 1998. Dispositional effects on job and life satisfaction: The role of core evaluations. *Journal of Applied Psychology* 83 (1998), 17–34. https://pdfs.semanticscholar.org/9912/e58168ca993de3fa8105bd1c64fd63f7ddb3.pdf

[37] Michael Kearns. 1988. Learning Boolean formulae or finite automata is as hard as factoring. *Technical Report TR-14-88 Harvard University Aikem Computation Laboratory* (1988).

[38] Chak Fu Lam and Suzanne T. Gurland. 2008. Self-determined work motivation predicts job outcomes, but what predicts self-determined work motivation? *Journal of Research in Personality* 42, 4 (2008), 1109–1115. https://doi.org/10.1016/j.jrp.2008.02.002

[39] BA Lameijer, J Antony, A Chakraborty, RJMM Does, and JA Garza-Reyes. 2021. The role of organisational motivation and coordination in continuous improvement implementations: an empirical research of process improvement project success. *Total Quality Management & Business Excellence* 32, 13-14 (2021), 1633–1649.

[40] Per Lenberg, Robert Feldt, and Lars Göran Wallgren. 2015. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software* 107 (2015), 15 – 37. https://doi.org/10.1016/j.jss.2015.04.084

[41] Josh Lerner and Jean Tirole. 2002. Some Simple Economics of Open-Source. *Journal of Industrial Economics* 50 (02 2002), 197–234. http://www.people.hbs.edu/jlerner/simple.pdf

[42] Brenda L. Mak and Hy Sockel. 2001. A confirmatory factor analysis of IS employee motivation and retention. *Information & Management* 38, 5 (2001), 265–276. https://doi.org/10.1016/S0378-7206(00)00055-0

[43] Christina Maslach, Susan Jackson, and Michael Leiter. 1997. The Maslach Burnout Inventory Manual. *Evaluating Stress: A Book of Resources* 3 (01 1997), 191–218.

[44] John D Mayer, Michael A Faber, and Xiaoyan Xu. 2007. Seventy-five years of motivation measures (1930–2005): A descriptive analysis. *Motivation and Emotion* 31 (2007), 83–103.

[45] D. C. McClelland. 1961. *The Achieving Society*. Van Nostrand, Princeton, NJ.

[46] E. Murphy-Hill, C. Jaspan, C. Sadowski, D. Shepherd, M. Phillips, C. Winter, A. Knight, E. Smith, and M. Jorde. 2019. What Predicts Software Developers' Productivity? *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2900308

[47] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data Programming: Creating Large Training Sets, Quickly. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 3567–3575. http://papers.nips.cc/paper/6523-data-programming-creating-large-training-sets-quickly.pdf

[48] Eric S. Raymond. 1999. *The Cathedral and the Bazaar* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[49] Bishal Sainju, Chris Hartwell, and John Edwards. 2021. Job satisfaction and employee turnover determinants in Fortune 50 companies: Insights from employee reviews from Indeed.com. *Decision Support Systems* 148 (2021), 113582. https://doi.org/10.1016/j.dss.2021.113582

[50] Robert E Schapire. 1990. The strength of weak learnability. *Machine learning* 5, 2 (1990), 197–227.

[51] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. 2015. To Document or Not to Document? An Exploratory Study on Developers' Motivation to Document Code. In *Advanced Information Systems Engineering Workshops*, Anne Persson and Janis Stirna (Eds.). Springer International Publishing, Cham, 100–106.

[52] Frank J Smith. 1977. Work attitudes as predictors of attendance on a specific day. *Journal of applied psychology* 62, 1 (1977), 16.

[53] Sonja A Swanson, Matthew Miller, James M Robins, and Miguel A Hernán. 2015. Definition and evaluation of the monotonicity condition for preference-based instruments. *Epidemiology (Cambridge, Mass.)* 26, 3 (2015), 414.

[54] Kristi Toode, Pirkko Routasalo, and Tarja Suominen. 2011. Work motivation of nurses: A literature review. *International Journal of Nursing Studies* 48, 2 (2011), 246–257. https://doi.org/10.1016/j.ijnurstu.2010.09.013

[55] Maferima Touré-Tillery and Ayelet Fishbach. 2014. How to Measure Motivation: A Guide for the Experimental Social Psychologist. *Social and Personality Psychology Compass* 8 (07 2014). https://doi.org/10.1111/spc3.12110

[56] Leslie G Valiant. 1984. A theory of the learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.

[57] Steven G Vandenberg. 1966. Contributions of twin research to psychology. *Psychological Bulletin* 66, 5 (1966), 327.

[58] V. Vapnik and A. Chervonenkis. 1971. On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory of Probability & Its Applications* 16, 2 (1971), 264–280. https://doi.org/10.1137/1116025 arXiv:https://doi.org/10.1137/1116025

[59] Victor. H. Vroom. 1964. *Work and Motivation*. Wiley, New York.

[60] Yunwen Ye and Kouichi Kishida. 2003. Toward an Understanding of the Motivation Open Source Software Developers. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 419–429. http://dl.acm.org/citation.cfm?id=776816.776867

4.5 Other Papers

In addition to the four papers included above, we wrote some other works that are not part of this thesis yet contributed to it.

Linguistic analysis of commit messages was an important tool in our work. We worked with Swanson's commit taxonomy and defined commits as corrective (bug fix), adaptive (adding a new feature), and perfective (refactor) [69]. We built a classifier for corrective commits for the CCP investigation, and a perfective classifier for the refactoring investigation. In "ComSum: Commit Messages Summarization and Meaning Preservation" [18] we improved the classifiers (later used in Section 4.4).

"Software development task effort estimation" [1] was done while working at Acumen and was submitted as a patent. We investigated in depth commit duration as a metric for productivity. We showed that file reuse leads to higher bug detection efficiency and lower CCP. Commit duration was used in the investigation of CCP and the influence of motivation.

The use of static alerts assumes causality — improve the code by fixing the alert. In "Follow Your Nose — Which Code Smells are Worth Chasing?" [3] we checked which alerts indeed behave like that. We required predictive power, alone and when controlled by the developer or file length, improvement when removed, and monotonicity. Out of more than 170+ alerts provided by CheckStyle, less than 20% had such support. Only 5 had an advantage of more than 10%, needed to justify a practical use. This was a systematic work going over all alerts of a static analyzer that further supported the hypothesis that single interventions are limited in impact. From a methodological point of view, we defined properties for causality and noted that most "best practices" are not up to it. We showed the large influence of the developer and code length and the importance of controlling them. The dataset that we built, and the small set of beneficial alerts, allows an intervention experiment to settle the question of their causality. One can choose a subset of the alerts, do a small change fixing them, and observe the outcome.

In "A fine-grained data set and analysis of tangling in bug fixing commits" [39] we joined many other researchers and investigated tangling commits, serving multiple goals in the same commit. The work provided insights and a dataset on the multiple purposes of individual commits. Tangling commits might be a threat on commit analysis, and therefore measuring how common they are and awareness in the design are important.

"Framework for semi-supervised learning when no labeled data is given" was done in my work at Palo Alto Networks and was submitted as a patent [6]. The content was cyber-security and the method was a way to represent fuzzy concepts (e.g., maliciousness) using labeling functions and finding a maximum likelihood estimation for them. This method, known internally as Archimedes, structured the framework that we used. Weak supervision is a framework where one is required to perform supervised learning

yet with a limited number of labels or even none. The method minimizes disagreements between unrelated labeling functions, which are only required to be weak classifiers. Minimizing the disagreements is done by minimizing the entropy of probability predictions of a parametric model built upon the weak classifiers. Minimizing the entropy is equivalent to finding a representative from the parametric family minimizing the Kullback–Leibler divergence with the underlying distribution. In turn, minimizing the KL-divergence is equivalent to finding the maximum likelihood estimation from the parametric family (e.g., [34], page 148). This framework led to our use of labeling functions for motivation and was extended (e.g., with co-change) to further validate that they measure a similar concept.

Chapter 5

Discussion and Conclusions

5.1 Contributions

Quality and motivation are correctly considered to be important in software development. Our first contribution was to show the large gaps in effort invested in bug correction. These gaps suggest that interventions to reduce bugs might be very beneficial. Our research also supports "Quality is Free" which claims that investment in quality is beneficial in general and leads to *increased* productivity [20].

The second contribution is providing simple ways for code improvement. Feedback about the code, in many forms, is valuable for code improvement. Tests reduce time to detect bugs, and so do many eyeballs [5]. Removal of Self Admitted Technical Debt (e.g., TODO), areas marked as problematic by a developer, is a beneficial refactoring. Code review also increases the probability of beneficial refactoring [4]. Defensive programming interventions (e.g., avoiding wide exception catching that might hide unexpected exceptions) have properties indicating their benefit [3].

The way that software is composed is also an important factor. The benefit of short source code files is well known, and we added co-change analysis to support it [3]. Reducing coupling is also beneficial. Following these two simple pieces of advice is enough to have higher quality than the median [5]. Abstraction is another useful guideline. Refactoring done in order to improve software structure (e.g., reduce coupling) [4] and fixing design static alerts [3] improves software quality.

Aesthetics is also an important guideline, yet it is hard to follow or investigate. 72% of the developers in our survey answered at least 'somewhat agree' (9 on a scale of 1 to 11) on "I want my code to be beautiful" and 35% responded with the maximal answer. Our personal experience, and many informal investigations, indicated the high benefit of aiming for beautiful code and keeping the bar high despite deadlines, trade-offs, and other constraints. Files involved in commits that had the word "ugly" appear in their message have CCP 292% higher and their commit duration was 23% higher than the rest.

Software is developed by millions of developers who do not tend to read software engineering research papers. The concept of feedback is easy to understand, justify, and apply. Composition into small parts is rather easy, abstraction is harder to interpret, and aesthetics leaves the developer with guts feelings. The old recommendation of code reuse (e.g., [14, 11, 29]) suggests how to achieve all of them.

The simplest justification of reuse is that a code reused is code not written again, saving time, and giving less space for new bugs. Refactors that involve reuse are effective in reducing future bug rate [4]. We also showed that file reuse leads to higher bug detection efficiency and lower CCP [1]. Each new reuse case helps to differentiate between the code needed to implement the new functionality and the code encapsulated in the reused component, advocated by the single responsibility principle [52], increasing abstraction and decomposition. Each reuse case is another pair of "eyeballs" that takes a look at the reused component and provides feedback, and by Linus's law [64] increases bug detection efficiency [5]. Aesthetics is personal and subjective yet writing variants of the same logic again and again is considered to be ugly. All this is avoided by reuse.

In the area of motivation, we showed that it can also lead to large performance gaps. We showed that while common motivators indeed tend to be predictive and improvement in them tends to lead to motivation improvement, their influence is limited. None of the motivators is sufficient or necessary. Hence, this hints that one should check with each developer how to increase their motivation. Recognition, and specifically expressing interest, seems to be a general beneficial recommendation, which is easy to apply.

Software development aside, another major contribution is the methodology developed. Our methodology enables defining and quantifying vague concepts and evaluating the definition's reliability and validity. Given a set of concepts, we can investigate the relations between them. We can also predict the outcome of changes by a reduction to supervised learning. These abilities are valuable in many domains.

5.2 Future Work

We built a method to find beneficial recommendations for software improvement and evaluate their value. This left important gaps: application of the method on a large scale, and experimental evaluation of the causal relation.

In order to be able to investigate code patterns that were not suggested before, we created a large dataset for end-to-end learning for software engineering [2]. It contains the source code of 15k projects, fetched every two months from June 2021 to December 2022, along with process metrics. This allows "end-to-end" learning — use two versions of the same file in different dates and use machine learning to find patterns that predict improvement.

We provided 4 labeling functions for motivation in open-source development, verified with dozens of people. More labeling functions, for motivation and other concepts, should be found. Different contexts will shed light on different aspects. Work on other areas will show generalization. Work in companies can provide a stable similar context of employees' work and reduce selection bias. Investigation of disagreements between labeling functions (e.g., using [6]), can help to tune the functions.

We investigate causal relations using observational data. Our representation of change might be limited by lacking data or inappropriate assumptions. However, once we have a low number of potential patterns, we can use intervention experiments, the classical testbed for causality (suggested in software improvement context in [3]). Open-source projects provide plenty of intervention opportunities (e.g., files in which a static alert appears). One can sample some of them, and intervene and fix them in a small, clean change. Then one can wait and check the result of the intervention and compare it to the outcome in the control group.

5.3 Conclusions

Our goal was to look for justifiable recommendations for software quality improvement.

For that, we developed a methodology for capturing and evaluating fuzzy concepts and the relations between them. The methodology for defining concepts started in adjacent years stability and validation by comments [5], was applied to evaluate survey validity (Section 4.3), and further extended by simultaneous twin and co-change relations (Section 4.4).

We applied the methodology to evaluate software concepts like quality and human concepts like motivation, showing large gaps in the metric values and their meaning in software development. The method is applicable at high level as a project and low levels as a file, providing different ways to benefit from it.

We found justifiable recommendations for quality improvement. While there is no single silver-bullet, the influence of recommendations accumulates. In principle, the gap of 650% that we found in the CCP distribution can be explained by 10 independent factors contributing 20% each ($\ln(650)/\ln(120)$). And we indeed found some such factors. Some influence program structure: code length, coupling, cohesion, and reuse. Others influence feedback: tests, code review, defensive programming, and eyeballs. Additional factors that are influential, though not actionable, are software complexity (domain and size) and programming language. An early model based on some of these features predicted CCP groups with 68% accuracy [5].

Of course, this is a too simplistic model; relations between these factors, other factors, and quality are more complex. However, our current knowledge is already enough to

make beneficial improvements. More than that, we provide a methodology that enables the identification and evaluation of more recommendations.

We hope that the methodology will be further developed, that new justified recommendations will be found, that people will apply them and benefit, and that the methodology will contribute to more domains.

5.4 Acknowledgments

I ate many lunches while discussing how to develop software. The discussions were interesting, heated, and usually insightful. However, they tended to be based on incident-analysis, anecdotes, and guts-feeling. I felt that we should have a more effective way to find ways to improve development. So, my lunch mates are part of the reasons I entered this research and influenced my views on the area. Thank you for the pleasant time and discussions.

While teaching in Ehud Lamm's software engineering course, I started not only practicing software engineering but thinking of it as a discipline that can be investigated and developed.

For a long period I thought of applying supervised learning in order to find differences between high and low quality code. However, this requires labels of code quality code. When I attended a lecture describing the work of Zieder et al. [76] on 'just in time defect prediction', I understood that bugs can be used as indications of low quality, which lead later to CCP.

At "Rounds", I decided to treat improvement as a direct goal. The team and I decided to "become an excellent team in six months". We wondered what is the definition of "excellent team", what we should do to improve, how hard will it be, and how long will it take. We were surprised to become an excellent team, by our metrics, our feeling, and external feedback in less than six months.

The main technical methods that we used were code review and testing. However, we were using them before too. The change was the bar that we set. Typically, code contains the grey not-even-tech-debt but could-be-better parts. We stopped compromising and though deadlines and other constraints we invested in fixing them. These experiences lead me to think that there is huge difference in performance level, there are ways to improve the performance, yet the personal aspect and not the technical aspect is the main one.

Eyal Firstenberg and Yinnon Meshi are my co-inventors of Archimedes, a weakly supervised method. Archimedes shed light on most of this work, directly and indirectly.

I worked in "Acumen" a start-up aiming to improve software development, perfectly matching my PhD goals. That was a wonderful opportunity to observe the software im-

provement field from the industrial provider point of view, being in touch with customers wanting to improve their development. I'd like to thank "Acumen" founders Nevo Alva, Daniel Shir, and Itamar Mula for founding it, and to all Acumen's employees for the common path and for emphasizing for me the differences and similarities between software engineering research in the academy and in the industry.

Eitan Frachtenberg, a partner to many discussions on programming, matched me with Dror and helped in the research. Amiram Yehudai and Shahar Maoz, my committee, were supportive and gave plenty of useful advice and feedback. This is a good place to thank our reviewers too. Though accepting their feedback required training, it was beneficial and improved our work, just as feedback improves code.

My family shaped me and helped in numerous ways. Starting with my parents, raising me in an environment of curiosity and a feeling that everything is possible. A special thank to my father, for being interested in every step of the research, discussing them, and suggesting wonderful insights and ideas. My wife and children tolerated my hours in the lab extension and more, helping and supporting me in this journey.

I'd like to thank Dror, my advisor, first for being an ideal of an intellectual. Dror demonstrated in every step curiosity driven research and setting a high bar in every aspect. I understood that the first time we edited together. Dror can edit a very long paper, focus on a single word and say "it is not optimal" and spend a long time finding a better phrasing. All that was done while Dror gave me the freedom to go and explore research directions that become more and more far from his own areas of interest. Thank you for this wonderful experience!

Bibliography

[1] I. Amit. Software development task effort estimation. U.S. patent application #US20220122025A1, 2020.

[2] I. Amit. End to end software engineering research. *arXiv preprint arXiv:2112.11858*, 2021.

[3] I. Amit, N. B. Ezra, and D. G. Feitelson. Follow your nose – which code smells are worth chasing? *arXiv preprint arXiv:2103.01861*, 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, page 12–15, New York, NY, USA, 2019. Association for Computing Machinery.

[5] I. Amit and D. G. Feitelson. Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal*, 29(4):817–861, Aug 2021.

[6] I. Amit, E. Firstenberg, and Y. Meshi. Framework for semi-supervised learning when no labeled data is given. U.S. patent #US11468358B2, 2017.

[7] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[8] 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.

[9] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp. Motivation in software engineering: A systematic literature review. *Information & Software Technology*, 50(9-10):860–878, 2008.

[10] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.

[11] B. Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.

[12] 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.

[13] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, Oct 1988.

[14] F. P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[15] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[16] O. Burn. Checkstyle. *http://checkstyle.sourceforge.net/*, 2007.

[17] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, Jun 1994.

[18] L. Choshen and I. Amit. Comsum: Commit messages summarization and meaning preservation. 2021.

[19] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[20] P. Crosby. *Quality Is Free: The Art of Making Quality Certain*. McGrawHill, 1979.

[21] 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.

[22] T. DeMarco and T. Lister. Programmer performance and the effects of the workplace. In *Proceedings of the 8th international conference on Software engineering*, pages 268–272, 1985.

[23] T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams*. Dorset House, 1987.

[24] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, mar 1968.

[25] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[26] S. A. Frangos. Motivated humans for reliable software products. In D. Gritzalis, editor, *Reliability, Quality and Safety of Software-Intensive Systems: IFIP TC5 WG5.4 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, pages 83–91. Springer US, Boston, MA, May 1997.

[27] C. França, F. Q. B. da Silva, and H. Sharp. Motivation and satisfaction of software engineers. *IEEE Transactions on Software Engineering*, 46(2):118–140, Feb 2020.

[28] J. E. Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, (4):459–464, 1984.

[29] J. E. Gaffney Jr and T. A. Durek. Software reuse—key to enhanced productivity: some quantitative models. *Information and Software Technology*, 31(5):258–267, 1989.

[30] P. A. Gagniuc. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[31] 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.

[32] Y. Gil and G. Lalouche. On the correlation between size and metric validity. *Empirical Softw. Eng.*, 22(5):2585–2611, Oct 2017.

[33] GitHub. Github activity data https://console.cloud.google.com/marketplace/product/github/githu repos, September 2019.

[34] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[35] 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.

[36] 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.

[37] 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.

[38] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[39] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, et al. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, 27(6):1–49, 2022.

[40] A. B. Hill. The environment and disease: association or causation?, 1965.

[41] International Organization for Standardization. Iec 9126-1 (2001). software engineering product quality-part 1: Quality model. page 16, 2001.

[42] International Organization for Standardization. Iso/iec 25010:2011. systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models, 2011.

[43] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, Inc., New York, NY, USA, 1991.

[44] C. Jones. Software quality in 2012: A survey of the state of the art, 2012. [Online; accessed 24-September-2018].

[45] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[46] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*, pages 295–315. Springer, 2003.

[47] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[48] 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.

[49] P. Lenberg, R. Feldt, and L. G. Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, 2015.

[50] D. Lewis. Causation. *The journal of philosophy*, 70(17):556–567, 1973.

[51] M. Lipow. Number of faults per line of code. *IEEE Transactions on software Engineering*, (4):437–439, 1982.

[52] R. C. Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall, 2002.

[53] J. D. Mayer, M. A. Faber, and X. Xu. Seventy-five years of motivation measures (1930–2005): A descriptive analysis. *Motivation and Emotion*, 31:83–103, 2007.

[54] T. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, SE-2(4):308–320, Dec 1976.

[55] 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.

[56] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22:3219–3253, 2017.

[57] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 580–586, 2005.

[58] L. Ottenstein. Predicting numbers of errors using software science. In *Proceedings of The 1981 ACM workshop/symposium on Measurement and Evaluation of Software Quality*, pages 157–167, 1981.

[59] J. Pearl. *Causality.* Cambridge university press, 2009.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] E. Raymond. The cathedral and the bazaar. *First Monday*, 3(3), 1998.

[65] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 341–350, New York, NY, USA, 2008. Association for Computing Machinery.

[66] 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.

[67] R. E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[68] H. Shen, J. Fang, and J. Zhao. Efindbugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 299–308, 2011.

[69] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497, 1976.

[70] S. A. Swanson, M. Miller, J. M. Robins, and M. A. Hernán. Definition and evaluation of the monotonicity condition for preference-based instruments. *Epidemiology (Cambridge, Mass.)*, 26(3):414, 2015.

[71] 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, Nov 2002.

[72] S. G. Vandenberg. Contributions of twin research to psychology. *Psychological Bulletin*, 66(5):327, 1966.

[73] V. Vapnik. *The nature of statistical learning theory.* Springer science & business media, 2013.

[74] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[75] R. S. Wahono. A systematic literature review of software defect prediction. *Journal of software engineering*, 1(1):1–16, 2015.

[76] G. Zieder, B. Kozorovitzky, O. Eliasaf, E. Egozi Levi, and O. Assulin. Software commit risk level. PCT #WO2015065367A1, 2013.

סקר של חקר מוטיבציה במשך שבעים וחמש שנים מצביע על כך שכלי המחקר העיקרי הוא שאלונים, הסובלים מבעיות אפשריות רבות.
התשובות סובייקטיביות ולא בהכרח קשורות להתנהגות בפועל.
השאלון רלוונטי לנקודת זמן בודדת.
עלות מציאת נשאלים מובילה להיקף מוגבל ולכן לאמינות ולרזולוציה מוגבלים.

בניגוד לשימוש בשאלונים, אנחנו רצינו לחקור מוטיבציה באמצעות שיטות מדעי המחשב --- בצורה כמותית, בעזרת התנהגות בפועל, ועל בסיס נתונים גדולים, המכסים פרק זמן ארוך.
לשם כך ערכנו סקר גדול, בו שאלנו מפתחים שאלות לגבי מוטיבציה אבל גם ביקשנו את פרופיל ה- GitHub שלהם.
הפרופיל איפשר לנו לקשר את התשובות להתנהגות בפועל.

בסקר חקרנו אחד עשר סוגי מוטיבציה מהספרות.
זיהינו בעיות תקפות כמו מתאם מתון בין תשובות לשאלות קשורות, מתאם מתון בתשובות לאותה שאלה בסקר ובסקר מעקב, וכן הערכה עצמית מוגזמת וטעויות בתשובות.
למרות זאת מצאנו שהסוגים השונים אכן מנבאים מוטיבציה גבוהה.
סקר המעקב אפשר לנו לגלות ששיפור ברוב הסוגים חוזה שיפור במוטיבציה.

כדי לחקור מוטיבציה בבסיס הנתונים ההתנהגותי של GitHub, אנחנו צריכים להבדיל בין מפתחים בעלי מוטיבציה לאחרים.
אנחנו מזהים מפתחים בעלי מוטיבציה בעזרת כמה פונקציות תיוג (labeling functions), כמו עבודה בשעות מגוונות וכתיבת הודעות commit ארוכות.
וידאנו שפונקציות התיוג הן weak classifiers למוטיבציה על ידי השוואתן לתשובות בסקר.
עשינו אימות נוסף באמצעות בסיס הנתונים, בו בדקנו את ההסכמה בין פונקציות התיוג באופן כללי ואת ההסכמה בניתוח co-change.

לאחר שאימתנו שהפונקציות הן weak classifiers למוטיבציה, השתמשנו בהן כדי לחקור ולכמת את הקשר בין מוטיבציה לביצועים.
ההבדל בביצועים בין מפתחים בעלי מוטיבציה וחסרי מוטיבציה עשוי להגיע למאות אחוזים.
פער זה גם מסביר את הפערים הגדולים בין מפתחים וגם רומז שמועיל מאוד להגביר את המוטיבציה.
בנינו גם מודל החוזה retention, השארות מפתחים בפרוייקט בשנה הבאה.
למודל היה כוח חיזוי גבוה יותר ממודל המשתמש רק ב control variables שלנו, מה שמצביע על כך שלפונקציות התיוג יש כוח חיזוי משלהן.
ביצועי המודל היו גבוהים, דבר ההופך את המודל לבעל ערך כשלעצמו.

המתודולוגיה שיצרנו במחקר זה גמישה וניתנת ליישום בתחומים רבים.
היא מספקת דרך למדוד מושגים ערטילאיים וחסרי הגדרה מדויקת, כמו איכות ומוטיבציה.
היא גם מאפשרת לנו למנף הגדרות שונות של אותו מושג, תוך פיצוי על המגבלות של כל הגדרה בודדת.
בהינתן הגדרות המושגים נוכל לחקור את היחסים ביניהם מעבר למתאם,
לייחס שינויים לאדם או לפרויקט,
ולחזות את השינוי במושג המטרה (לדוגמה, ביצועים) בהינתן שינוי במושג המקור (לדוגמה, מוטיבציה).

תקציר

איכות תוכנה הינה בעלת השפעה ניכרת ולכן אנו מעוניינים לשפרה.
על מנת לעשות זאת, ראשית עלינו להגדיר מהי איכות ולספק מדדים שמכמתים אותה.
תחילה חקרנו היבטים טכניים של פיתוח תוכנה.
גילינו פערי ביצועים גדולים בין פרויקטי תוכנה.
מצאנו שמאפייני קוד רבים משפיעים על הביצועים, אך באופן מתון בלבד.
הבחנו בפערים דומים לאלו שבפרוייקטים רק כאשר חקרנו מפתחים.
לכן, בחרנו להתמקד במפתחים ובמוטיבציה שלהם כגורם מוביל.
מצאנו המלצות להגברת המוטיבציה וכימתנו את השפעת המוטיבציה על הביצועים בפיתוח תוכנה.
כך אנו מספקים דרך לשפר ביצועים בתוכנה דרך שיפור המוטיבציה.

כעת נסביר את העבודה ביתר פירוט.
לאיכות קוד יש היבטים רבים, אך קשה לכמת אותה בשל מורכבותה ואופיה הסובייקטיבי (למשל, שביעות רצון המשתמש, מתן משקל ראוי להיבטים שונים).
לפעמים איכות הקוד מוערכת באופן עקיף באמצעות דפוסים אחרים המקובלים כמצביעים על איכות נמוכה, כמו Code Smells או Self Admitted Technical Debt.
מוסכם שגם באגים מעידים על איכות נמוכה, כפי שעולה מהמחקר הנרחב על חיזוי באגים.
באגים שימשו כאינדיקטור לאיכות בעבר, בווריאציות שונות.
אנו פיתחנו את מדד ה Corrective Commit Probability בו מספר התיקונים מנורמל במספר השינויים, וכך הוא אגנוסטי לגודל ושימושי כהסתברות.
ההבדל ב-CCP בין פרויקטים הוא מאות אחוזים.

במהלך חיפוש גורמים המסבירים את פערי האיכות בין פרוייקטים, חקרנו את ההשפעה של משוב (לדוגמה code review) ומבנה התוכנה (אורך קבצים, coupling).
לאחר מכן חקרנו אילו סוגי refactoring מפחיתים את ההסתברות לתיקוני באגים.
כמו כן חקרנו למעלה מ-170 סוגי התראות סטטיות.

מצאנו שהרבה best practices אינם מועילים, ושההשפעה של המועילים מביניהם הייתה מתונה בלבד.
כדי להעריך את ההשפעות האלו, אנו משתמשים ברדוקציה ל-supervised learning, המאפשרת לנו למנף את העבודה שנעשתה בתחום זה.
בנינו מודלים המשתמשים ב- co-change, חיזוי השינוי במדד המטרה לפי שינויים במשתני האובייקט.
השתמשנו גם במונוטוניות, controls, וניסויי תאומים על מנת להפחית את הסכנה שהתוצאות הינן בשל גורמים אחרים.

רק כאשר חקרנו מפתחים מצאנו פערי ביצועים גדולים, כמו בפרויקטים.
לפיכך, דרך פשוטה להגיע לביצועים טובים יותר היא על ידי מפתחים טובים יותר.
עם זאת, קשה למצוא מפתחים טובים.
אינדיקציה לכך היא רמת הביצועים הדומה של כלל המתכנתים, של בוגרי אוניברסיטאות יוקרתיות, ושל עובדי חברות יוקרתיות.
יחד עם זאת, ביצועים מושפעים מאוד ממוטיבציה.
לפיכך, השיטה שלנו לשפר את הביצועים עוברת דרך שיפור המוטיבציה.

עבודה זו נעשתה בהדרכתו של פרופ' דרור פייטלסון

שימוש בלמידה חישובית למתודולוגיה לשיפור פיתוח תוכנה

חיבור לשם קבלת תואר דוקטור לפילוסופיה

מאת

עידן עמית

הוגש לסנט האוניברסיטה העברית בירושלים

5/2024

שימוש בלמידה חישובית
למתודולוגיה לשיפור פיתוח תוכנה

חיבור לשם קבלת תואר דוקטור לפילוסופיה

מאת

עידן עמית

הוגש לסנט האוניברסיטה העברית בירושלים

5/2024