Aligning Offline Metrics and Human Judgments of Value for Code Generation Models

Victor Dibia[1], Adam Fourney[1], Gagan Bansal[1], Forough Poursabzi-Sangdeh[1], Han Liu[2], Saleema Amershi[1]
[1]Microsoft Research, Redmond, United States
{victordibia, adam.fourney, gaganbansal, fpoursabzi, samershi}@microsoft.com, hanliu@uchicago.edu
[2]University of Chicago, Chicago, United States

Abstract

Large language models have demonstrated great potential to assist programmers in generating code. For such human-AI pair programming scenarios, we empirically demonstrate that while generated code are most often evaluated in terms of their *functional correctness* (i.e., whether generations pass available unit tests), correctness does not fully capture (e.g., may underestimate) the *productivity* gains these models may provide. Through a user study with $N = 49$ experienced programmers, we show that while correctness captures high-value generations, programmers still rate code that fails unit tests as valuable if it reduces the overall effort needed to complete a coding task. Finally, we propose a hybrid metric that combines functional correctness and syntactic similarity and show that it achieves a 14% stronger correlation with value and can therefore better represent real-world gains when evaluating and comparing models.

1 Introduction

Large language models trained on code (e.g., Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), CodeGen (Nijkamp et al., 2022), InCoder (Fried et al., 2022)) have shown impressive capabilities on code generation tasks. One important application for such models is *Human-AI pair programming*, where a model suggests in-line code completions (e.g., within an IDE) that programmers can choose to ignore, accept, or edit as needed. Early studies suggest that this paradigm may dramatically boost productivity and transform the practice of software development (Ziegler et al., 2022; Kalliamvakou, 2022).

As is common with model development more generally, code-generation advances are largely driven by comparing model performance on *offline metrics* (i.e., metrics computed automatically over held out evaluation data) that can be easily tracked on leaderboards. *Functional correctness* metrics

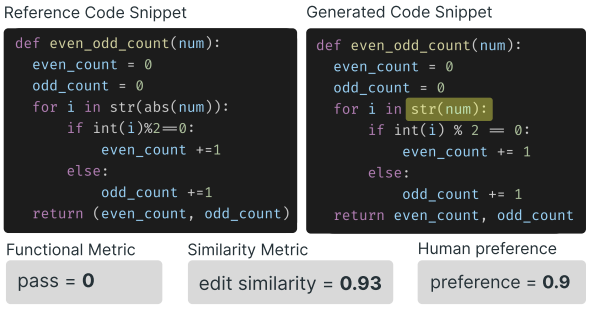

Figure 1: In the example above (counting even and odd numbers), code suggested by a model fails unit tests but is deemed useful by programmers because adding a short check (*abs* value) fixes the generation.

such as *pass@k* (Chen et al., 2021) currently represent the state-of-best-practice (Chen et al., 2021; Fried et al., 2022; Austin et al., 2021; Chowdhery et al., 2022; Nijkamp et al., 2022; Hendrycks et al., 2021; Kulal et al., 2019). These metrics evaluate generations by executing a set of unit tests and assessing whether the generations pass or fail. While functional correctness is clearly important, it does not fully capture the productivity gains programmers may value about code generation assistance. For example, a generation that fails unit tests might yet provide critical hints to solve a task (see example in Fig 1), or serves as boilerplate that can be adapted with minimal effort. Likewise, functionally correct code might be difficult to read or maintain, or may contain other vulnerabilities.

With developer productivity in mind (Forsgren et al., 2021), we investigate syntactic similarity-based offline performance metrics (e.g., (Svyatkovskiy et al., 2020; Chowdhery et al., 2022; Papineni et al., 2002)) as proxies of programmer effort needed to modify or correct automatic code generations. Similarity-based metrics compute how similar a generation is to reference or ground truth code, typically available in the offline setting. We then conducted a user study with N=49 experienced programmers to assess how well self-reported utility

Findings of the Association for Computational Linguistics: ACL 2023, pages 8516–8528
July 9-14, 2023

(Forsgren et al., 2021) correlates with similarity-based and functional correctness metrics. Our work answers the following key research questions:

1. Do programmers still value code generations that may be incorrect (fail unit tests)?
2. How well do existing offline performance metrics align with programmer-rated value, accuracy and effort?
3. Does a metric that captures both functional correctness and effort saved better align with programmers' perceived value?

In our studies, we showed participants code generated by AI models and asked them to provide ratings in terms of the accuracy of the code, overall value of the code and effort associated with fixing the code (if any). We find that while ratings on effort and accuracy both correlate with value, effort is significantly more correlated. In other words, code that is perceived as easy-to-fix is judged to be more valuable. Conversely, when considering offline metrics, we find that while functional correctness metrics are more correlated to value compared to similarity based metrics, similarity based metrics offer complementary information. Specifically, we find 42% of generations that failed unit tests were still rated as valuable - and similarity based metrics provide a better signal as to value in this regime. We therefore propose a metric that combines functional correctness and similarity and show that it increases correlation with perceived value by 14%.

2 Related Work

Offline performance evaluation of AI models typically consists of running models as isolated components over benchmark datasets and then computing aggregate *metrics* (e.g., accuracy, AUC, and precision/recall) that can be easily compared and tracked on leaderboards. While these evaluation practices have led to rapid advancements in AI by enabling efficient apples-to-apples model comparison, a growing body of work has raised concerns about the mismatch between popular metrics and what people need and value in the real world (Thomas and Uminsky, 2022; Raji et al., 2022; Hellendoorn et al., 2019; Hand, 2006; Jacobs and Wallach, 2021; Chandar et al., 2020; Zhou et al., 2022). Using metrics that fail to appropriately capture what people value can result in deploying models that are at best less effective than they could be, and at worst harmful to people and society (Thomas and Uminsky, 2022; Raji et al., 2022; Hand, 2006).

In this work, we investigate the extent to which common offline code generation metrics capture what professional programmers value about code generation models. In particular, we examine how well existing code generation metrics capture notions of developer effort and productivity Forsgren et al. (2021).

The current most popular family of code generation metrics is based on measuring *functional correctness*. Functional correctness metrics seek to evaluate generated code against known objective properties such as passing unit tests (Chen et al., 2021; Austin et al., 2021; Li et al., 2022; Roziere et al., 2020). Following the release of Codex and the HumanEval dataset (Chen et al., 2021)—which is a dataset of 164 hand-written problems in python with associated unit tests—the functional correctness metric of $pass@k$ (where k code samples are generated per problem and a problem is considered solved if any of the k generations passes the corresponding unit tests) has emerged as the dominant method for evaluating code generation models (e.g., (Fried et al., 2022; Xu et al., 2022; Li et al., 2022; Austin et al., 2021)). Advocates of functional correctness metrics argue for their resemblance to programming best practices (e.g, test-driven development) and fidelity to capturing functional behaviour (Chen et al., 2021). However, in this work we demonstrate that functional correctness does not fully capture what people value about code generation models.

Similarity-based metrics compare tokens from generated code to tokens of known solutions, with code that is more similar to given solution(s) being considered better. Multiple similarity-based metrics have been proposed for evaluating code generation models including exact match (Lu et al., 2021), edit distance (Svyatkovskiy et al., 2020; Chowdhery et al., 2022), BLEU (Papineni et al., 2002), CodeBLEU (Ren et al., 2020), and ROGUE (Lin, 2004). Analyses of similarity-based metrics and other measures of code quality have been mixed (e.g., (Ren et al., 2020) vs Austin et al. (2021)). However, in most of these cases, similarity was considered a proxy for functional correctness. In this work, we revisit similarity-based metrics as proxies for effort saved in coding tasks Svyatkovskiy et al. (2020) and demonstrate how they can be used to better capture value.

In this work we focus on $pass@k$ as a proxy for functional correctness and we experiment with

two similarity-based metrics, namely, normalized edit similarity (Lu et al., 2021; Svyatkovskiy et al., 2020; Chowdhery et al., 2022) (which measures how many single-character edits— including insertion, substitution, or deletion—are required to convert generated code to some reference code) and BLEU (which measures the token overlap between the generated and reference text) to investigate how these metrics approximate different facets of what programmers value in practice.

3 User Study

We designed a user study to evaluate how well functional correctness- and similarity-based offline metrics approximate value of code generations for programmers. The study showed experienced programmers various programming tasks, together with code generations and reference solutions. Programmers then rated the generations on perceived accuracy, effort, and value.

3.1 Dataset for Programming Tasks

We selected programming tasks from the HumanEval dataset (Chen et al., 2021), which consists of 164 hand-crafted programming tasks and solutions written in Python. Each task includes a task description (i.e., a function header followed by a comment describing the task with some sample test cases (115 – 1360 characters)), a canonical handwritten solution, and a set of associated unit tests. HumanEval has been extensively used to evaluate code generation systems (e.g., (Chen et al., 2021; Fried et al., 2022; Xu et al., 2022; Chowdhery et al., 2022; Nijkamp et al., 2022)). To the best of our knowledge, HumanEval is not part of any model's training data, and its simple standalone tasks makes it an ideal choice for user studies.

3.2 Offline Metrics for Code Generation

We experimented with three offline metrics, one of which served as a proxy for functional correctness and the other two served as a proxy for a programmer's effort.

PASS: As a proxy for functional correctness, we computed the $pass@k$ metric (Chen et al., 2021). $pass@k$ takes k generations for a problem and considers the problem solved if any generation passes the accompanying unit tests (in our case the unit tests provided in the HumanEval dataset). While related work has presented $pass@k$ results for values of k including 1, 10, and even up to 1M (Chen et al., 2021; Li et al., 2022), we focus our analysis on $k = 1$ which most closely resembles the real-world scenario where a programmer sees a single generation inline within a coding editor.

EDIT-SIM: As one proxy for effort, we computed normalized edit similarity (Svyatkovskiy et al., 2020) as follows:

$$\text{EDIT-SIM} = 1 - \frac{lev(gen, ref)}{max(len(gen), len(ref))}$$

where gen is code generated by a model for a problem in the HumanEval dataset, ref is the handwritten reference solution to the problem and lev is the character Levenshtein edit distance.

BLEU: As another proxy for effort, we computed BLEU using the formulation introduced by Papineni et al. (2002) (generated code compared with a single reference), and based on the implementation in the Tensorflow library (Abadi et al., 2015).

We focused on syntactic similarity-based metrics like EDIT-SIM Lu et al. (2021); Svyatkovskiy et al. (2020); Chowdhery et al. (2022) and BLEU Barone and Sennrich (2017); Karaivanov et al. (2014); Nguyen et al. (2013); Ahmad et al. (2021); Wang et al. (2021) because they have been commonly-used metrics for evaluating text-based generative models, especially, for code generation scenarios.

3.3 Code Generation Models

We selected 5 publicly available autoregressive large language models trained on code, varied mostly by the parameter size of each model. The first two models are variants of the CodeGen model introduced by Nijkamp et al. (2022) (CodeGen350 Multi, CodeGen2B Multi) - autoregressive transformers with the regular next-token prediction language modeling as the learning objective trained on a natural language corpus and programming language (C, C++, Go, Java, JavaScript, and Python) data curated from GitHub. Next, we use three publicly available variants of the Codex model (Chen et al., 2021), a GPT language model fine-tuned on publicly available code from GitHub (Cushman, Davinci1, Davinci2). Note that the goal of this work is to compare code-generation *metrics* and not to assess the performance of models. We used models of different sizes to help ensure our findings on how metrics behave translate across a range of model qualities. Following guidance from Chen

et al. (2021) who demonstrate the importance of optimizing sampling temperature for particular values of k, we used a low temperature value of $t = 0.2$ for $k = 1$ so that each model generates the most likely code tokens.

3.4 Tasks

We used programming tasks from the HumanEval dataset, where for each task, participants were shown the task description (function header and docstring describing the task along with sample test cases), the corresponding unit tests, and two code snippets — the reference solution from the HumanEval dataset and a generation for that task from one of the models — shown in a random order. Each snippet was randomly assigned a name - *Code Snippet A* or *Code Snippet B* for easy reference in the subsequent questions. All parts of the interface showing code were syntax highlighted to improve readability.

For each task, participants answered questions designed to collect their judgements along three dimensions of interest: overall value, accuracy, and effort which we hypothesized would impact value. Each question used 5-point Likert scales and were shown sequentially only after the previous question had been answered. The questions were as follows:

ACCURACY: The first question asked participants to judge whether both snippets were functionally equivalent. Since the reference solution is correct, functional equivalence can be used to infer perceived accuracy of a generation (complete equivalence indicates the participant believes the generation would produce the same outputs for all the same inputs as the reference solution which passes the provided unit tests). We used this equivalence question-based approach to assess perceived accuracy because our pilots suggested that judging equivalence is easier than solving the coding task from scratch, and also because it enabled us to design a simpler, consistent survey – the other two survey questions (as described next) also compared the generation to the reference.

At this point in the task, participants were not told which snippet corresponded to the generation and which was written by a human programmer to minimize the impact of any existing biases about the capabilities of AI models.

VALUE: Once participants advanced to the second question, the interface disclosed which snippet (A or B) was AI generated and which was a reference solution. They were then asked how useful the generated snippet would be assuming they were a programmer attempting to solve the task themselves. We described usefulness in terms of whether participants believed the generation provided a useful starting point, ranging from Extremely useful (they "would definitely accept it and use it as a starting point") to Not at all useful (they "would not even want to see the generation" let alone accept and use it as a starting point).

EFFORT: The final question asked participants how much effort they believed it would require them to modify the AI generated solution into a correct solution similar to the snippet written by a human programmer, if any.

3.5 Study Protocol and Participants

The study consisted of four sections: consent form, instructions, main study, and a brief post-study feedback section. The instructions section was a sample task designed to familiarize participants with the mechanics of the study interface (e.g., they will be shown problems and asked to provide ratings, they will not be allowed to go back and revise previous responses) and to anchor them to pair programming scenario.

The main study was made up of 12 tasks. We chose 12 because our pilot studies showed that participants could complete 12 tasks within an hour. For each task, participants were shown a generation from one randomly chosen model from our set of 5 models.

A key goal of our study was to assess how well our offline metrics of interest align with what programmers value. We were particularly interested in understanding the tradeoffs between functional correctness and similarity as they relate to value and so we wanted to probe cases where these metrics disagreed. Therefore, to select study tasks, we first considered taking a random sample from HumanEval. However, the number of generations falling into regions where these metrics agreed on the largest model (Davinci2) was over-represented compared to the disagreement region (70% agreement vs 30% disagreement). Therefore, we chose a stratified sampling method where we first assigned each HumanEval problem into one of three buckets: PASS = 1 and EDIT-SIM is low, PASS = 0 and EDIT-SIM is high, PASS and EDIT-SIM agree .[1] Then,

[1]According to Davinci2 and where similarity was thresholded along the median similarity value for that model.

we sampled equally across each bucket aiming to annotate 20 problems per bucket for this study.

Because we intended to recruit professional programmers, we aimed to obtain up to 2 annotations per problem-model pair. With 60 problems (20 per bucket), 5 models, and 2 annotations per task and a budget of 12 problems per participant, this required us to recruit 50 participants for this study. We assigned annotation tasks to participants by randomly sampling a problem from our sample of 60 and then randomly sampling a generation for that problem, without repeating a problem for any participant, until each problem-model pair was assigned 2 annotations.

We recruited professional programmers from a large technology company for this study and recruitment emails were sent out to a randomly sampled subset of software engineers. Participants were required to have at least 1-2 years of programming experience with Python and to program in Python at least a few times a year. 61% of respondents indicated they had worked on a python project in the last month and 59% had never used a pair programming AI assistant like GitHub Copilot.

The study was deployed as a web application. Participants were given five days to complete the study, and could pause and resume using their personalized study link. At the end of the study, participants were given a $50 online gift card. As an additional incentive, we awarded the top 5 performers an additional $50 gift card. We determined top performers based on the rate at which participants correctly indicated a generation was equivalent to the reference code when it passed vs when it failed the given unit tests. This experiment was approved by our organization's internal IRB process.

4 Study Results

At the end of the study period, we obtained responses from 49 participants. We then applied the following criteria to evaluate the quality of responses: First, we computed the median response time per task for all participants and also computed a performance rating on the code equivalence task in the same way we determined top performers in our study. Data from three participants who fell within the bottom 10th percentile of the median task completion times and their performance was worse than the probability of random chance (given the questions they responded to) was excluded from the data analysis. The final dataset includes data from 46 participants with 552 annotations across 290 unique tasks and 1.96 annotation per task. Finally, across tasks where we obtained multiple annotations, we verified that there was agreement between annotators[2] and then computed the mean annotation per task for use in our analysis. In this section, we present the main findings based on this data.

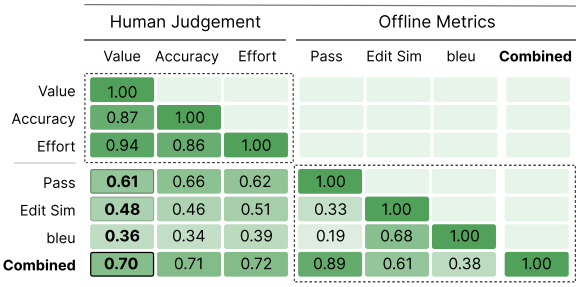

Figure 2: Correlation (Pearson) between human judgements (perceived value, accuracy and effort) and offline metrics (functional correctness, edit similarity and a combined metric (see section 4.4)). All correlations are significant with $p < 0.001$.

4.1 Accuracy is Valuable, but Effort Matters

Our first finding is that the VALUE of a generation is nearly perfectly correlated with the perceived **EFFORT** needed to correct a generation (Pearson $r = 0.94$; 95%-confidence interval $[0.92 - 0.95]$). Recall that **EFFORT** is reverse-coded such that a score of 5 indicates "no effort" is needed. **ACCURACY** is also highly correlated (Pearson $r = 0.87$; 95%-confidence interval $[0.84 - 0.90]$), but significantly less so – we note that their confidence intervals do not overlap. **From this we conclude that ACCURACY isn't everything, and EFFORT is at least as important a signal for capturing VALUE.** We will return to this theme throughout the paper. Correlations between these dimensions are presented in the top-left quadrant of Figure 2.

4.2 Offline Metrics Highly Correlate with Programmers' Judgements, But There is Room for Improvement

Our second finding confirms **that the metrics used in practice (PASS, EDIT-SIM, and BLEU) are indeed positively correlated with VALUE**, but there are important differences (Fig. 2, bottom-left quadrant). As an example, PASS shows the strongest

[2] In 50% of cases, annotators are in perfect agreement; 75% differ by at most one point in valence (on a rating scale of 1-5) and the mean difference is 0.89.

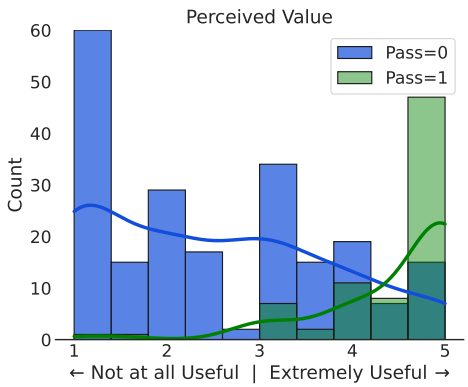

Figure 3: Distributions of VALUE judgments, in cases where generations pass unit tests (PASS = 1, green), and fail unit tests (PASS =0, blue). When generations pass unit tests, they are likely to be judged as valuable (VALUE $\geq$ 3). In fact, only 3% of generations fall below this score. When functions fail unit tests, they are somewhat more likely to have lower VALUE scores, but 90 generations (42%) are still considered "somewhat useful" by participants. The metric misses many high-value generations.

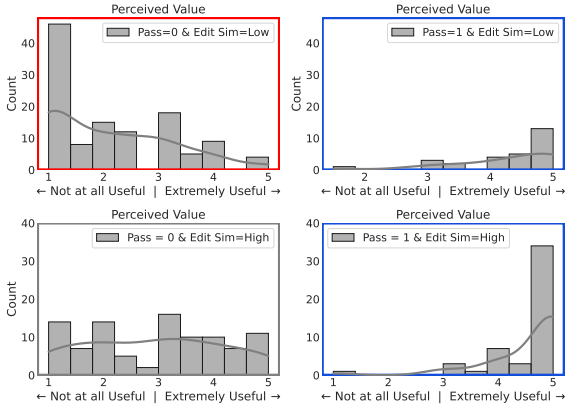

Figure 4: Distributions of VALUE judgments, in the four possible combinations of PASS outcomes and low/high EDIT-SIM scores. The top and bottom rows indicate cases where EDIT-SIM falls below and above the 50th-percentile, respectively. The left and right columns indicate cases where PASS = 0 and PASS = 1 respectively. When PASS = 1, generations are likely be high value (blue regions). When both PASS = 0 and EDIT-SIM =low, generations are likely to be low value (red region). The VALUE is more uniformly distributed in the remaining region where both PASS = 0 and EDIT-SIM =high.

association with **ACCURACY** of the three metrics ($r = 0.66; p < 0.001$). This is unsurprising, given that PASS is a direct measure of functional correctness. More surprising to us, however, is that PASS is also the most strongly correlated metric to both **EFFORT** and VALUE ($r = 0.62; p < 0.001$, and $r = 0.62; p < 0.001$ respectively). This was unexpected since EDIT-SIM is a direct measure of the number of changes needed to correct a suggestion, and yet shows weaker correlation to **EFFORT** ($r = 0.48; p < 0.001$). With a correlation of $r = 0.36; p < 0.001$, **BLEU under-performs all other metrics**. Finally, given that none of the metrics correlate better than $r = 0.66$, there is significant opportunity to develop improved metrics.

4.3 Code That Passes Unit Tests (PASS = 1) is Extremely Likely to be High-Value

Our third finding is that **when PASS = 1 (i.e., when generations pass unit tests), we can be reasonably certain that they will be of high VALUE** (Figure 3). In fact, only 2 of 77 (3%) generations that passed unit tests were found to have a VALUE scores less than 3. Recall, a VALUE score of 3 indicates that the participant found a suggestion to be at least "somewhat useful."

However, PASS = 0 is less good at filtering low-value suggestions; Only 123 of the 213 (58%) generations that failed unit tests scored lower than 3 on value. Stated differently, 90 generations (42%) were found to be at least somewhat valuable. This finding confirms existing qualitative work that while programmers value functionally correct code, they may still find considerable value in code that is not functionally correct (Weisz et al., 2021).

4.4 Improved Metrics Through Combination

Upon further inspection, we realized that EDIT-SIM was itself a useful lens through which to understand how VALUE is distributed when unit tests are failed. Figure 4 shows a partitioning of results such that left and right columns correspond to (PASS = 0) and (PASS = 1) respectively. Top and bottom rows correspond to cases where the EDIT-SIM is below and above the 50th-percentile respectively (referred to as EDIT-SIM =Low and EDIT-SIM =High). As before, we find that when (PASS = 1), the VALUE tends to be high (blue outlined regions). However, we also find that **when a generation both fails the unit test and has low EDIT-SIM (i.e., PASS = 0; EDIT-SIM = *low*), it tends to be judged to have low VALUE** (red outlined region). Conversely, in the final region (PASS = 0; EDIT-SIM = *high*), VALUE is distributed more uniformly, and the signal is less clear. This strongly suggests that if human labeling is limited by budget, it may be worthwhile oversampling this

region to maximally recover some of the missing VALUE signal.

This also suggests that there is an opportunity to combine metrics because PASS $= 1$ is good at spotting high-value generations, while PASS $= 0$; EDIT-SIM $= high$ is good at spotting low-value generations. To investigate this further, we formally define a simple combined metric as follows:

$$\textbf{COMBINED} = min(1.0, \text{PASS} + \text{EDIT-SIM})$$

Figure 2, row 7, shows some merit to this approach: The combined metric correlates better with human judgments of value ($r = 0.70; p < 0.001$) than PASS ($r = 0.61; p < 0.001$) and EDIT-SIM ($r = 0.48; p < 0.001$) for EDIT-SIM. This is an extremely promising result, but was also only our first attempt at combining metrics. Future work is needed to explore other potential combinations.

5 Discussion & Future work

5.1 What Do Programmers Value?

Much of the current research evaluating code generation models aims to approximate overall value via some notion of correctness (Chen et al., 2021; Fried et al., 2022; Austin et al., 2021; Chowdhery et al., 2022; Nijkamp et al., 2022; Hendrycks et al., 2021; Kulal et al., 2019). Even research exploring similarity-based metrics have tended to validate these against some general notion of code quality (e.g., Mathur et al. (2020) consider "adequacy" while Ren et al. (2020) consider "good vs bad"). In this work, we aim to tease out distinct aspects of value to better understand how they contribute what programmers want from their AI-pair programmers. In this study, we examine the impact of correctness and effort. Our findings show that effort indeed matters to programmers. Accuracy also matters but, interestingly, our findings suggest that effort savings may be even more valuable to programmers than accuracy.

In general, we take the position that value is a multidimensional theoretical construct (Thomas and Uminsky, 2022). As such, while our findings showed effort as more valuable to programmers than accuracy, because both are still highly correlated with value, we recommend considering both when assessing the impact of human-AI pair programming. Moreover, there are likely many other properties of AI-pair programmers that developers find valuable (Forsgren et al., 2021) and future work warrants investigating how these may also be captured in offline evaluations.

5.2 How can Developers Approximate Value?

Our results show that when developers have access to evaluation data containing high quality unit tests (as in HumanEval), generations that pass unit tests are highly likely to be valuable to programmers. This suggests that PASS could be used as a reasonable filter in high precision scenarios (e.g., if an AI-pair programmer was tuned to limit distractions by only showing generations when they most likely to be valuable).

That said, however, PASS alone may miss a significant fraction of generations that programmers might find valuable. Our findings show that another offline metric – EDIT-SIM can help overcome this issue when we combine it with PASS according to Equation 4.4. This new metric is similar in spirit to *hinge-loss* in support vector machines.[3] In that setting, misclassifications are penalized based on their distance to the hyperplane decision boundary. Conversely, correct classifications all receive a fixed loss of 0, following the intuition that they don't become *more correct* the further they move from the hyperplane. In our setting, we expect VALUE to increase as generations become more similar to a reference solution, but once it reaches functional correctness it doesn't become *more correct* the closer it gets (syntactically) to the reference solution.

We emphasize, however, that metrics can have varying implications on model development decisions and therefore the choice of when or if to combine them is important. For example, when developers are seeking to make deployment decisions between models, selecting models that rank highest in terms of the overall value they may provide to programmers seems reasonable. In this case, the theoretical construct being approximated is perceived VALUE and our **COMBINED** metric is better at estimating this than PASS or EDIT-SIM alone. However, when developers are diagnosing issues during model development (e.g., via error analyses) *we recommend that* PASS *and* EDIT-SIM *be applied independently to get a clearer picture of model behavior* (Thomas and Uminsky, 2022) and to ensure appropriate mitigation strategies are used for different issues. For example, PASS failing on

[3] https://en.wikipdia.org/wiki/Hinge_loss

certain types of problems (e.g., recursive problems) or code blocks (e.g., conditional statements, error handling) may suggest additional data is needed in fine tuning. Whereas, EDIT-SIM failures may warrant new user interface techniques to help programmers focus attention to parts of the code most likely needing edits.

5.3 Approximating Accuracy and Effort

Our results show that programmers value both accuracy and effort savings when it comes to their AI pair programmers. We demonstrate that PASS is a reasonable proxy for accuracy. Surprisingly, however, we found that EDIT-SIM is only moderately correlated with effort and in fact is less correlated with effort than PASS. This is somewhat counter-intuitive since EDIT-SIM directly measures the number of characters that need to be changed to convert a generation to the reference solution (Svyatkovskiy et al., 2020; Lu et al., 2021).

This, along with our finding that programmers value effort reduction from their AI pair-programmers, suggests that an important area for future work is to experiment with alternative ways to operationalize effort for offline evaluation. This also, emphasizes the importance of validating that metrics faithfully capture the theoretical constructs they are trying to measure (Jacobs and Wallach, 2021).

5.4 When Should Developers Use EDIT-SIM?

Our findings show that EDIT-SIM is moderately correlated with PASS. This is important because there are many situations where computing PASS may be undesirable. For example, PASS requires executing arbitrary generated code which can be resource intensive and may pose security risks (Chen et al., 2021). PASS and other functional evaluation metrics also require the availability of comprehensive, high-quality unit tests as well as language-specific test infrastructure, assumptions which may not hold in some evaluation scenarios (e.g., testing functions in the wild). Therefore, *while we recommend* PASS *when it is appropriate because it is more strongly correlated with value than* EDIT-SIM*, our findings suggest that* EDIT-SIM *may be a reasonable alternative when it is desirable to avoid limitations of functional evaluation*.

Of course, limitations of similarity metrics should also be weighed against their benefits. For example, similarity metrics can fail when tasks have multiple syntactically divergent solutions - e.g. an algorithm may have an iterative vs recursive implementation with low token overlap, leading to noisy similarity metric. However, we intuit that this scenario is relatively infrequent given the structured nature of programming languages and existing research on developer behaviour e.g., Allamanis et al. (2018) who mention that *developers prefer to write (Allamanis et al., 2014) and read code (Hellendoorn et al., 2015) that is conventional, idiomatic, and familiar, because it helps in understanding and maintaining software systems*. A convergence towards idiomatic solutions make it more likely the solutions and patterns learned by large language models of code coincide with ground truth solutions, limiting the scenario where generated code is syntactically different from but functionally equivalent to ground truth.

6 Conclusion

We studied how well two types of offline metrics for evaluating code generation models (i.e., functional correctness such as $pass@k$ based on unit tests and similarity-based metrics such as edit similarity) align with human judgements of value when used for human-AI pair programming. Our user study with 49 experienced programmers suggests that while programmers find functionally correct code generations valuable, the effort to edit and adapt generations also matters. Existing offline metrics show high correlation with human judgements of value, but there is room for improvement. One reason is that while code that passes unit tests is very likely to be rated high-value, code that fails unit tests is often still considered valuable by programmers. Based on this observation, we propose a combined offline metric inspired by hinge-loss in support vector machines that allows for partial credit by combining strengths of functional correctness and similarity-based metrics. Our analysis shows that this combined metric aligns better with human judgements of value in code generations than functional correctness or similarity alone. Overall our work highlights the importance of validating that offline metrics in AI capture what people value and that human-centered metrics, inspired by what people value, can provide better estimates of what people want from their AI-pair programmers.

Limitations

In this work, we focused on problems posed in the hand-crafted HumanEval dataset (Chen et al., 2021). A potential pitfall of a curated dataset such as HumanEval is that the results *may* not generalize to real-world scenarios where developers often deal with more complex problems and code bases (e.g, code with multiple dependencies across multiple files). To address this limitation, we originally explored the use of datasets mined from GitHub. However, our experiments indicated memorization issues (e.g., verbatim generation of solutions to niche problem), potentially due to the sample code already being included in the model training set(Lee et al., 2021). In practice, high quality code deduplication required to avoid this specific limitation is challenging. Work by Allamanis (2019) find that the impact of duplicate code can be severe, sometimes inflating model performance scores by up to 100%. Furthermore, in our early pilot tests, functions extracted in the wild were found to contain insufficient context (e.g. absence of docstring) for even expert human annotators and isolating functional tests is challenging without heavy curation. Further research is therefore needed to understand how our findings might generalize to a wider variety of deployment settings as well as research on designing diverse evaluation datasets. In addition, future work may also explore the impact of problem difficulty on the observed results in our study.

Ethics Statement

While our study informs current practices in evaluating code generation models, we acknowledge that measures of value can differ across multiple demographics with impact on productivity. For our experiments (approved by an internal IRB board), we generate code snippets based on a publicly available dataset, using publicly available models that are annotated by experienced developers. These choices make our work readily reproducible. We also developed a library that implements multiple metrics for bench marking code generation models which we will make available as an open source library (MIT license) at the time of publication.

References

Martín Abadi, Ashish Agarwal, and Paul Barham et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Miltiadis Allamanis. 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*, pages 143–153.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

Praveen Chandar, Fernando Diaz, and Brian St. Thomas. 2020. Beyond accuracy: Grounding evaluation metrics for human-machine learning systems. In *Advances in Neural Information Processing Systems*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Tom Zimmermann, Brian Houck, and Jenna Butler. 2021. The space of developer productivity: There's more to it than you think. *ACM Queue*, 19(1):1–29.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

David J. Hand. 2006. Classifier technology and the illusion of progress. *Statistical Science*, 21(1).

Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this? evaluating code contributions with language models. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 157–167. IEEE.

Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 960–970.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Abigail Z. Jacobs and Hanna Wallach. 2021. Measurement and fairness. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. ACM.

Eirini Kalliamvakou. 2022. Quantifying GitHub Copilot's impact on developer productivity and happiness. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2021. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Nitika Mathur, Timothy Baldwin, and Trevor Cohn. 2020. Tangled up in BLEU: Reevaluating the evaluation of automatic machine translation evaluation metrics. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4984–4997, Online. Association for Computational Linguistics.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Inioluwa Deborah Raji, I. Elizabeth Kumar, Aaron Horowitz, and Andrew Selbst. 2022. The fallacy of AI functionality. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. ACM.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443.

Rachel L. Thomas and David Uminsky. 2022. Reliance on metrics is a fundamental challenge for ai. *Patterns*, 3(5):100476.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? human-ai partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*, pages 402–412.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*.

Kaitlyn Zhou, Su Lin Blodgett, Adam Trischler, Hal Daumé III, Kaheer Suleman, and Alexandra Olteanu. 2022. Deconstructing nlg evaluation: Evaluation practices, assumptions, and their implications. *arXiv preprint arXiv:2205.06828*.

Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
Section 7: Limitation

☑ A2. Did you discuss any potential risks of your work?
Section 8: Ethics statement.

☑ A3. Do the abstract and introduction summarize the paper's main claims?
Section 1: Introduction

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

Section 3.3. We use 5 code generation models for our experiments. Two models are open source models available on HuggingFace (CodeGen350 multi, CodeGen2B multi) and 3 models from OpenAI (Codex Cushman, Davinci001, Davinci002)

☑ B1. Did you cite the creators of artifacts you used?
Section 3.3

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
In section 3.3. we point the reader to the source of models used in the experiment. In the ethics section we also mention we will be releasing a library (api and user interface) which we used generating code snippets used in our human evaluation study.

☑ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Section 3.3. The models we use were created specifically for the task of code generation. In Section 2, we mention how the use of similarity based metrics for evaluation text generative models have been mixed, but contextualize our use of this metric as a surrogate for effort associated with fixing generated code.

☒ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Our work explores a specific domain (code generation) that does not usually cover names or offensive content.

☑ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
In Section 3.3., we provide documentation on the models used. In Section 3.1, we provide details on the dataset used.

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

In Section 3.3., we provide documentation on the models used. In Section 3.1, we provide details on the dataset used. In Figure 2 we provide information on the correlation between metrics and report that they are statistically significant with $p < 0.001$. In section 4.1, our statistics on correlation between perceived value, effort and accuracy is reported with confidence intervals.

C ☑ Did you run computational experiments?

Section 3.3

☐ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
No response.

☒ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
We used pretrained models without any modification. In section 3.3 we point the reader to the exact models used to enable reproducibility.

☑ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
Correlations reported in this paper specify significance and confidence interval as needed. See Figure 2 and section 4.1

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
In section 3.2, we mention the details of our implementation of relative edit similarity and BLEU. In section 3.3 we also point the reader to details on the models used in our study.

D ☑ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Section 3.5, 3.5

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
No response.

☑ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Section 3.5

☑ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Section 3.5

☑ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Section 3.5

☑ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Section 3.5