StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code

Hannah McLean Babe
Oberlin College

Sydney Nguyen
Wellesley College

Yangtian Zi
Northeastern University

Arjun Guha
Northeastern University and Roblox

Molly Q Feldman
Oberlin College

Carolyn Jane Anderson
Wellesley College

Abstract

Code LLMs have the potential to make it easier for non-experts to understand and write code. However, current CodeLLM benchmarks rely on a single expert-written prompt per problem, making it hard to generalize their success to non-expert users. In this paper, we present a new natural-language-to-code benchmark of prompts written by a key population of non-experts: beginning programmers. StudentEval contains 1,749 prompts written by 80 students who have only completed one introductory Python course. StudentEval contains numerous non-expert prompts describing the same problem, enabling exploration of key factors in prompt success. We use StudentEval to evaluate 12 Code LLMs and find that StudentEval is a better discriminator of model performance than existing benchmarks. Our analysis of student prompting strategies reveals that nondeterministic LLM sampling can mislead students about the quality of their descriptions, a finding with key implications for Code LLMs in education.

1 Introduction

Large language models trained on code (Code LLMs) have the potential to democratize programming by enabling less-experienced programmers to write code in natural language. A growing body of work shows their utility to professional programmers (Vaithilingam et al., 2022; Ziegler et al., 2022; Barke et al., 2023), but to broaden the accessibility of programming, models must also work well for non-experts. Code LLM benchmarks (Kulal et al., 2019; Hendrycks et al., 2021; Chen et al., 2021; Austin et al., 2021; Lai et al., 2023) are largely modelled after experts, both in the choice of task, and by having professionals write benchmark prompts. Achieving good performance on these benchmarks indicates that a model will perform well *if the user can write prompts equally as well as the expert*.

Our goal is to facilitate research on how to better align Code LLMs with non-expert programmers, who may talk about code differently than experts. Towards this goal, we present StudentEval, a dataset with 1,749 prompts written by beginning CS students and validated with expert-written test cases. Prompts were collected using 48 beginner-appropriate problems, with numerous different prompts for each problem. Our prompts exhibit the variation in technical vocabulary and lack of familiarity with how to describe code that are common with non-experts.

While other work explores Code LLM use in classrooms (Leinonen et al., 2023; Kazemitabaar et al., 2023; Prather et al., 2023), StudentEval is the first benchmark based on student interactions. It differs from existing benchmarks in three key ways: **1)** Other benchmarks have prompts authored by experienced programmers, whereas StudentEval has *prompts authored by students who have only completed one computer science course*. **2)** Other benchmarks contain tricky problems designed to stress-test the problem solving capabilities of Code LLMs. In contrast, StudentEval has problems that are *easily solved with expert descriptions, but often fail with student descriptions*. **3)** Other benchmarks only have a single prompt per problem, whereas StudentEval *has on average 36 prompts per problem, representing a variety of prompting skill levels.* This diversity lets us explore what it means to write a "good" prompt.

Our key contributions are:

- StudentEval, a benchmark consisting of 1,749 student-written descriptions of natural-language-to-code tasks.
- Using four key subsets of the StudentEval benchmark, consisting of descriptions that pass (fail) on the first (last) attempt by a student, we evaluate 12 state-of-the-art Code LLMs. Our results show that StudentEval

is better able to discriminate between models than the popular HumanEval benchmark.

- We conduct an in-depth analysis of the prompts and find that even successful student prompts lead models to generate multiple semantically distinct programs.

2 Background

Although Code LLMs allow both code and natural language prompting, we focus on the *natural-language-to-code* task. Liang et al. (2023) report that this is a popular and effective strategy for Code LLM use among experts; it is also more accessible to non-experts than code prompting.

Existing benchmarks pair natural language descriptions of code with test cases to check the validity of generated programs. The two most widely used benchmarks, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), are in Python. There are also multi-language benchmarks that translate problems from one language to another (Athiwaratkun et al., 2023; Cassano et al., 2023). Finally, there are alternate formats, including multi-turn evaluation (Nijkamp et al., 2023) and docstring generation (Lu et al., 2021).

General-purpose benchmarks Most existing benchmarks have a single natural language description of a problem, typically written by an expert programmer. There are a few exceptions that scrape the web or crowdsource (Hendrycks et al., 2021; Lai et al., 2023; Amini et al., 2019), but expert-written benchmarks predominate. These benchmarks provide wide coverage, but come with limitations. First, they have a *single* prompt per problem. Consider this HumanEval prompt:

Imagine a road that's a perfectly straight infinitely long line. n cars are driving left to right; simultaneously, a different set of n cars are driving right to left. The two sets of cars start out being very far from each other. All cars move in the same speed. Two cars are said to collide when a car that's moving left to right hits a car that's moving right to left. However, the cars are infinitely sturdy and strong; as a result, they continue moving in their trajectory as if they did not collide. This function outputs the number of such collisions.

While the correct solution is simply n^2, the prompt is designed to be confusing. Models succeed or fail based on this *specific phrasing*. Having a single prompt precludes explorations of how crucial word choice, grammar, etc. is to model success. STUDENTEVAL's non-expert, multi-prompt construction enables us to analyze what makes a successful prompt: each problem has at least 14 prompts that describe the task in a different way.

Second, existing benchmarks contain problems at widely varying difficulty levels. Compare the problem above, which requires mathematical reasoning that may challenge many programmers, with a trivial problem from the same benchmark (Chen et al., 2021): *Return length of given string*. Although these benchmarks cover a wide range of programming tasks, it is difficult to interpret their results as evidence that a model will or won't suit a particular group of programmers, since they aggregate over very different skill levels.

Domain-specific benchmarks There are also a handful of domain-specific benchmarks, such as DS-1000 (Lai et al., 2023) and MathQA (Austin et al., 2021). Like these domain-specific benchmarks, we target a specific population of programmers; however, we target a particular skill level rather than an application area. In addition, we provide numerous non-expert prompts per problem.

Scalable oversight Our study is related to the problem of scalable oversight (Bowman et al., 2022). Models are capable of solving our problems, since we selected ones for which reliable prompts exist. However, models are unaligned with students. The students understand the task at hand (we remove cases where they did not), but they do not have the prompt-writing skills to guide the model. Our work is thus a first step towards aligning Code LLMs with non-expert programmers.

3 The STUDENTEVAL Dataset

In this section we describe STUDENTEVAL, a many-prompt-per-problem benchmark that targets a specific programmer skill level. The dataset consists of 1,749 English-language prompts for 48 programming problems, with at least 14 prompts per problem. All prompts were written by university students who had completed a single semester of computer science in Python (CS1). These students represent a population of programmers with a uniform knowledge base, allowing us to choose problems that they all should be able to solve.

Problem Selection and Format Given our goal of collecting many non-expert descriptions for each

Function signature (visible)	`def convert(lst):`	
Expert tests (visible to student; hidden from model; automatically run on generated code)	**Input**	**Expected Output**
	`[0,1,2,3]`	`['ABCD']`
	`[0,-1,1,-1,2]`	`['A','B','C']`
	`[1,1,1,-1,25,25,-1,0,1,2]`	`['BBB','ZZ','ABC']`
Student description (pass@1 = 0.8)	*takes a list of numbers. Create a ABC list with the capital letters in the alphabet and create an answer string. Iterate through the input list, if there is "-1" then add ' ' to the answer string, or otherwise, add the letter with the corresponding index of the answer string. Split the answer string at ' '. return the answer string.*	
Student description (pass@1 = 0.0)	*Assign a number from 0~25 to each alphabet, and create a list of string of alphabetical letters based on their assigned numbers in the lst. When there is -1 in the lst, create a new string and add it to the list. Return a list of created strings.*	

Figure 1: An example StudentEval problem. Our web-based experiment platform shows students the signature and expert-written tests. When students submit their description, we use a Code LLM to generate code, test it, and flag failed tests for the students. StudentEval has numerous student-written descriptions of each problem.

problem, we compiled a suite of 48 tasks at an appropriate level for students. The majority were pulled directly from CS1 course materials, with light editing to avoid publishing answers to assignments still in use. Thus, we expect participants to be able to understand and solve the problems in Python themselves. We explore whether they can also describe them in natural language so that Code LLMs can solve them. The problems exercise a variety of Python features. For topic diversity, we define 8 core concepts: lists, loops, strings, conditionals, math, nested data, sorting, and dictionaries.

Each StudentEval problem consists of three components: a function signature, a reference implementation, and 3+ test cases (Figure 1). When we gather student data, which we describe below, we show participants the function's signature and test cases. From this information, they produce a description, which we we automatically validate using the problem's test cases.

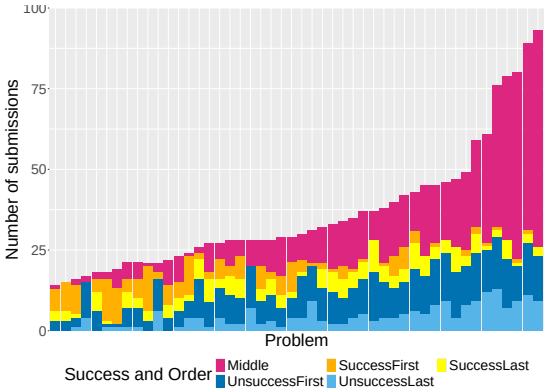
Our participants had taken a CS1 course that routinely uses input/output pairs and signatures to present tasks, so this problem format is familiar to them. Since this format does not show students natural language descriptions, it avoids biasing student prompts with expert descriptions. Thus it mimics a realistic situation in which a student has a task to solve and tries to describe it in natural language to a Code LLM.

Problem Validation We validated our problems in several ways. For common problems (e.g. factorial), LLMs can produce working implementations from the function name alone. To weed out these problems, we produced Codex (Chen et al., 2021) generations from each function signature with no docstring and measured mean pass@1 rate. Overall, the mean pass@1 for our signatures without docstrings is 0.0519 with a variance of 0.0364. The maximum pass@1 is 0.925, for the problem `exp`.

We also validated the test suites for each problem. The test cases serve two purposes: helping students understand the problem, and ensuring that the generated code is correct. Liu et al. (2023) show that the test cases for widely-used Code LLM benchmarks frequently miss important corner cases. To avoid this, we use test coverage and mutation testing of the reference implementation to validate the StudentEval test cases. Unlike in Liu et al. (2023), the StudentEval tests need to be understood by beginners. We strive for a balance between exhaustiveness and comprehensibility: each problem has 3-4 tests that achieve 100% code coverage. Mutation testing (Jia and Harman, 2010) is a more rigorous measure of test suite quality; we used MutPy (Hałas, 2013) to compute mutation scores. All mutation scores below 90 are the result of MutPy only producing trivial mutants that are semantically identical to the reference solution.

Subset	Items	Word Count
First Failure	450	28.8 (25.5) ± 16.7
First Success	187	28.8 (25.0) ± 17.4
Last Failure	205	35.9 (30.0) ± 22.6
Last Success	185	37.8 (35.0) ± 18.4

(a) Sizes and word counts.



(b) Attempts per problem.

Figure 2: The four subsets of STUDENTEVAL.

Gathering 1,749 Student-Written Prompts We recruited 80 beginning CS students from three U.S. higher education institutions to build the STUDENTEVAL benchmark. The IRB-approved study was conducted over Zoom, using a web-based platform designed for STUDENTEVAL (see Appendix). The platform presents the function signature and tests for one problem at a time. Students enter a problem description into a text box. Our server constructs a prompt with the function signature and their problem description formatted as a Python docstring, and sends this prompt to Codex to produce the function body. The server then tests the function in a sandbox and presents the test results to the participant. Students had the option to reattempt the problem or move on. Participants completed 3 tutorial and 8 STUDENTEVAL problems in 75 minutes, receiving a $50 gift card for participation.

Dataset Subsets and Basic Statistics Students generated 1,749 prompts, with an average of 36 prompts per problem. There is significant variation in how prompts differ from each other: some are small, iterative changes (+/- a few words) whereas a student's first, last, and successful prompts tend to be very different. To refine the dataset for evaluation, we partition STUDENTEVAL into four disjoint subsets (Figure 2a): students most frequently failed to solve problems on their first attempt, and this is the largest subset of problems (*First Failure*); about half as many first attempts were successful (*First Success*); slightly fewer students gave up after multiple attempts (*Last Failure*); and others succeeded after multiple attempts (*Last Success*).

These subsets omit "Middle" prompts (Figure 2b), which are intermediate failures. When a student succeeded on their first try, or gave up after their first try, we classified that prompt as a First Success and First Failure respectively. Students never resubmitted after all tests passed. Figure 2a shows that *Last* descriptions are significantly longer than *First*, which suggests students add detail even when starting afresh might be better.

Filtering Prompts A prompt may fail not because the model could not understand the description, but because the student did not understand the problem. We had two expert annotators with extensive CS1 teaching experience label each failing prompt independently. We asked the annotators to determine *Reading the prompt, is it clear that the student understood the problem?*[1] We removed 74 prompts (11%) from the Failure subsets using this criterion.

4 Results

We evaluate 12 Code LLMs. We focus our comparison on gpt-3.5-turbo, the three "Python specialist" Code Llama models (Baptiste Rozière et al., 2023), the four StarCoderBase models (Li et al., 2023), and Phi-1 (Gunasekar et al., 2023). Appendix H presents results for several other models. We confirm that none of the STUDENTEVAL prompts appear in The Stack, the open training dataset for StarCoderBase and other models.

As with other benchmarks, we use hidden unit tests to evaluate the correctness of model-generated code. To account for their nondeterminism, we use the standard *pass@1* metric (Chen et al., 2021), which estimates the probability that the Code LLM produces a solution that passes all hidden unit tests in one shot, calculated over 200 samples.

[1]The annotators studied the gold solution and test cases to understand the task themselves before labeling any prompts.

Table 1: Mean pass@1 for the models that we evaluate on the four subsets of StudentEval.

Model (Size)	First Failure	Last Failure	First Success	Last Success	HumanEval
GPT-3.5-Turbo-0301 (?)	11.76	**13.90**	44.84	47.40	48.1
Phi-1 (1.3B)	12.59	9.64	59.16	36.36	51.22
Replit-Code-v1 (2.7B)	4.25	3.24	33.62	18.33	21.09
SantaCoder (1.1B)	2.32	2.42	30.87	21.71	17.81
StarChat-Alpha (15.5B)	11.23	10.07	63.58	51.06	30.03
StarCoderBase-1B (1B)	1.98	1.39	24.86	13.00	15.17
StarCoderBase-3B (3B)	6.60	6.52	51.73	32.20	21.46
StarCoderBase-7B (7B)	6.13	7.86	62.35	46.42	28.37
StarCoderBase (15.5B)	8.70	7.73	65.28	51.74	30.40
Code-Llama-Py-7B (7B)	7.26	9.88	66.88	55.36	40.48
Code-Llama-Py-13B (13B)	10.66	10.71	70.22	62.26	42.89
Code-Llama-Py-34B (34B)	**12.65**	11.68	**73.51**	**64.65**	**53.29**

4.1 Evaluating Models On StudentEval

Table 1 reports the mean pass@1 rate for every model on the four subsets of StudentEval. We include HumanEval pass@1 rates for comparison.

Code Llama models perform best We find that *the Code Llama models significantly outperform all other models on the* First/Last Success *prompts.* The 13B model outperforms StarCoderBase-15B, the closest competing model, by 5-10% (absolute). The 34B model performs even better.

StudentEval reveals performance differences between small and large models HumanEval is the de facto standard code benchmark; many Code LLM developers focus on HumanEval scores. However, we observe that the difference between pass@1 rates for large and small models is more substantial with StudentEval. 1) For the StarCoderBase models, pass@1 on *Last Success* is almost 4x higher with the 15B model vs the 1B model, but the gap is much smaller (2x) on HumanEval. 2) Phi-1 (1.3B) approaches Code Llama (34B) on HumanEval, but Code Llama is 1.7x better on *Last Success* than Phi-1. One possibility is that Phi-1's textbook training data helps with HumanEval's expert-written prompts, but not on StudentEval: non-experts do not write textbook quality prompts. This shows that small models may perform competitively on expert-written prompts while still struggling with student-written prompts.

4.2 Variation in Pass@1

Most Code LLM papers only report mean pass@1 for a benchmark, averaging over problems with widely varying pass rates. Because StudentEval contains multiple prompts per problem, it illuminates the extent to which luck plays a role in whether a Code LLM produces the right answer for a user. In Figure 3, we group prompts by problem, so the plots show the percentage of problems (Y) with pass@1 lower than the indicated value (X).

For a given model, we define a *reliable failure* as a *First/Last Failure* with pass@1 above 0.8 (to the right of the 0.8 dashed line in the CDF). These are unlucky cases: the prompt failed for the student, but turned out to be reliable. We find that GPT-3.5-Turbo-0301 and StarCoderBase have one and two reliable failures. Similarly, we define an *unreliable success* as a *First/Last Success* prompt with pass@1 less than 0.2. These are lucky cases: the prompt worked once, but that success is hard to replicate.

We find that nearly 10% of successful prompts are unreliable for small models, but less than 3% are unreliable with larger models (Appendix Table 7). This has implications for model selection. It is not adequate to optimize a model to achieve high pass@1 on any benchmark (including StudentEval). Instead, an ideal Code LLM would both maximize pass@1 and minimize its variability.

4.3 Participant Success Rates

Examining prompt success rates by participant reveals a wide spectrum of prompting ability among participants (Figure 4). Although some achieve success rates over 50% with StarCoderBase, a large number struggle to write reliable prompts.

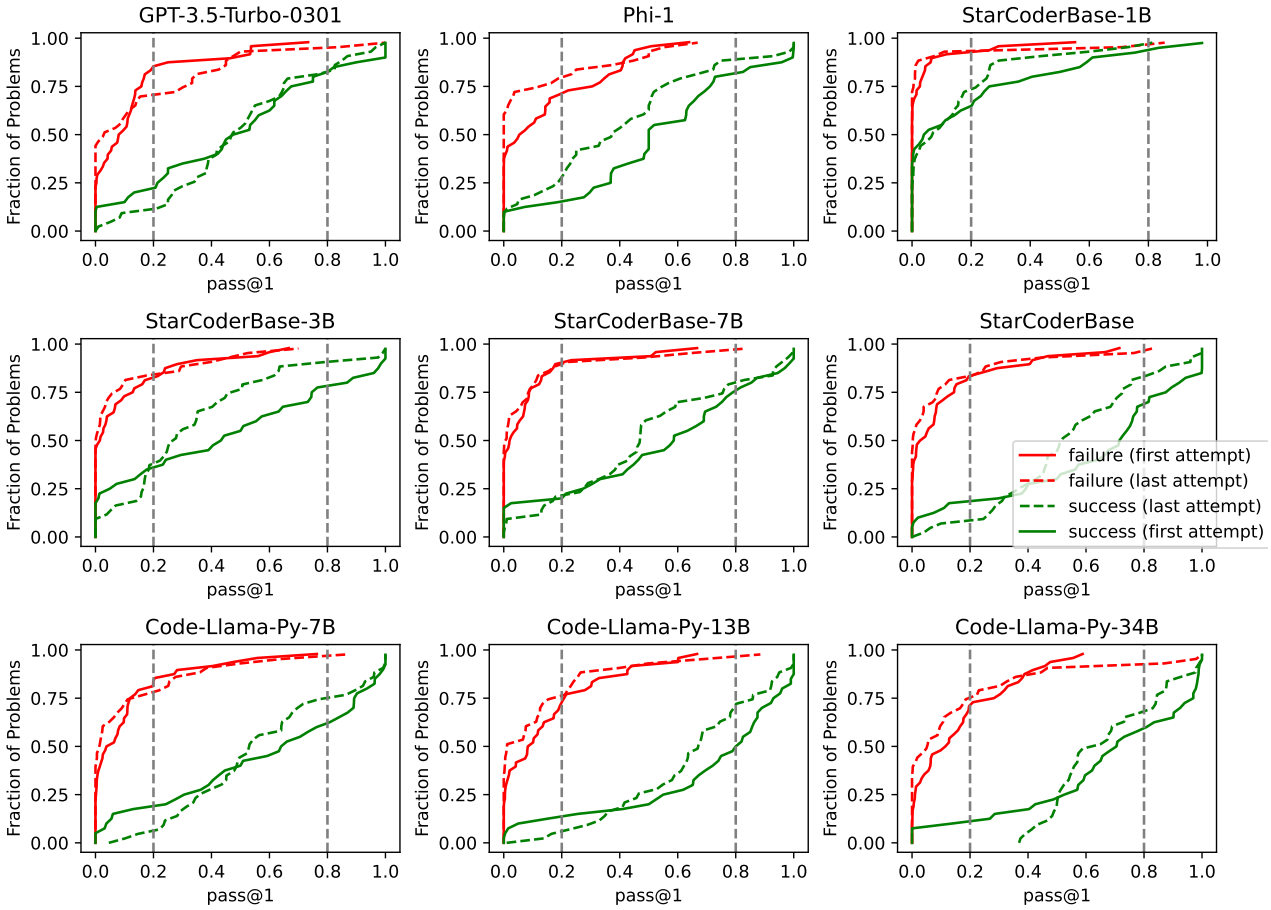


Figure 3: CDFs of mean per-problem pass@1 for Code LLMs on the four subsets of StudentEval. The y-axis shows the fraction of problems in each subset. The x-axis shows by-problem mean pass@1 for student prompts.

5 What Makes a Successful Prompt?

A participant may have a low success rate for various reasons: their prompting skills may be bad, their descriptions may be wrong (we filter these from the benchmark), or they may be writing clear explanations in a style the model does not understand. A low success rate indicates a miscommunication between the model and the user, which is an opportunity to improve the robustness of Code LLM understanding of how beginners describe code. In this section, we explore the factors that impact the success of non-expert-written prompts. Our findings have implications for teaching with Code LLMs, since they highlight successful and unsuccessful student strategies. We use StarCoderBase as an example model in our discussion, but we have replicated key results with Code Llama.

5.1 Trends in Student Word Choice

To explore the relative importance of different words, we computed TF-IDF values, treating each prompt as a document. We used a tokenization approach that supports a mix of Python and English (see Appendix G.1). We computed the mean TF-IDF values calculated across documents in each of the four subsets for the top 25 features (words) in each. Figure 5 shows a heatmap of the overlaps. The top words are a mix of English and Python terms, including many related to types, sequencing, or choice (Figure 5). The inclusion of "return" may be related to the fact that Codex seems to default to printing output, causing tests to fail; students may learn to specify "return" through experience. We see a similar trend for parameter names.

5.2 Statistical Significance of Prompt Wording

We fit mixed-effects regression models to the data to test the impact of prompt length and wording. All models include random effects for problems and use StarCoderBase pass@1 rates as the response variable. For vocabulary features, we use indicator variables: 1 if the prompt uses the word, else 0. Appendix G.3 provides full estimate tables.

Figure 4: Participant mean pass@1 rates with StarCoderBase

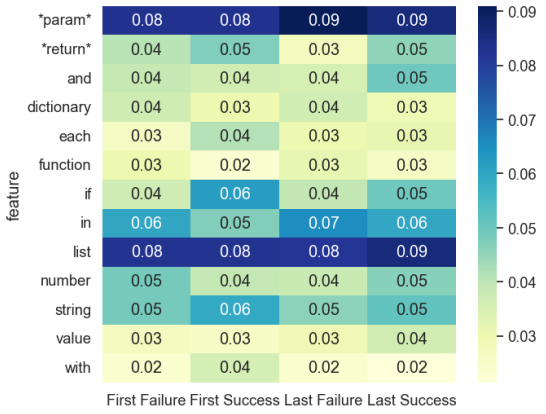


Figure 5: Mean TF-IDF values for overlapping words in the top 25 words for each subset.

Length Contrary to our expectations, we observed a statistically significant positive effect of prompt length on pass@1 rates ($\widehat{\beta}$=0.06, p=0.008). However, this finding seems driven by last submissions, where successful prompts are on average longer; the average length is similar for passing and failing first prompts (Figure 2a). Qualitatively, we have observed that students tend to add text instead of modifying earlier text, which is likely a factor.[2]

Input/output word choice We found a significant positive effect of mentioning "return" in the prompt ($\widehat{\beta}$=0.07, p=0.00002). This likely resolves the problematic ambiguity associated with prompts that mention "output" rather than specifying whether the function should return or print (Figure 6b).

Datatype mentions We explored the effect of mentioning dictionaries, lists, and number types, as well as including instances of lists and dictionaries in the prompt. We found a reliable positive effect of mentioning "list" ($\widehat{\beta}$=0.04, p=0.02), and a borderline negative effect of mentioning "array" ($\widehat{\beta}$=-0.07, p=0.048). This suggests that StarCoderBase is sensitive to Python terminology conventions.

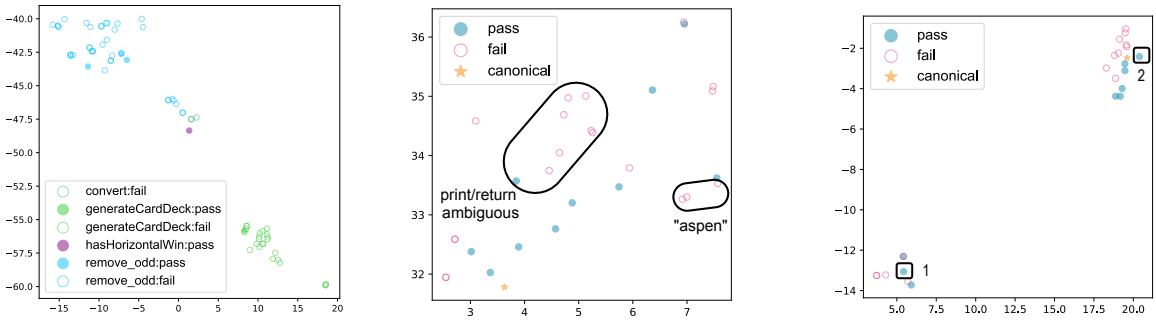
[2]CS1 students have exhibited the same tendency when writing code (Norris et al., 2008; Spacco et al., 2015).

Function and parameter names We found no reliable effect of mentioning the parameter names in the prompt, but a significant negative effect of mentioning the function name ($\widehat{\beta}$=-0.07, p=0.02).

5.3 Inspecting Visual Representations

We generated embeddings of each prompt from the last-layer attention weights of StarCoderBase and Code Llama 34B in order to explore prompt similarities and differences. Figure 6 shows key clusters with StarCoderBase embeddings plotted using t-SNE (Van der Maaten and Hinton, 2008); Appendix G.4 contains a plot of all clusters and Code Llama results, which exhibit the same trends.

Multiple prompt formulations exist Some problems form multiple clusters comprised of different prompting strategies. The `combine` problem prompts form two clusters (Figure 6c). The top right cluster contains succinct prompts, like Prompt 2: *Combine lists from l1 to lists from l2*. The bottom left ones provide step-by-step directions: *Takes an input of two lists, l1 and l2, each of which also contains lists. It combines the first list in l1 with the first one in l2, then continues for all items in l1 and l2. It outputs this final list which is a combination of l1 and l2* (Prompt 1). Both approaches can generate passing programs; future work could explore



(a) Test case cluster (b) `check_for_aspen` prompts (c) `combine` prompts

Figure 6: Prompt embeddings generated using StarCoderBase and reduced using t-SNE.

whether there are style differences between the programs generated by different prompting methods.

Errors and ambiguities pattern together Examining problem sub-clusters gives insight into failure patterns (Figure 6b). One sub-cluster contains prompts that do not say whether the function should print or return the desired value. Another consists of prompts that contain the string "aspen" (lowercase) rather than "Aspen" (upper-case), causing the generated code to fail test cases.

Certain prompting styles are challenging Most prompt embeddings cluster by problem, but some clusters contain prompts for multiple problems, representing cases where the model struggles to distinguish prompts for distinct problems. One such cluster consists of prompts that describe the function's behavior in terms of expected input/output pairs (Figure 6a). Although there are passing examples in this style, it does not seem to work for complex data, such as nested lists or dictionaries; the embeddings for prompts that describe several distinct problems in this way are clustered together. This style of prompt is easy for humans to understand, but appears to challenge current Code LLMs.

5.4 Semantic Ambiguity in Prompts

Most work on Code LLMs asks if models produce correct code for a given prompt. However, it is possible for an ambiguous prompt to generate *semantically* different functions. Semantic equivalence is undecidable, but we can compute a lower bound on the number of semantically different functions: for each prompt completion, we use the test case inputs as a vector of examples. We run each completion to collect a vector of outputs: the function's *test signature* (Udupa et al., 2013). When two functions have distinct test signatures, they are semantically distinct; identical results are inconclusive.

Subset	#Functions
failure (first attempt)	2.2 (2.0) ± 1.6
failure (last attempt)	2.4 (2.0) ± 1.6
success (first attempt)	1.9 (1.0) ± 1.3
success (last attempt)	2.2 (2.0) ± 1.3

(a) Mean (median) & standard deviation of the number of functions produced by StarCoderBase for each prompt.

The function takes a string of text as an input. For words in the string with an odd number of letters, every other letter is capitalized starting with the first letter. For words in the string with an even number of letters, every other letter is capitalized starting with the second letter.

(b) A *First Success* prompt that produces 7 functions.

Figure 7: StudentEval prompts can be ambiguous to LLMs and produce several distinct functions.

Figure 7a summarizes results for each StudentEval subset. Overall, prompts generate a surprising number of distinct functions, and even prompts that are relatively clear to humans can generate many distinct functions (Figure 7b). This highlights the importance of evaluating prompt *reliability*. Though the Figure 7b prompt produced a passing function during the experiment, it was likely to fail. This has key implications for Code LLMs as teaching tools (see Finnie-Ansley et al. (2022); Leinonen et al. (2023)): reliability issues may mislead students into thinking their descriptions are better than they are or into over-complicating descriptions that are actually high-quality.

6 Conclusion

We present StudentEval, a large Code LLM benchmark of prompts written by students who have taken a single CS1 course. A key feature of

StudentEval is the numerous natural language descriptions per problem, written by beginning programmers with varying levels of prompting skill.

We show that larger models are more capable of following student-written instructions than smaller models. We also find that many student-written prompts are unreliable (have low pass@1): students get lucky (or unlucky) when using Code LLMs, an issue for educational use of Code LLMs. Finally, we investigate the question of what makes a good prompt from several angles, finding that models struggle to understand some valid strategies, such as giving examples of complex data.

We hope that StudentEval will make it easier to evaluate how well Code LLMs work for natural language instructions written by non-experts, leading to the development of models that are better aligned with this key group of users.

Acknowledgements

We thank our colleagues who helped us recruit participants and for the problems that we adapted, Northeastern Research Computing and the New England Research Cloud for providing computing resources, Loubna ben Allal for help integrating StudentEval into the BigCode Evaluation Harness, and the ARR reviewers for their thoughtful feedback. This work is partially supported by the National Science Foundation (SES-2326173, SES-2326174, and SES-2326175).

Limitations

Although our findings shed light on how well Code LLMs work with descriptions written by one key group of non-experts, there is more work to be done. We study only one group of non-experts (beginning students); moreover, our participants were recruited from three selective institutions within the US. Other groups of students or other populations of non-experts may use different strategies to describe code. Moreover, because there are few multilingual Code LLM models, we look only at prompting in English. This highlights the need for more work exploring how diverse populations of non-experts might interact with Code LLMs.

Our participants wrote their prompts interactively while using a single model (Codex). It is possible that they would have revised their problems differently with a different model. This is one reason we do not emphasize comparisons between Codex and other Code LLMs in our evaluation.

When we piloted in November 2022, the Codex model we used (code-davinci-002) was the most capable Code LLM available. Despite being "old," it remains as good as gpt-3.5-turbo on established benchmarks: gpt-3.5-turbo and code-davinci-002 score 48% and 46% respectively on HumanEval. Code Llama 34B scores 48%. This suggests that the Codex model we use is as capable at code completion as many newer models.

The capabilities of Code LLMs also vary by programming language (Cassano et al., 2023, 2024). Our results may not generalize to languages other than Python.

Ethics Statement

There are two main ethical concerns for this work: (1) ethical concerns about the involvement of student research participants and (2) concerns about how the dataset could be used in future work.

Our work was conducted in accordance with the Brandeis University Human Research Protection Program. Potential harms to student participants were a first-class consideration in the design. We sought to address power dynamics and protect participant autonomy with a number of measures. We collected data in an opt-in manner, outside of the classroom, and with informed consent. The researcher conducting the study was not affiliated with the participant's institution. Students were asked to complete programming assignments with familiar content and were alerted to potential discomfort associated with using an AI-based tool.

All identifying information has been removed from the dataset. We have released the full dataset via the Open Science Framework; participants consented to the release of their anonymized data. The Appendix also contains a "Datasheet for Dataset" outlining pertinent dataset information. We provide pertinent screenshots and text illustrating the experimental platform in the Appendix. The full experimental protocol is available through the Open Science Framework.

Our second ethical concern is that releasing this dataset may lead to the development of technology that we would not build ourselves, such as attempts to automate education in a way that negatively impacts the educational experience of students. We feel that the benefits of providing this data, which we hope will lead to Code LLMs that work better for non-expert users, outweigh this risk.

It is also possible that future users may general-

ize results from the dataset beyond what is appropriate; our study involves early CS students in a particular educational context (selective US institutions) and may not generalize to other populations.

Finally, this research was only possible due to model access and funding. To obtain the benchmark results from the 12 LLMs, we used around 2 weeks of GPU time on an H100 GPU. There are ongoing ethical concerns about access to models and infrastructure. The evaluation of the dataset in this paper centers both open-source and small-scale models, but fully addressing these issues should be a priority for the broader community.

References

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2357–2367, Minneapolis, Minnesota. Association for Computational Linguistics.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

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*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Ellen Tan, Yossef (Yossi) Adi, Jingyu Liu, Tal Remez, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez, Jade Copet, Faisal Azhar, Hugo Touvron, Gabriel Synnaeve, Nicolas Usunier, and Thomas Scialom. 2023. Code Llama: Open Foundation Models for Code.

Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111.

Samuel R. Bowman, Jeeyoon Hyun, Ethan Perez, Edwin Chen, Craig Pettit, Scott Heiner, Kamilė Lukošiūtė, Amanda Askell, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Christopher Olah, Daniela Amodei, Dario Amodei, Dawn Drain, Dustin Li, Eli Tran-Johnson, Jackson Kernion, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Liane Lovitt, Nelson Elhage, Nicholas Schiefer, Nicholas Joseph, Noemí Mercado, Nova DasSarma, Robin Larson, Sam McCandlish, Sandipan Kundu, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Timothy Telleen-Lawton, Tom Brown, Tom Henighan, Tristan Hume, Yuntao Bai, Zac Hatfield-Dodds, Ben Mann, and Jared Kaplan. 2022. Measuring progress on scalable oversight for large language models. *arXiv preprint arXiv:2211.03540*.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *arXiv preprint arXiv:2308.09895*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

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*.

James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA. Association for Computing Machinery.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644*, (arXiv:2306.11644).

Konrad Hałas. 2013. *Cost Reduction of Mutation Testing Process in the MutPy Tool*. Ph.D. thesis, Instytut Informatyki.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and

Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678.

Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–23, Hamburg Germany. ACM.

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.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: a natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 124–130, New York, NY, USA. Association for Computing Machinery.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*. Reproducibility Certification.

Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the usability of ai programming assistants. *arXiv preprint arXiv:2303.17125*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Cindy Norris, Frank E. Barry, James B. Fenwick Jr., Kathryn Reid, and Josh Rountree. 2008. Clockit: collecting quantitative data on how beginning software developers really work. In *Annual Conference on Innovation and Technology in Computer Science Education*.

James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction*, page 3617367.

Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. 2015. Analyzing student work patterns using programming exercise data. In *ACM Technical Symposium on Computer Science Education*, page 18–23, New York, NY, USA. Association for Computing Machinery.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. Transit: Specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the*

2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22, New York, NY, USA. Association for Computing Machinery.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research*, 9(11).

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*, pages 21–29.

A Access

The dataset can be accessed on the Open Science Framework (https://doi.org/10.17605/OSF.IO/WDPSX). Analysis and evaluation code are available through Github (github.com/Wellesley-EASEL-lab/StudentEval). The benchmark problems are also part of the BigCode Evaluation Harness (github.com/bigcode-project/bigcode-evaluation-harness) and available on the Hugging Face Hub (huggingface.co/datasets/wellesley-easel/StudentEval).

B Author Statement

As authors, we acknowledge that we bear all responsibility in case of violation of rights, etc. Our dataset is licensed under an OpenRAIL-D license.

C Hosting, Licensing, and Maintenance Plan

Our dataset is hosted on the Open Science Foundation and licensed under an OpenRAIL-D license. We plan to maintain public access to the dataset, but we do not plan to accept new contributions to it. For more details, see our Datasheet (below).

D Datasheet for Dataset

D.1 Motivation

For what purpose was the dataset created? Was there a specific task in mind? Was there a specific gap that needed to be filled? This dataset was created as a benchmark for code generation models. It was created with the goal of filling two gaps in the existing benchmarks: 1) the need for a benchmark with multiple natural language descriptions per problem and 2) the need for a benchmark targeting a specific programmer skill level.

Who created the dataset and on behalf of which entity? This dataset was created by researchers at Northeastern University, Wellesley College, and Oberlin College: Drs. Arjun Guha, Carolyn Anderson, and Molly Feldman, along with students in their labs. Dr. Arjun Guha is also affiliated with Roblox Research.

Who funded the creation of the dataset? This work was funded by the National Science Foundation (SES-2326173, SES-2326174, and SES-2326175).

D.2 Composition

What do the instances that comprise the dataset represent? Are there multiple types of instances? The dataset consists of programming tasks where each task has the following components: a function signature; a test suite; an expert-written implementation; an expert-written prompt; and a set of student-written prompts (minimum 14).

How many instances are there in total? There are 48 programming tasks and 1,749 student-written prompts.

Does the dataset contain all possible instances or is it a sample of instances from a larger set? The dataset is not a sample of a larger set, but it does not contain all possible instances, since additional programming tasks or prompts could be devised.

What data does each instance consist of? "Raw" data or features? Each instance consists of text. The dataset as a whole is stored in CSV format.

Is there a label or target associated with each instance? Each prompt is labeled with the name of the programming task it is associated with.

Is any information missing from individual instances? If so, please provide a description, explaining why this information is missing. This does not include intentionally removed information, but might include, e.g., redacted text. We have intentionally de-identified the dataset so that prompts cannot be traced back to the students who wrote them.

Are relationships between individual instances made explicit? If so, please describe how these relationships are made explicit. Yes. Each prompt is associated with the programming task it described, and we have also provided an ID linking together prompts written by the same student.

Are there recommended data splits? If so, please provide a description of these splits, explaining the rationale behind them. No.

Are there any errors, sources of noise, or redundancies in the dataset? Some prompts may appear multiple times in the dataset, either because multiple students described the problem in the same way, or because a student re-submitted a prompt without editing it.

Is the dataset self-contained, or does it link to or otherwise rely on external resources? The dataset is self-contained and does not link to or rely on external resources.

Does the dataset contain data that might be considered confidential? No.

Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety? No.

Does the dataset relate to people? Yes.

Does the dataset identify any subpopulations? No.

Is it possible to identify individuals, either directly or indirectly from the dataset? If so, please describe how. We believe that this is not possible. We have de-identified the dataset, removing participant usernames and replacing them with randomly generated numeric IDs. We have also manually reviewed the dataset to ensure that there is no personally-identifying information that could link prompts back to participants.

Does the dataset contain data that might be considered sensitive in any way? No.

D.3 Collection Process

How was the data associated with each instance acquired? Was the data directly observable, reported by subjects, or indirectly inferred/derived from other data? If data was reported by subjects or indirectly inferred/derived from other data, was the data validated/verified? The data was collected during a human-subjects experiment, where participants wrote prompts to describe a function. Participants were then presented with a solution generated by the Codex code generation model (Chen et al., 2021), as well as the results of running a suite of test cases on the solution. Participants could submit any number of prompts for a particular problem (within the time limit of the experiment).

Participants submitted the data using our web-based experiment platform. There was no validation of their submissions, but the experiment was overseen in real time by an experimenter, who was available to answer questions and intervene when participants ran into issues with the task.

What mechanisms or procedures were used to collect the data? How were these mechanisms or procedures validated? The experiment was conducted on a web-based experiment platform built by the research team. We conducted a small-scale pilot study to assess the functionality of the platform.

If the dataset is a sampled from a larger set, what was the sampling strategy? The dataset is not sampled.

Who was involved in the data collection process and how were they compensated? We recruited participants who had taken an introductory Python programming course at Northeastern, Wellesley, or Oberlin within the past two years. Each participant was given a $50 Amazon gift card for particpating in the approximately 75 minute task.

Over what timeframe was the data collected? Does this timeframe match the creation timeframe of the data associated with the instances? The data was collected between November 2022 and May 2023.

Were any ethical review processes conducted? If so, please provide a description of these review processes, including the outcomes, as well as a link or other access point to any supporting documentation. The study was conducted under supervision of the Brandeis Human Research Protection Program, which acts as Wellesley College's IRB. Northeastern and Oberlin entered into authorization agreements with Brandeis. If you have questions or concerns about this project, you can contact the Brandeis HRPP: hrpp@brandeis.edu.

Does the dataset relate to people? Yes.

Did you collect the data from the individuals in question directly, or obtain it via third parties or other sources? We collected the data directly from participants as part of a lab-based study.

Were the individuals in question notified about the data collection? If so, please describe how notice was provided, and provide a link or other access point to, or otherwise reproduce, the exact language of the notification itself. Participants submitted an informed consent form prior to participating. They verbally affirmed their ongoing consent at the beginning of the study.

Did the individuals in question consent to the collection and use of their data? If so, please describe how consent was requested and provided, and provide a link or other access point to, or otherwise reproduce, the exact language to which the individuals consented. Yes. The informed consent form is available to view on the Open Science Framework site for this dataset.

If consent was obtained, were the consenting individuals provided with a mechanism to revoke their consent in the future or for certain uses? If so, please provide a description, as well as a link or other access point to the mechanism. Participants were allowed to retract their data prior to its public release, by contacting the researchers.

Has an analysis of the potential impact of the dataset and its use on data subjects been conducted? Yes, the impact of releasing this data was considering during the IRB process.

D.4 Preprocessing/cleaning/labeling

Was any preprocessing/cleaning/labeling of the data done? If so, please provide a description. If not, you may skip the remainder of the questions in this section. The dataset was de-identified to preserve participant anonymity; we removed all usernames and replaced them with randomly generated numeric IDs. We filtered out 74 prompts from the benchmark (section 3).

Tokenization was done to facilitate the analysis presented in the paper, but the released dataset contains the submitted responses, not the tokenized ones. The tokenization script is available on the Github for this dataset.

We had two expert annotators with extensive CS1 teaching experience assess each of the prompts in the failing subset. We asked the annotators to determine whether it was clear from the prompt that the student understood the problem. We removed 74 prompts (11%) from the Failure subsets using this criterion.

Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data? If so, please provide a link or other access point to the "raw" data. We will only release the de-identified dataset, not the raw dataset. This is in order to preserve participant anonymity.

Is the software used to preprocess/clean/label the instances available? Yes. All code is avail-

able at the Github repository linked at github.com/Wellesley-EASEL-lab/StudentEval.

D.5 Uses

Has the dataset been used for any tasks already? No.

Is there a repository that links to any or all papers or systems that use the dataset? No.

What (other) tasks could the dataset be used for? The primary intended use for this dataset is as a benchmark. However, it could also be used to fine-tune machine learning models. We imagine that it could be useful to fine-tune a code generation model to better handle the way students talk about code. It could also be used in tandem with pass@k rates to fine-tune a prompt classification model.

Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses? For example, is there anything that a future user might need to know to avoid uses that could result in unfair treatment of individuals or groups or other undesirable harms. If so, please provide a description. Is there anything a future user could do to mitigate these undesirable harms? Although a diverse population of students was involved in this research, representing many dialects of English, we have not examined how representative it is of the population of English-speaking students. As a result, benchmark results calculated with this dataset might not generalize to all populations of potential code generation users.

Are there tasks for which the dataset should not be used? If so, please provide a description. This dataset should not be used to build artifical intelligence that aims to deceive humans (e.g. by spreading misinformation or by impersonating a human). Certain uses are also restricted by the OpenRAIL license under which this dataset has been released.

D.6 Distribution

Will the dataset be distributed to third parties outside of the entity on behalf of which the dataset was created? The de-identified dataset will be made public.

How will the dataset will be distributed? Does the dataset have a digital object identifier (DOI)? The dataset is distributed through the Open Science Framework, available at https://doi.org/10.17605/OSF.IO/WDPSX.

When will the dataset be distributed? The dataset is currently available.

Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)? If so, please describe this license and/or ToU, and provide a link or other access point to, or otherwise reproduce, any relevant licensing terms or ToU, as well as any fees associated with these restrictions. The dataset is licensed under an OpenRAIL-D License Agreement. The license can be found on the Open Science Framework project associated with the dataset.

Have any third parties imposed IP-based or other restrictions on the data associated with the instances? If so, please describe these restrictions, and provide a link or other access point to, or otherwise reproduce, any relevant licensing terms, as well as any fees associated with these restrictions. No.

Do any export controls or other regulatory restrictions apply to the dataset or to individual instances? No.

D.7 Maintenance

Who will be supporting/hosting/maintaining the dataset? The Open Science Framework will host the dataset.

How can the owner/curator/manager of the dataset be contacted (e.g., email address)? Questions or concerns about this dataset can be directed to Arjun Guha (a.guha@northeastern.edu); Carolyn Anderson (carolyn.anderson@wellesley.edu); or Molly Feldman (mfeldman@oberlin.edu).

Is there an erratum? No, but if errors are discovered, we will post one to the Open Science Framework project.

Will the dataset be updated? No. The dataset is stable.

If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances? If so, please describe these limits and explain how they will be enforced.

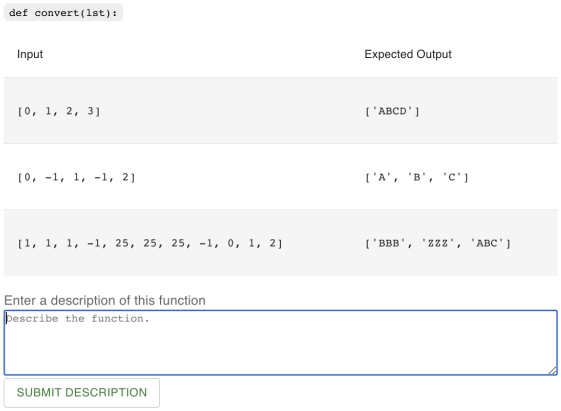


Figure 8: The screen that students see when entering a problem description.

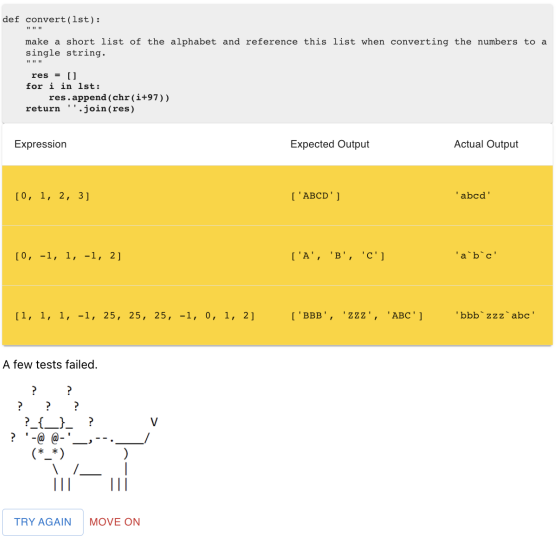


Figure 9: We run expert tests automatically and highlight ones they fail.

Participants were informed that their de-identified data would be published online and remain accessible in perpetuity. All identifying information will be destroyed, as outlined in the IRB protocol.

Will older versions of the dataset continue to be supported/hosted/maintained? We do not plan to update this dataset.

If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so? If so, will these contributions be validated/verified? If not, why not? We welcome replications of our work, but we think these should be published as separate datasets in order to make the population and data collection times clear. Students are a dynamic population, and their experience with writing prompts is likely to change in the near future.

E Details of Web Application for Data Collection

We collected data via a purpose-built web-based application. See Figures 8, 9, and 10 for screenshots of the interface. Students encountered 8 different problems selected from the 48 overall as part of the study. The interface remained the same for each problem.

E.1 Eligibility Criteria

StudentEval is built from problems selected from three undergraduate, introduction to computing courses. Specifically, the problems were chosen from previous instances of CS111 at Wellesley College, CS150 at Oberlin College, and DS2000 at Northeastern University. All three courses are taught using Python and cover the same basic topics.

Students were eligibility for the study if they were currently an undergraduate student at one of the three institutions and had completed one of the above courses over five possible terms (Fall 2021 through Spring 2023). Students could be currently enrolled in a subsequent course, but were not eligible for the study if they had completed subsequent courses.

E.2 Tutorial/Training Information

Instructions for how study participants should interact with the Code LLM were provided via three tutorial problems. Each problem was chosen specifically to exhibit the range of possible Code LLM interactions. We provided a working prompt alongside the first problem (to showcase model success), the second problem was an "impossible" problem (to showcase model failure), and the third was an easy-to-solve problem that modeled the full participant interaction. Relevant text describing each tutorial problem is provided below:

- **Problem 1:** "The inputs and expected outputs are examples of what the function should do, and the text box is where you'll describe the function's behavior. Try copy and pasting: "Takes in a string, and returns an integer representing the maximum number that the same letter appears consecutively in the string." then click SUBMIT DESCRIPTION."

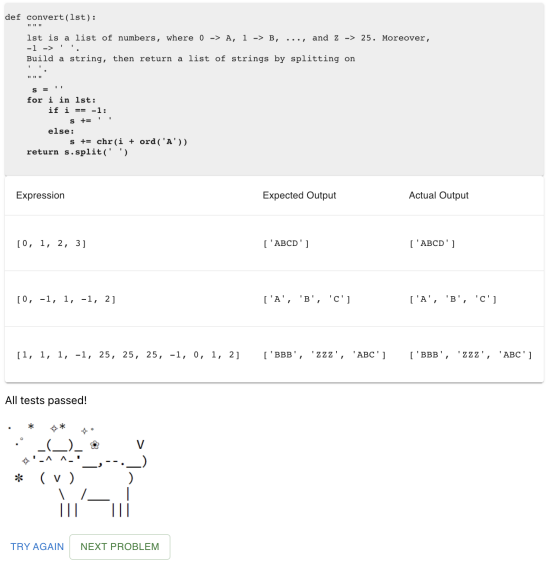


Figure 10: The screen students see when all tests pass.

- **Problem 2:** "Looking at the inputs and expected outputs below, try to figure out what the function `apply_operations` should do, and type it in the text box below."
- **Problem 3:** "Try writing a description for this function on your own based on the inputs and expected outputs, then click SUBMIT DESCRIPTION."

E.3 Study Timing Information

Participants completed the tutorial, provided descriptions, and performed an exit survey & interview over approximately 75 minutes. During the main experiment, students were randomly assigned eight problems out of the total 48 problems in StudentEval. The first four problems were untimed and students had, at maximum, 5 minutes to work on each of the second set of four problems. Throughout the study, students were given the option to "Try Again" or "Move On" to the next problem at will. This is one of the causes of the variation in the number of responses per user, per problem in StudentEval.

F Dataset Details

F.1 Dataset Examples

To illustrate the variety of student-written descriptions, we show three descriptions of the `generateCardDeck` problem that are in the *Last Success* subset:

1. For each character in suits, create a list that appends each of the suits with each of the values in vals. Sort each card in deck by alphabetical order of suit character.

2. There are two lists of strings. Combine the first element with the first list with the first element in the second list and add that to a new list, which will be returned [ex, if the first element of the first list is F and an element of the second list is 7, combine them like F7]. Keep adding the first element of the first list to each element of the second list and add each addition to the new list. Reorder the new list to make it in alphabetical and numerical order, then return

3. when given two lists containing strings, the function will return 1 list. Each individual string within the first list will be combined with each string in the other list, with letters appearing first and numbers appearing second. the new strings in the new list will appear in alphabetical order. if multiple strings have the same first letter, then they will appear in numerical order.

And three descriptions of the same problem in the *Last Failure* subset:

1. This function inputs two lists. The letter that comes first within the alphabet in the first list adds on the lowest number of the second list and then the letters in the alphabet. This keeps going and forms a new list with all the combined letters and numbers.

2. Enter two arrays of characters. Reverse the order of the first array. Print an array with the the first array multiplied by the second array.

3. there are two lists, both with strings. you should use a nested for loop to concatenate the two strings and then add them to a list in decreasing order. Increasing order means spades (S) are greater than hearts (H), and hearts are greater than diamonds (D). "J" is smaller than "Q", which is smaller than "A". suit takes precedence over rank. when you concatenate the letter should be before the number.

F.2 Prompt reliability

Table 7 shows the number of reliable and unreliable prompts in each subset for all benchmarked models.

G Prompt Analysis Details

G.1 Tokenization & Pre-Processing

As a pre-processing step, we replace functionally equivalent words with placeholders. This pre-processing was done as, across all problems in StudentEval, there are 48 different function names (e.g., `convert`, `fib`) and 57 different argument names (e.g., `val`, `meetings`). Therefore, we replace references to functions and parameters with *FUNCTIONNAME* and *PARAM*, respectively. Our approach does not handle capital function or argument names, as their meaning is ambiguous (e.g., "Convert" is a verb, but `convert` is a function name). We also replace "return"/"returns" with *RETURN*.

Student prompts consist of a mix of Python terminology (including code snippets) and English words. Therefore, standard English tokenization libraries were insufficient. We perform a best-effort tokenization using a regex-based Python function that performs multiple passes. The overall goal was to maintain meaningful code-related items as single terms. Specifically, we treat list indexing, lists, dictionaries, single/double quote strings, numbers, and comparison operators as single tokens. There may be possessives and/or contractions that are tokenized as strings rather than separate terms in the dataset. Terms were additionally lowercased and basic stopwords which are not meaningful in a programming context were filtered out.

G.2 TF-IDF Analysis

We used `scikitlearn`'s `TfidfVectorizer` with our tokenizer to generate the TF-IDF values presented in this paper. The list of all prompts was provided to `fit_transform`. Figure 14 presents the mean values for each of the top 25 words for each of the four subsets.

G.3 Regression Analysis

We fitted mixed-effects regression models to predict StudentEval pass@1 rates estimated with completions obtained from StarCoderBase. Models were fitted using the `lme4` library in R. All models included random intercepts for problems; random slopes were omitted due to model complexity. For vocabulary-level features, we use indicator variables: a 1 if the prompt uses the word and a 0 otherwise. Values that are statistically significant with a threshold of $p = 0.5$ are displayed in **bold**.

Prompt length Prompt length was calculated by number of tokens using the tokenizer discussed in Section G.1. The raw token count was divided by 100 for scaling purposes. The full estimates are shown in Table 3.

Fixed effects	$\widehat{\beta}$	z	p
Intercept	0.15 (+/-0.04)	3.5	**0.001**
totalLength	0.06 (+/- 0.02)	2.8	**0.008**

Table 3: Mixed-effects regression results for problem length

Input/output wording We explored the impact of mentioning "return", "input", "print", and "output". We counted all stemmed mentions. The full estimates are shown in Table 4.

Fixed effects	$\widehat{\beta}$	z	p
Intercept	0.20 (+/- 0.03)	6.2	**< 0.0001**
returnInd	0.07 (+/- 0.02)	4.2	**< 0.0001**
inputInd	0.022 (+/- 0.03)	0.8	0.45
printInd	-0.008 (+/- 0.03)	-0.3	0.76
outputInd	0.025 (+/- 0.02)	1.1	0.27

Table 4: Mixed-effects regression results for input/output terms

Datatype mentions We explored the impact of mentioning "list", "dictionary", "array", "variable", "number", "int", as well as giving example lists and dictionaries (indicated by use of square or curly braces). The full estimates are shown in Table 5.

Fixed effects	$\widehat{\beta}$	z	p
Intercept	0.22 (+/- 0.03)	6.9	**< 0.0001**
list	0.042 (+/- 0.02)	2.3	**0.02**
dict	0.006 (+/- 0.05)	0.13	0.90
squareBrace	-0.21 (+/- 0.4)	-0.6	0.57
curlyBrace	0.37 (+/- 0.2)	1.7	0.08
array	-0.07 (+/- 0.04)	-2.0	**0.048**
variable	0.03 (+/- 0.04)	0.7	0.49
number	0.009 (+/- 0.02)	0.48	0.63
int	0.023 (+/- 0.02)	1.2	0.24

Table 5: Mixed-effects regression results for datatype mentions

Function and parameter names We explored the effect of mentioning the function name and the name of parameters. The full estimates are shown in Table 6.

Model (Size)	First Failure	Last Failure	First Success	Last Success	HumanEval
Code-Llama-Py-13B (13B)	9.56	9.33	70.22	62.26	42.89
Code-Llama-Py-34B (34B)	11.40	10.14	73.51	64.65	53.29
Code-Llama-Py-7B (7B)	6.51	8.59	66.88	55.36	40.48
GPT-3.5-Turbo-0301 (?)	10.86	12.41	44.84	47.40	48.1
Phi-1 (1.3B)	11.28	8.37	59.16	36.36	51.22
Replit-Code-v1 (2.7B)	3.84	2.83	33.62	18.33	21.09
SantaCoder (1.1B)	2.08	2.11	30.87	21.71	17.81
StarChat-Alpha (15.5B)	10.10	8.78	63.58	51.06	30.03
StarCoderBase (15.5B)	7.82	6.74	65.28	51.74	30.40
StarCoderBase-1B (1B)	1.77	1.21	24.86	13.00	15.17
StarCoderBase-3B (3B)	5.91	5.66	51.73	32.20	21.46
StarCoderBase-7B (7B)	5.49	6.82	62.35	46.42	28.37

Table 2: Results from evaluation on additional models.

Fixed effects	$\widehat{\beta}$	z	p
Intercept	0.25 (+/- 0.03)	8.3	**< 0.001**
param	0.01 (+/- 0.02)	0.5	0.62
functionname	-0.07 (+/- 0.03)	-2.3	**0.02**

Table 6: Mixed-effects regression results for function name and parameter name mentions

G.4 Embedding Plot

Figures 11- 13 are analoguous to Figure 6, but with Code Llama 34B instead of StarCoderBase. See Figures 15 and 16 for a plot of the t-SNE (Van der Maaten and Hinton, 2008) projection for student description embeddings.

H Results on More Models

Table 2 reports mean pass@1 with StudentEval on 12 models.

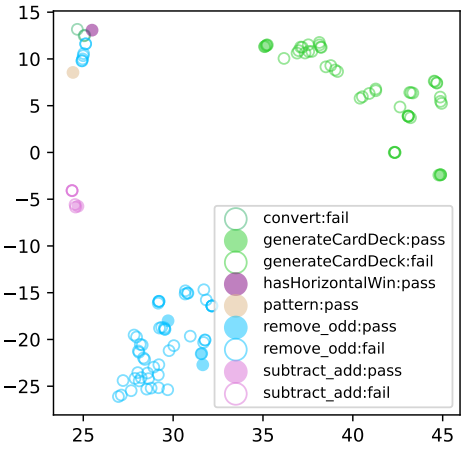


Figure 11: Prompt embeddings generated using Code Llama 34B and reduced using t-SNE: Test case cluster

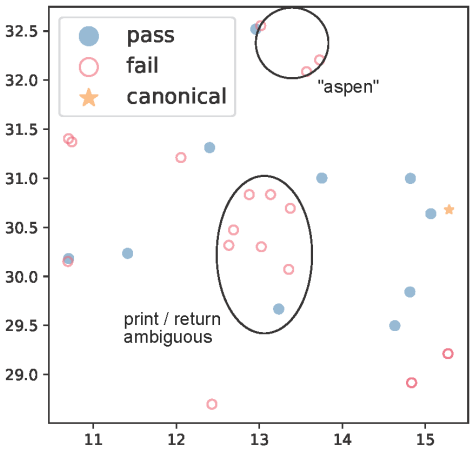


Figure 12: Prompt embeddings generated using Code Llama 34B and reduced using t-SNE: `check_for_aspen` prompts

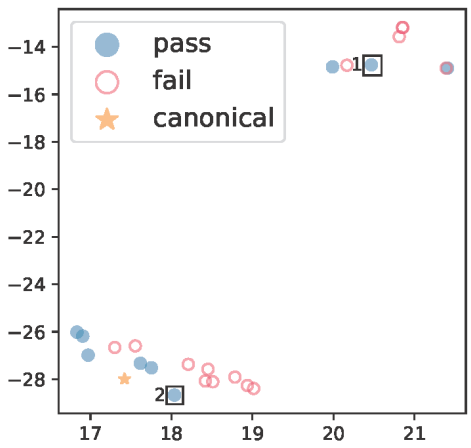


Figure 13: Prompt embeddings generated using Code Llama 34B and reduced using t-SNE: `combine` prompts

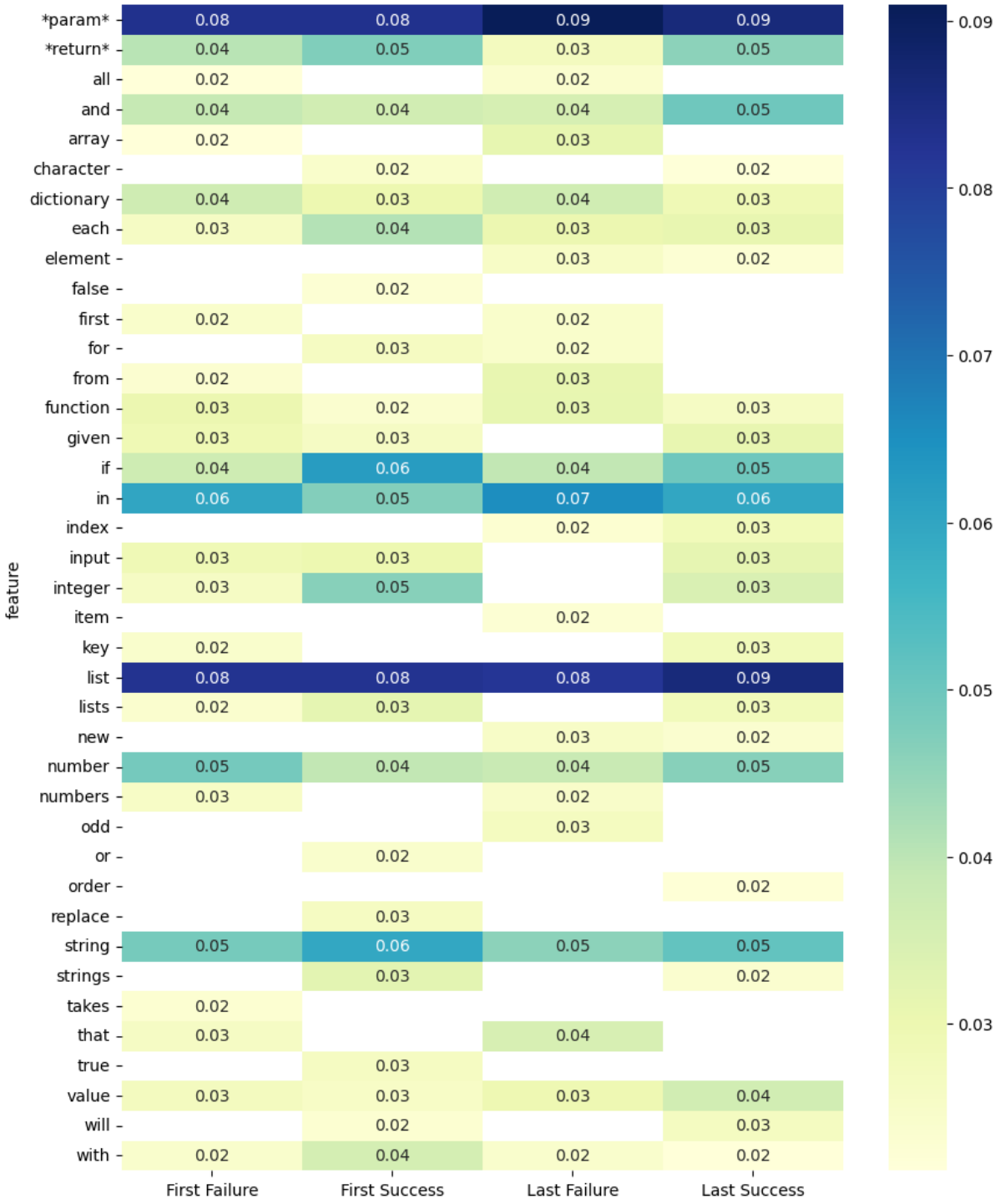


Figure 14: TF-IDF values for top 25 words in each prompt subset category.

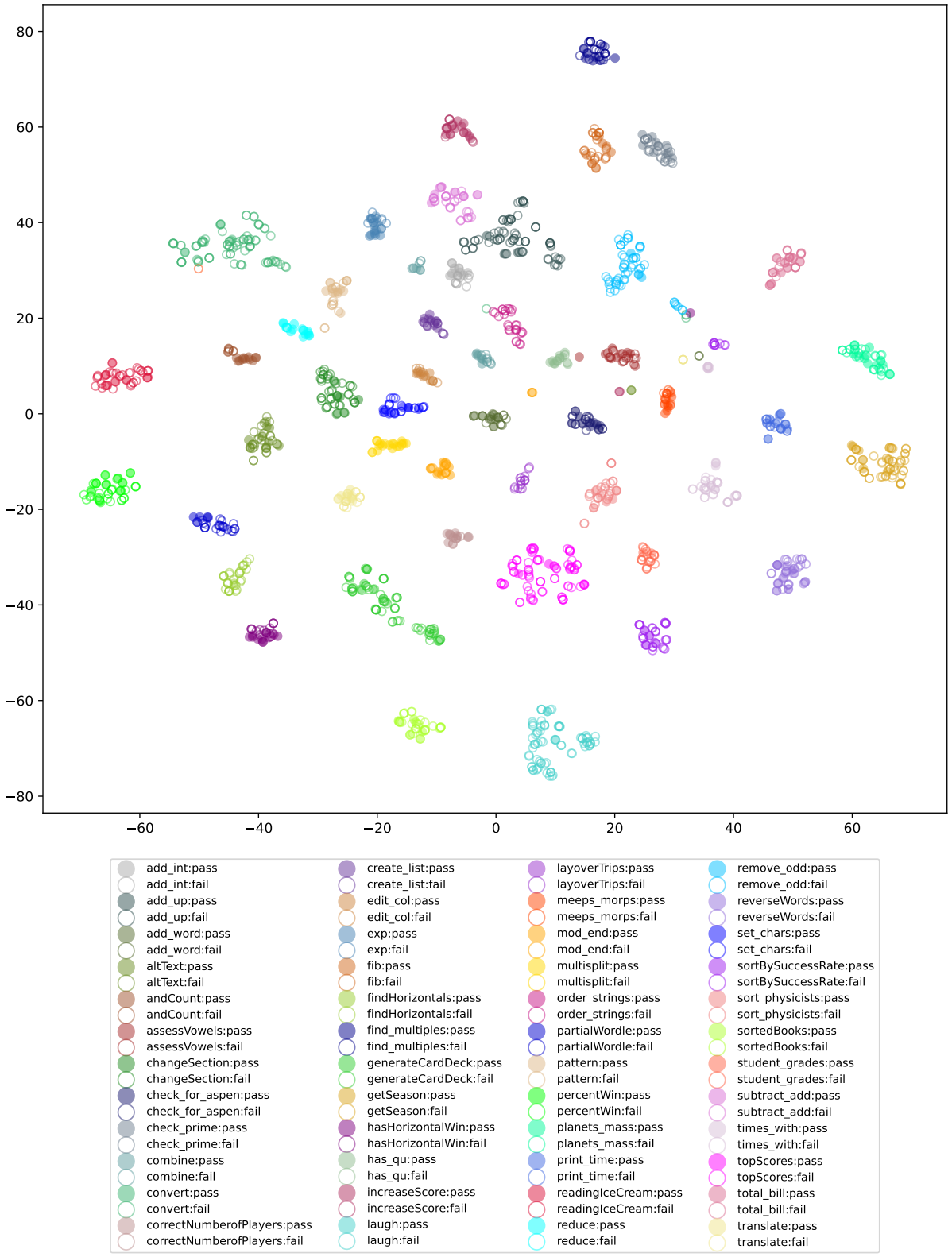


Figure 15: t-SNE projections of student prompt embeddings with colored by problems. Filled and hollow circles represent passed and failed prompts, respectively.

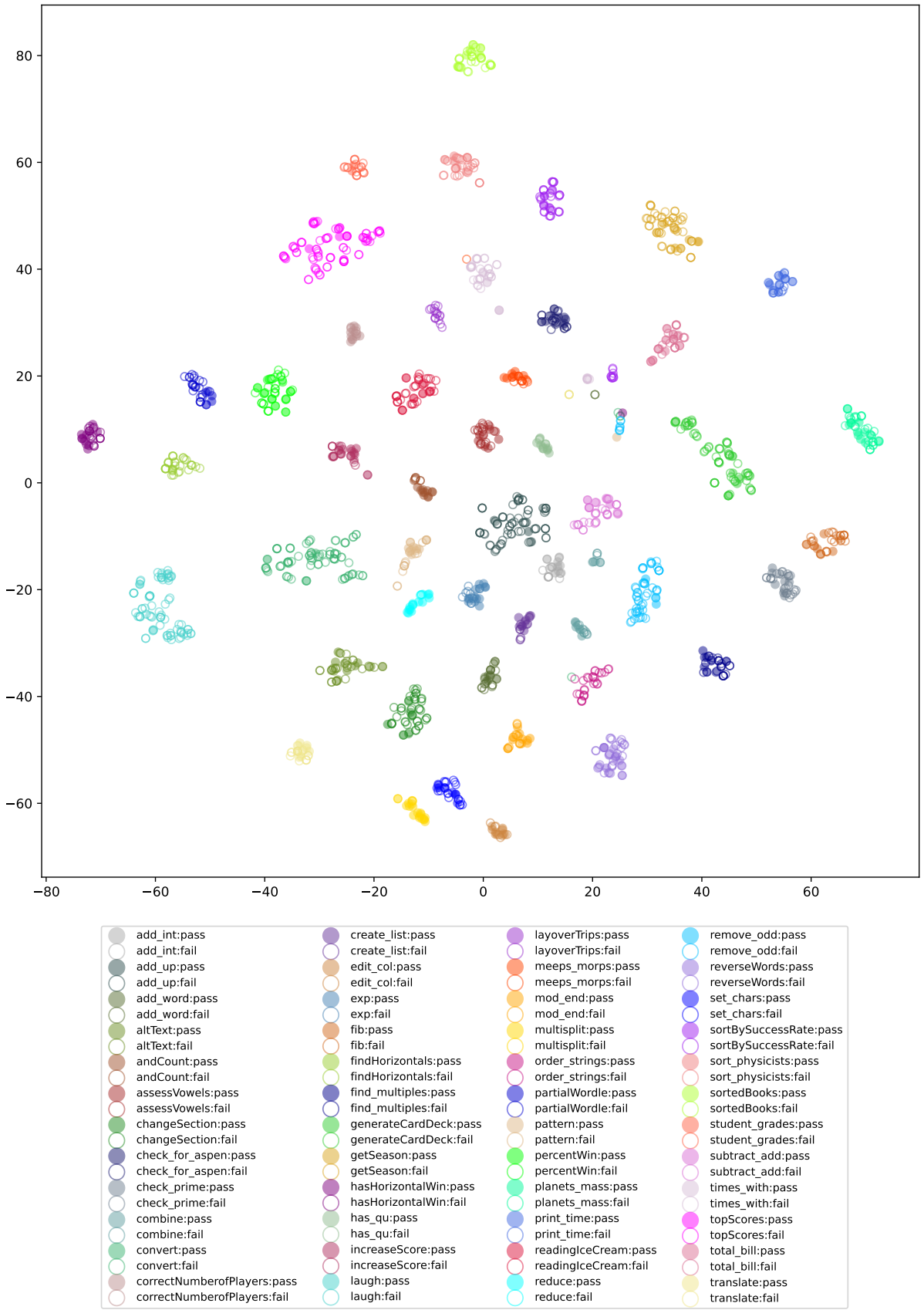


Figure 16: t-SNE projections of student prompt embeddings with Code Llama 34B, colored by problems. Filled and hollow circles represent passed and failed prompts, respectively.

Table 7: Number of reliable and unreliable prompts by model and subset.

Model	Rate / Subset	pass@1 < 0.2	pass@1 > 0.8
Code-Llama-Py-13B	failure (first attempt)	38	-
	failure (last attempt)	34	1
	success (first attempt)	6	19
	success (last attempt)	3	12
Code-Llama-Py-34B	failure (first attempt)	36	-
	failure (last attempt)	34	3
	success (first attempt)	5	16
	success (last attempt)	-	13
Code-Llama-Py-7B	failure (first attempt)	41	-
	failure (last attempt)	36	1
	success (first attempt)	8	15
	success (last attempt)	3	10
GPT-3.5-Turbo-0301	failure (first attempt)	42	-
	failure (last attempt)	31	2
	success (first attempt)	9	6
	success (last attempt)	5	7
Phi-1	failure (first attempt)	35	-
	failure (last attempt)	37	-
	success (first attempt)	7	7
	success (last attempt)	13	4
Replit-Code-v1	failure (first attempt)	43	-
	failure (last attempt)	40	-
	success (first attempt)	18	2
	success (last attempt)	25	-
SantaCoder	failure (first attempt)	45	-
	failure (last attempt)	42	-
	success (first attempt)	21	3
	success (last attempt)	23	1
StarChat-Alpha	failure (first attempt)	38	-
	failure (last attempt)	37	-
	success (first attempt)	7	11
	success (last attempt)	5	6
StarCoderBase	failure (first attempt)	41	-
	failure (last attempt)	37	1
	success (first attempt)	8	12
	success (last attempt)	4	7
StarCoderBase-1B	failure (first attempt)	45	-
	failure (last attempt)	41	1
	success (first attempt)	26	2
	success (last attempt)	32	1
StarCoderBase-3B	failure (first attempt)	40	-
	failure (last attempt)	38	-
	success (first attempt)	15	8
	success (last attempt)	17	3
StarCoderBase-7B	failure (first attempt)	44	-
	failure (last attempt)	39	1
	success (first attempt)	9	9
	success (last attempt)	10	8