S🗿cratic Questioning of Novice De🐞gers: A Benchmark Dataset and Preliminary Evaluations

Erfan Al-Hossami[1], **Razvan Bunescu**[1], **Ryan Teehan**[2], **Laurel Powell**[1],
Khyati Mahajan[1], and **Mohsen Dorodchi**[1]

[1]University of North Carolina at Charlotte, Charlotte, NC
[2]New York University, New York City, NY
`{ealhossa, rbunescu}@uncc.edu`

Abstract

Socratic questioning is a teaching strategy where the student is guided towards solving a problem on their own, instead of being given the solution directly. In this paper, we introduce a dataset of Socratic conversations where an instructor helps a novice programmer fix buggy solutions to simple computational problems. The dataset is then used for benchmarking the Socratic debugging abilities of GPT-based language models. While GPT-4 is observed to perform much better than GPT-3.5, its precision, and recall still fall short of human expert abilities, motivating further work in this area.

https://github.com/taisazero/socratic-debugging-benchmark

1 Introduction and Motivation

Educational needs for computer science (CS) are on the rise, due to increased enrollments in CS programs (Camp et al., 2017). Higher education institutions in particular are affected by the lack of sufficient instructional staff, often resorting to hiring undergraduate Teaching Assistants (TAs) in their computer science courses. An effective TA benefits students by providing timely feedback and assistance that is tailored to each student's level of proficiency, with measurable and significant impact on student retention rates (Mirza et al., 2019). In practice, however, not all educational institutions benefit uniformly from their TAs. Depending on class sizes and TA allocations, it is often the case that a teaching assistant cannot spend their time equally with all students who need help, especially when nearing office hours or an assignment deadline. Moreover, students who lack fundamental knowledge from prerequisite courses consume significant TA time throughout the course. This comes at a time when there is also a shortage of K-12 computer science teachers, a lack of appropriate training for K-12 educators interested in teaching CS effectively (Yadav et al., 2016), and rising TA and peer instruction demand in flipped computer science classrooms (Maher et al., 2015).

Overall, the lack of instructional staff, ranging from TAs to K-12 teachers and college educators, motivates the automation of various types of teaching tasks by leveraging the increasing capabilities of AI models, especially in terms of understanding and generating language and code. Prior work in AI for programming education is primarily composed of intelligent tutoring systems (ITS) and learning support systems for programming courses. While some ITS systems allow interactions with a learner through a chat interface (Hobert, 2019), the range of interactions is often limited, as tutoring systems typically focus on giving hints constructed for predefined solutions or predefined Socratic utterances that are specific to a known set of programming exercises (Jeuring et al., 2014; Gerdes et al., 2017; Hobert, 2019; Alshaikh et al., 2020b). Consequently, traditional ITS systems in the programming domain do not generalize to new courses or new coding assignments without human intervention. This situation is however rapidly changing, due to the substantial leaps in performance exhibited by large language models recently, on a wide array of problems. Language models are now capable of solving introductory programming exercises (Hendrycks et al., 2021; Chen et al., 2021) including custom problems created by instructors (Finnie-Ansley et al., 2022). Furthermore, solutions generated by these models are unique and can fool plagiarism software such as MOSS (Biderman and Raff, 2022), presenting educators with further challenges in maintaining academic integrity.

In Socratic questioning, a teacher assists a learner trying to solve a problem beyond their zone of proximal development (Quintana et al., 2004). Language Models (LMs) have been used effectively for generating a particular type of Socratic questions for solving word math problems, wherein

Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023), pages 709–726
July 13, 2023 ©2023 Association for Computational Linguistics

they leverage the sequential structure of steps that compose the solution (Shridhar et al., 2022). Other applications of LMs include automated feedback on student code submissions (Wu et al., 2021), as well as generating programming exercises, unit tests, and code explanations (Sarsa et al., 2022). However, there still remains a substantial gap in leveraging LMs effectively for guiding novice programmers through a coding exercise in a way that maximizes their learning outcomes, similar to how an effective, experienced TA would guide a beginner programmer. For Socratic questioning, in particular, the difficulty of building an effective system is compounded by the scarcity of examples, whereas the limited data that can be found (Chen et al., 2011) does not have sufficient structure to enable the automatic evaluation of Socratic questioning systems.

In this paper, we focus on the task of Socratic questioning for debugging (Wilson, 1987), or Socratic debugging, defined as a conversation between a knowledgeable programmer and a beginner student who comes for help fixing a buggy solution for a simple computational problem (Section 2). To enable the development and evaluation of LM-based instructional agents, we introduce a manually created dataset of dialogues where the main objective is for the student to repair their buggy code themselves by leveraging guidance received from the instructor at every turn (Section 3). However, as originally observed by Wilson (1987), "no precise formula, or line of questioning" is needed to achieve the goals of Socratic questioning. Furthermore, depending also on their expectations with respect to the student's abilities, an instructor can often think of multiple ways of guiding the student at any particular turn in the conversation, leading to a very large space of possible dialogues. Socratic questions lie in a continuum ranging from providing direct hints that give out the answer to offering minimal guidance, enabling instructors to pose queries at an appropriate level that challenges the student while remaining within each student's ability to answer. To facilitate the automatic evaluation and benchmarking of future Socratic questioning systems in terms of their precision and recall, the dataset contributors are asked to provide all alternative utterances that they think could help the student, at every turn in the conversation. This is a currently ongoing, cognitively demanding data generation effort, requiring contributors with substantial experience in tutoring beginner programmers. We use the current version of the dataset, containing 86 main conversations, to benchmark the Socratic debugging abilities of two large language models in the GPT family, namely GPT-3.5 and GPT-4 (Section 4), noticing a large discrepancy in performance in favor of the more recent GPT-4. We conclude the paper with related work and limitations.

2 Task Definition

We formulate the Socratic debugging task as a dyadic conversation between a Student and an Instructor. In this scenario, the Student is assumed to be a beginner programmer who has recently started learning how to code in Python. As part of his[1] learning to code curriculum, the Student is given a coding problem for which he needs to write a function implementing the specified input-to-output relationship. The Student writes the code for the function, however, the code is buggy and he cannot make progress on his own without help, therefore he seeks help from the Instructor. The Instructor is assumed to be a proficient programmer in Python with experience in teaching novice programmers how to code. When contacted by a Student for help, her main aim is to maximize the learning outcomes by following a Socratic guidance approach through which, over one or more dialogue turns, she helps the students figure out where the bug is and how to fix it on their own.

2.1 Input

Since the focus of this work is on generating Socratic guidance and not bug identification or fixing bugs, we assume that the AI agent implementing the Instructor also has access to a description of the bug and of one or more bug fixes. The decision to separate Socratic advice generation from bug identification and debugging was motivated by the fact that these subordinate tasks can already be solved efficiently by large LMs with high accuracy. Therefore, at the start of each conversation, we assume the Instructor has access to the *problem description*, a number of *test cases*, the student's *buggy code*, the *bug description*, and one or more *bug fixes*, as shown below in a sample from our dataset. At each turn in the conversation, the Instructor's task is to generate Socratic guidance in response to the Student's current progress in addressing the

---

[1] The genders were selected at random by tossing a coin.

bug. Consequently, we assume that the Instructor is also given as input a history of the conversation so far, ending with the last utterance from the student. Shown below is an example ending with the second turn from the student, where the turn number is indicated between brackets.

➢ **Problem description**:
Write a function `factorial(n)` that computes the factorial `n!` of a natural number `n`, which is defined mathematically as:

```
0! = 1
n! = n x (n - 1)!
```

Additionally, if the input integer `n` is negative the function should return `0`.

➢ **Test cases**:

```
assert factorial(-1) == 0
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(2) == 2
assert factorial(3) == 6
assert factorial(4) == 24
assert factorial(5) == 120
```

➢ **Buggy code**:

```
1. def factorial(n):
2.    if n < 0:
3.      return 0
4.   fact = 1
5.   for i in range(n):
6.     fact = fact * i
7.   return fact
```

➢ **Bug description**:
On line 6, `fact` is multiplied with `0` in the first iteration of the for loop. Consequently, at every iteration `fact` stays equal with `0` instead of being updated to be equal with factorial of `(i + 1)`. Therefore, the function will return `0`, irrespective of `n`.

➢ **Bug fixes**:

1. Replace `i` with `(i + 1)` on line 6.
2. Replace `range(n)` with `range(1, n + 1)` on line 5.

To summarize, the input for the Instructor agent consists of:

1. The **problem description**, a number of **test cases**, the student's **buggy code**, the **bug description**, and one or more **bug fixes**.
2. The **conversation so far**, ending with the last turn from the Student.

➢ **Conversation so far**:
[1] STUDENT: Hi! I implemented the factorial function but it doesn't work and I do not know why. Can you help?
[1] INSTRUCTOR: Sure. Can you tell me for what values of `n` it fails and what values it returns in those cases?
[2] STUDENT: For `n = 1` or larger it returns the same value, `0`.
[2] INSTRUCTOR: ⟨*Socratic guidance*⟩

2.2 Output

Using the input data described above, the Instructor is expected to generate Socratic guidance appropriate for the current state of the conversation, as shown below.

➢ **Socratic guidance**:
Main responses:

> Let's see what happens when `n` is 1. What is the first value that is assigned to variable `i` in line 5?

Alternative responses:

1. Let's see what happens when `n` is 1. Before line 6 is evaluated in the first iteration of the for loop, what are the values of the variables `fact` and `i`?
2. Let's see what happens when `n` is 1. Can you insert a new line between lines 5 and 6 that prints the values of the variables `fact` and `i`?
3. Let's see what happens when `n` is 1. What does `range(n)` do when `n` is 1?
4. Can you tell me what `range(n)` does?

The example above shows a total of 5 Socratic responses, partitioned into 1 main response and 4 alternative responses. Most of the time there are different ways of guiding the student, and ideally, the Instructor should be able to generate all different types of Socratic guidance that are different from each other in non-trivial ways. For example, the

4th alternative focuses the student on correcting the potential misuse of the `range` function, whereas the main response provides a different kind of guidance wherein the student is expected to first notice the wrong code behavior that is caused by the misuse of `range`. Further justification for the decision to include alternative responses will be provided in Section 3 when introducing the data contribution guidelines. Note that only the main response is used to create the history of the conversation so far that is used as input for generating future Instructor turns.

3 Benchmark Dataset

To facilitate the development of conversational agents that act under the task definition above, we manually created a dataset of dialogues where a student fixes buggy code on his own by leveraging the Socratic guidance received from an instructor. The dataset is created by sequentially specifying the *Coding problem* → *Bugs* → *Conversations* → *Threads*. First, a coding problem is selected, normally a simple coding exercise situated at a novice level of coding proficiency, such as `Factorial` or `Fibonacci`. The coding problem is specified through the problem description and the associated test cases. Next, one or more buggy implementations are created, with the constraint that each implementation contains exactly one bug. The bugs were selected to reflect common types of mistakes that beginner programmers make, such as forgetting that indexing of sequences starts at 0, boundary bugs, operator misuse, or misunderstanding of basic programming constructs.

For each buggy implementation, a main conversation is created, where a fictional Student, the author of the buggy code, interacts with a fictional Instructor. The aim of the instructor is to guide the student to discover the cause of the bug and fix it on his own through Socratic dialogue. The dialogue always starts with a student utterance. The instructor and the student then take turns in a dialogue, until the bug is successfully fixed. At each turn, the student may also provide a block of code if he made edits to the code at that turn.

Following research in dialogue systems (Gupta et al., 2019), we create multiple reference instructor utterances at each turn. The Main utterance may be optionally followed by one or more Alternative utterances. Given that the aim of this dataset is to benchmark the ability of an artificial Instructor agent to generate Socratic guidance, it is especially important that the contributed main and alternative utterances for the Instructor comprehensively explore the entire range of Socratic advice at that point in the conversation. These alternative utterances should be semantically distinct in a non-trivial manner; in particular, they should not be mere paraphrases of the main utterance or of each other. Upon inspection of the conversations created manually, we discovered that one contributor used a vending machine as an analogy to guide the user to conclude that `print` is not the same as `return`. While using analogies can substantially enhance the impact of Socratic questioning, it can lead to an open-ended range of alternatives, as the number of possible analogies is virtually infinite. Since our aim is to create a dataset that can be used to estimate both the recall and precision of a Socratic guidance generator, at this stage we decided to require that Socratic utterances be *literal*, leaving the generation of figurative utterances as a direction for future work. For the Student, alternative utterances may give different or conflicting answers to an Instructor question, reflecting different levels of understanding. Students may give correct or incorrect answers; they may also introduce new bugs when fixing the original bug.

Problems	23
Bugs	34
Dialogues	86
Student turns	537
Student utterances	763
Instructor Turns	497
Instructor utterances	1,329
Total turns	1,034
Total utterances	2,092

Table 1: Summary of the benchmark dataset: Number of programming problems, bugs, dialogues (including all threads), turns, and total utterances (main and alternatives) for both roles (student and instructor).

Once the main conversation ends with the student successfully correcting their code and passing all test cases, the contributors are instructed to create up to three conversational threads.

The dialogues in the dataset were created by 10 contributors with extensive experience in CS education as instructors, teaching assistants, or tutors. The starting problems and buggy implementations were selected to contain a variety of syntactic

Language Model	Manual			BLEU-4			BERT F1			Rouge-L		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
GPT-3.5	19.8	31.5	24.3	3.2	1.9	1.8	56.0	38.6	37.4	21.0	13.3	12.8
GPT-4	52.9	50.0	51.4	3.2	5.6	3.8	35.4	63.3	42.0	14.1	24.9	16.7

Table 2: Preliminary evaluation of GPT-3.5 (`gpt-3.5-turbo`) and GPT-4 on our benchmark dataset. Manual evaluation is performed on all instructor turns from a sample of 5 dialogues, whereas automatic evaluation is performed on the entire dataset. We report the Precision (P), Recall (R), and F1 for the manual evaluation, and BLEU-4, BERT F1, and Rouge-L for the automatic evaluation. All results are percentages (%).

and semantic mistakes that are frequently made by novice Python programmers.

To streamline and standardize the collection of Socratic dialogues and code edits for each input problem description and buggy implementation, we developed a 7-page web application using the Streamlit[2] and gsheetsdb[3] libraries. The application guides contributors through selecting a bug, creating initial and conversational threads, and reviewing and submitting their work. During the process, contributors can add main and alternative utterances, undo actions, and edit the chat history. The application also allows importing and exporting dialogues in a standardized form for review. For more details and images of the application, the reader is referred to Appendix A.

4 Experimental Evaluations

We evaluate the GPT-3.5 (OpenAI, 2022) and GPT-4 (OpenAI, 2023) language models in terms of their capacity to generate, at each instructor turn, Socratic utterances that match those contributed in the benchmark dataset. Each test example is composed of an input prompt to the language model containing: a steering prompt for Socratic questioning adapted from the GPT-4 blog post[4], the problem description, the buggy code, the bug description, the bug fixes, the unit tests, the dialogue history so far, and an instruction to the language model to generate all possible semantically distinct Socratic utterances, as shown below.

> Respond to the user with all possible distinct Socratic utterances that guide the user to discover and fix the bug described between '<bug_desc>' and '</bug_desc>'. Student code is written between '<code>' and '</code>' throughout the conversation. Utterances that have the same meaning but different words are considered duplicates. Assume that the student has run the test cases.

The list of utterances generated by the LM is then used to estimate precision and recall. After conducting a preliminary, qualitative evaluation of various prompts and instructions we select the prompt and instruction used in this paper. For more details about prompting, the reader is referred to Appendix B.

In all experiments, LM outputs are generated using a greedy decoding setting (i.e. temperature = 0). We set a maximum generated token threshold of 1,024 and do not apply any frequency or presence penalties. We perform manual evaluation of the LM generations for a subset of problems, and automatic evaluations for all problems in the benchmark dataset.

4.1 Manual Evaluation

In the manual evaluation process, we aim to estimate the performance of GPT-3.5 and GPT-4 by manually assessing the quality of their generated instructor utterances. At each instructor dialogue turn, we manually examine each LM utterance to determine if it is an appropriate Socratic utterance at that turn. We sample a total of 17 instructor turns across 5 dialogues from the benchmark. Using the example listed in §2.1, during the second instructor turn a good-matching generated utterance example is: "How does the range function work in your loop, and what values does it generate for i?" because it is semantically close with the ground truth utterance: "Can you tell me what range(n) does?". If the LM utterance is good but not present in our dataset, we mark it as missing to compute an overall upper bound on recall for the dataset itself. These missing alternatives can later be used to augment the

---

[2] https://streamlit.io/

[3] https://github.com/betodealmeida/gsheets-db-api

[4] https://openai.com/research/gpt-4

dataset. An example of a good LM utterance that is not in the dataset: “Let’s take a closer look at the loop in your code. Can you explain how the loop iterates and what it does in each iteration?", this utterance is distinct from the first alternative response as the generated utterance gives more autonomy to the student by simply asking the student to explain the buggy portion of the code with less guidance on what to explain or look for. If the LM output is not good, it is considered a false positive (FP), which decreases the precision of the LM. An example of a poor-matching utterance in the same setting is: “Can you think of a way to modify the loop so that it starts with a different value of i?". This utterance is generated too early before the student realizes that the loop starts with an ‘i’ value of 0. For each alternative in the benchmark dataset at that turn, we check if it is missing from the list of LM utterances. If missing, it is considered a false negative (FN), which decreases the recall of the LM. If the dataset utterance is present in the LM utterances, it is considered a true positive (TP). LM and dataset instructor utterances are matched only if they are semantic equivalent. If the LM generates two or more paraphrases of the same Socratic guidance, for the purpose of evaluation they are considered as one Socratic utterance. The precision (P), recall (R), and their harmonic mean (F1) presented in Table 2 highlight GPT-4’s superior performance over GPT-3.5 in generating relevant and diverse Socratic utterances. We emphasize GPT-3.5’s poor precision as it tends to generate many poor Socratic questions (93 FP) compared to GPT-4 (41 FP) that may contain keywords in common with a ground truth utterance but are irrelevant. In addition to the evaluation of language models, we compute the (R) for our benchmark dataset during human evaluations, obtaining a value of 70.0. This score suggests that most of the high-quality Socratic utterances generated by language models are effectively captured within the dataset by contributors, further validating the dataset’s usefulness for benchmarking purposes.

4.2 Qualitative Analysis

We examine the LM-generated outputs in our manual evaluation setup and observe interesting phenomena. In general, GPT-3.5 seemed to refer to example cases in the problem description to ask for the expected output in its utterances whereas GPT-4 rarely did so. Furthermore, GPT-4 has generated utterances that are good (32) but not in the dataset 3 times more than GPT-3.5 (10) indicating that GPT-4 generates more diverse (e.g. in the level of guidance) and comprehensive Socratic utterances. GPT-3.5 has generated over 96 poor utterances (FP) which is more than double that of GPT-4 (42). Below are cherry and lemon picked LM output samples given a buggy code containing a return statement within the for loop leading to an early termination. Good responses are bolded and colored in **green** while poor utterances are italicized and colored in *red*.

➢ **Buggy code**:

```
def removeEven(nums):
    newNums = []
    for i in nums:
        if i % 2 != 0:
            newNums.append(i)
        return newNums
```

➢ **Conversation So Far**:

[1] STUDENT: Help, I’m stuck.

[1] INSTRUCTOR: What should your code be doing?

[2] STUDENT: Right now, it’s only returning the first number in the list, not all of them.

[2] INSTRUCTOR: Okay, let’s look at your loop. Do you see anything that could be causing it to end early?

[3] STUDENT: No I do not. I’m iterating through the whole list of numbers.

[3] INSTRUCTOR: ⟨*Socratic guidance*⟩

➢ **Sample GPT-4 Outputs**:

1. **Can you identify which part of your code is responsible for returning the final result?**
2. **What happens when the ‘return’ statement is executed inside the loop?**
3. **How does the placement of the ‘return’ statement affect the execution of your code?**
4. *How can you modify the position of the ‘return’ statement to ensure that the entire list is processed before returning the result? (direct)*

➢ **Sample GPT-3.5 Outputs**:

1. **What does the ‘return’ statement do?**

2. *How can you modify the code to return all odd numbers in the list? (early)*
3. *What happens when the condition on line 4 is true? (irrelevant)*
4. *Can you identify any issues with the loop? (repeated)*

In the example above, we observe that GPT-4 covers three distinct levels of guidance. The first utterance simply guides the student's focus onto the erroneous line and the second utterance exerts higher level of guidance by asking the student to demonstrate their understanding of the 'return' keyword, and lastly the third exerts even more guidance by asking the student to explain the impact of indentation on code execution. GPT-3.5's second utterance is illustrative of a poor utterance as it provides very little guidance and is unhelpful for the student in that conversation. Poor utterances for both LMs fall into 4 categories. The first and largest category are ***irrelevant*** utterances, where the SQ diverts the learner's attention away from the actual bug and may mislead them as a consequence. GPT-3.5 has generated over 53 irrelevant utterances significantly more compared to GPT-4 (8). An example of an irrelevant utterance is the third GPT-3.5 utterance where the LM directs the focus of the learner away from the loop and why it might be terminating early and towards explaining the if statement and its body where there is no bug. The sudden shift in the goal of the conversation from discussing possible causes of the bug to explaining non-buggy code lines may mislead the learner to thinking the if statement and its body may be causing the bug when they are not. This category of utterances must be minimized by systems performing Socratic questioning. The second category are ***repeated*** Socratic utterances that had been asked in a prior turn or the answer to the Socratic question was given by the student in a prior turn. For example, the fourth GPT-3.5 utterance asking if the student observes any issues with the loop coming right after the student had said they don't see anything causing the loop to end early. The third category are SQs that are ***too direct*** by making the bug fix pretty obvious early in the conversation. An illustrative example of this is the fourth GPT-4 utterance where it makes the bug fix obvious which is de-indenting the return statement before the student discovers the cause of the bug. These utterances lower the challenge level for students while learning and prevent students from engaging in a discovery process and potentially lowers learning outcomes. The last category is composed of SQs uttered ***too early*** in the conversation, where student is not yet aware of the issue, and the Socratic utterances guide the student towards changing the code before they realize what the issue is. Take the second GPT-3.5 utterance as an example, where the LM asks the student how can they modify their code to fix the bug before the student even discovers the cause of the bug. This category of poor utterances may cause confuse learners.

4.3 Automatic Evaluation

Following prior work in Socratic sub-question generation (Shridhar et al., 2022), we compute the similarity between an LM utterance and a ground truth utterance in the dataset using BLEU (Papineni et al., 2002) for n-gram overlap, BERT F1 Score (Zhang et al., 2020) for semantic similarity based on the DeBERTa language model[5] (He et al., 2020), and Rouge-L (Lin, 2004) for n-gram overlap based on Longest Common Subsequence (LCS) between generated and reference instructor utterances. Rouge-L is included for its flexibility in evaluating text similarity and capturing overall structure and content better than BLEU-4. BERTScore is included to handle paraphrases. Given a set of m LM-generated utterances and n manually created utterances, we create a complete bipartite graph between the two sets, with a total of mn edges, where the weight of each edge is computed using one of the text similarity measures above. We then apply Edmond's Blossom algorithm (Galil, 1986) for finding the maximum matching in this bipartite graph. This ensures that each manual utterance is matched with at most one LM utterance, effectively prohibiting semantically equivalent LM utterances from artificially increasing the evaluation measures. The number of true positives TP is computed by summing up the weights of all edges found in the optimal matching. Given that the weights are similarity scores in $[0, 1]$, if an LM utterance u is matched with a manual utterance v for a similarity weight of $s(u, v)$, the remaining weight mass of $1 - s(u, v)$ is considered to contribute towards the total number of false positives FP. Any unmatched LM utterance is considered to contribute the maximum of 1 towards the FP total. Overall, it can be shown that this results in $FP = m - TP$. The number of false

---

[5] https://huggingface.co/microsoft/deberta-xlarge-mnli

negatives is computed in an analogous way, resulting in $FN = n - TP$. Consequently, precision is $P = TP/m$ and recall is $R = TP/n$.

The results of evaluating GPT-3.5 and GPT-4 on the entire benchmark dataset using these automated metrics are shown in Table 2. We observe that there is a correlation in terms of F1 and R between the automatic metrics and the manual metrics. However, upon manual inspection reveals that automatic evaluation metrics tend to increase when generated Socratic questions contain variable names from the buggy code input or statements from the bug description. This occurs regardless of the question's relevance or usefulness to the student, emphasizing the importance of manual evaluation for this task.

5 Related Work

▶ **Education and Socratic Questioning.** Scaffolding is the process that enables a learner to achieve a goal through guided efforts (Wood et al., 1976). Scaffolding efforts typically focus on diversifying course content and difficulty (Saule, 2018; Dorodchi et al., 2020), however, scaffolding can also take the form of a conversation. Socratic Questioning (SQ), also referred to as guided inquiry, folds under the theory of scaffolding (Wood et al., 1976; Reiser, 2004) where a more knowledgeable person helps a learner solve a problem that is beyond their zone of proximal development (Quintana et al., 2004; Vygotsky, 2012) by interjecting with questions to guide the student towards a solution. Wood (1994) analyzed conversations in a math classroom and proposed two distinct types of questioning. The first is *funneling*, which aims to guide a learner using a set of questions toward the solution. The second is *focusing*, which draws a learner's attention to important aspects of a problem (Wood, 1994). *Focusing* questions can also probe a student to reflect and articulate their own thinking (National Council of Teachers of Mathematics, 2014; Alic et al., 2022).

Students can complete a programming exercise but still struggle to explain their own program (Lehtinen et al., 2021). To remedy this, Tamang et al. (2021) showed that using the Socratic method to guide students in explaining their code is effective at inducing learning gains in code comprehension tasks. To the best of our knowledge, the impact of Socratic questioning on learning outcomes when guiding student debugging has not been explored yet. In this work, we create Socratic conversations between an instructor and a student where the instructor aims at guiding the student towards fixing a bug in their code using both *funneling* and *focusing* questions while limiting instructor utterances that provide information or facts related to fixing the bug.

▶ **AI for Programming Education and Dialogue Tutoring Systems.** Prior work in AI for programming education includes intelligent tutoring systems (ITS) and learning support systems for programming courses. Learning support systems provide automated feedback on student code submissions and generate programming exercises, unit tests, and code explanations (Wu et al., 2021; Sarsa et al., 2022). Most ITS models rely on methods predating recent developments in large language models (Crow et al., 2018; Mousavinasab et al., 2021), such as action-rules, Bayesian networks, and Fuzzy rules-based systems (Costello, 2012; Butz et al., 2006; Chrysafiadi and Virvou, 2012). Some work has been done in building automatic Socratic tutoring systems, but the Socratic utterances are predefined and manually specified for each exercise, limiting their generalizability (Alshaikh et al., 2020b,a). Existing systems do not propose learning-centered conversational assistants that can generalize to unseen programming problems or focus on using Socratic questions as the main form of interaction with the learner. Automatically scaffolding learning content is important for personalized learning. Research by Kim et al. (2018) has shown that computer-based scaffolding techniques, such as hints, have a moderate impact on student learning in STEM education, paving the way for technologies to assist in the learning process. One such approach, proposed by Shridhar et al. (2022), involves automatically generating funneling Socratic sub-questions for a given math word problem using a T5 language model (Raffel et al., 2020) fine-tuned with reinforcement learning. Similarly, Tyen et al. (2022) introduce a re-ranking-based decoding strategy for language models, which adjusts the difficulty level of a chatbot to meet the needs of learners studying English as a new language.

▶ **Hint Generation.** With the goal of assisting students with programming exercises, recent work has proposed an array of techniques to automatically generate hints to guide novices by providing instant and relevant feedback to correct programming mistakes and advance through exercises (McBroom et al., 2021). Automated hint generation systems use various approaches including extracting common bugs and scaling up instructor feedback to the common bugs (Lee et al., 2018), extracting patterns from peer data (Iii et al., 2014; Lazar et al., 2017), and generating custom solution paths (Rivers and Koedinger, 2017) which typically generalize to unseen code states within an exercise. A super-bug is where a student incorrectly "attributes foresightedness" to the written program where the program executes beyond the information given or the student assumes there is more functionality in the written code than what was written (Pea, 1986). Fragile knowledge is broken down into four categories: missing knowledge where necessary knowledge has not been acquired, inert knowledge where the student has acquired the necessary knowledge but fails to retrieve it, misplaced knowledge where knowledge is used in the wrong context, and conglomerated knowledge where knowledge is misused by combining two or more known structures incorrectly (Perkins and Martin, 1986). Bugs caused by knowledge breakdowns where a student has a misconception are the most time-consuming to fix. For a survey on student misconceptions when learning programming the reader is referred to (Qian and Lehman, 2017).

► **Tutoring Dialogue Corpora.** Prior work in curating corpora of tutoring dialogues between an instructor and a learner includes the CIMA corpus focused on tutoring English speakers to learn Italian (Stasaski et al., 2020). Similarly, for learning English, the Teacher-Student Chatroom Corpus (TSCC), curates up to 260 chatroom dialogues between an experienced teacher and an English learner (Caines et al., 2020, 2022). TSCC was annotated according to the Self-Evaluation of Teacher Talk framework (Walsh, 2006) which includes: Enquiry (where the learner asks a question), Display Question (a question to which the teacher knows the answer), Form-focused feedback, and Instruction. Demszky et al. (2021) release a conversational corpus between math teachers and learners composed of 2,246 utterance exchanges along with annotations on teacher uptake where the teacher builds on what the student has said such as acknowledgment and rephrasing. Chen et al. (2011) examine computer science tutoring conversations and classify tutor utterances into 4 categories: The first category is *Direct Procedural Instructions*, in which the tutor directly tells the student what task to perform. The second category is *Direct Declarative Instruction*, where the tutor provides facts about the domain or problem. The third category is *Prompts*, in which the tutor attempts to elicit a contribution from the student, and the last category is *Feedback* where the tutor affirms or rejects a step a student has completed. One interesting phenomenon observed in the corpus is tutors using analogies to communicate data structures concepts such as using Legos as an analogy to explain stacks (Alizadeh et al., 2015). Prior work focuses on building corpora of tutoring dialogues that contain instructor teaching, and tutorials. There seems to be limited work on building corpora where the instructor's role is limited to guiding the student to discover the bug and any necessary knowledge to fix it on their own using Socratic questioning.

► **Evaluating the Educational Abilities of Language Models.** Tack and Piech (2022) propose using pairwise comparison tests to compare generated responses by BlenderBot (Roller et al., 2021) and GPT-3 (Brown et al., 2020) and find that both language models perform significantly worse than real teachers on understanding a student, helping a student, and speaking like a teacher on the TSCC (Caines et al., 2020, 2022), and the Uptake (Demszky et al., 2021) corpora which focus on English and Mathematics tutoring respectively.

6 Conclusion & Limitations

This paper presents a dataset of expert-curated Socratic conversations where instructors assist novice programmers in fixing buggy solutions to simple computational problems. The dataset serves as a benchmark for evaluating the Socratic debugging capabilities of LMs. While GPT-4 outperforms GPT-3.5, its precision, and recall remain below human expert levels (70.0), highlighting the need for further research. We find that GPT-family language models may generate repetitive and irrelevant Socratic utterances that could mislead learners. The utterances may also appear too early in the conversation, causing confusion, and can be overly direct, potentially diminishing learning outcomes. Study limitations include: The automatic metrics are limited in capturing the correctness, helpfulness, and relevance of a Socratic utterance, and the benchmark dataset may not represent all common novice misconceptions. Moreover, the manual evaluation is limited to 5 dialogues and could be expanded, but this process is highly time-consuming.

Acknowledgements

We would like to thank Sandra Wiktor, Anusha Reddy, Justin Smith, and Qiong Cheng for all their time and effort in contributing dialogues to the benchmark dataset. We also acknowledge Ilan Aktanova and Frank Garcia for their contributions to the programming exercises used in the dataset and Abraham Sanders for his discussions related to dialogue system evaluations. This research was partly supported by the United States Air Force (USAF) under Contract No. FA8750-21-C-0075. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the USAF.

References

Sterling Alic, Dorottya Demszky, Zid Mancenido, Jing Liu, Heather Hill, and Dan Jurafsky. 2022. Computationally identifying funneling and focusing questions in classroom discourse. *BEA 2022*, page 224.

Mehrdad Alizadeh, Barbara Di Eugenio, Rachel Harsley, Nick Green, Davide Fossati, and Omar AlZoubi. 2015. A study of analogy in computer science tutorial dialogues. *Trees*, 53(19.2):1–6.

Zeyad Alshaikh, Lasagn Tamang, and Vasile Rus. 2020a. A socratic tutor for source code comprehension. In *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part II 21*, pages 15–19. Springer.

Zeyad Alshaikh, Lasang Jimba Tamang, and Vasile Rus. 2020b. Experiments with a socratic intelligent tutoring system for source code understanding. In *The Thirty-Third International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*.

Stella Biderman and Edward Raff. 2022. Fooling moss detection with pretrained language models. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 2933–2943, New York, NY, USA. Association for Computing Machinery.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Cory J Butz, Shan Hua, and R Brien Maguire. 2006. A web-based bayesian intelligent tutoring system for computer programming. *Web Intelligence and Agent Systems: An International Journal*, 4(1):77–97.

Andrew Caines, Helen Yannakoudakis, Helen Allen, Pascual Pérez-Paredes, Bill Byrne, and Paula Buttery. 2022. The teacher-student chatroom corpus version 2: more lessons, new annotation, automatic detection of sequence shifts. In *Proceedings of the 11th Workshop on NLP for Computer Assisted Language Learning*, pages 23–35, Louvain-la-Neuve, Belgium. LiU Electronic Press.

Andrew Caines, Helen Yannakoudakis, Helena Edmondson, Helen Allen, Pascual Pérez-Paredes, Bill Byrne, and Paula Buttery. 2020. The teacher-student chatroom corpus. In *Proceedings of the 9th Workshop on NLP for Computer Assisted Language Learning*, pages 10–20, Gothenburg, Sweden. LiU Electronic Press.

Tracy Camp, W Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation cs: the growth of computer science. *ACM Inroads*, 8(2):44–50.

Lin Chen, Barbara Di Eugenio, Davide Fossati, Stellan Ohlsson, and David Cosejo. 2011. Exploring effective dialogue act sequences in one-on-one computer science tutoring dialogues. In *Proceedings of the Sixth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 65–75, Portland, Oregon. Association for Computational Linguistics.

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

Konstantina Chrysafiadi and Maria Virvou. 2012. Evaluating the integration of fuzzy logic into the student model of a web-based learning environment. *Expert systems with applications*, 39(18):13127–13134.

Robert Costello. 2012. *Adaptive intelligent personalised learning (aipl) environment*. Ph.D. thesis.

Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 53–62.

Dorottya Demszky, Jing Liu, Zid Mancenido, Julie Cohen, Heather Hill, Dan Jurafsky, and Tatsunori Hashimoto. 2021. Measuring conversational uptake: A case study on student-teacher interactions. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1638–1653, Online. Association for Computational Linguistics.

Mohsen M Dorodchi, Nasrin Dehbozorgi, Aileen Benedict, Erfan Al-Hossami, and Alexandria Benedict. 2020. Scaffolding a team-based active learning course to engage students: A multidimensional approach. In *2020 ASEE Virtual Annual Conference Content Access*.

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.

Zvi Galil. 1986. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys (CSUR)*, 18(1):23–38.

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas Van Binsbergen. 2017. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100.

Prakhar Gupta, Shikib Mehri, Tiancheng Zhao, Amy Pavel, Maxine Eskenazi, and Jeffrey P Bigham. 2019. Investigating evaluation of open-domain dialogue systems with human generated multiple references. In *Proceedings of the 20th Annual SIGdial Meeting on Discourse and Dialogue*, pages 379–391.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Sebastian Hobert. 2019. Say Hello to 'Coding Tutor'! Design and Evaluation of a Chatbot-based Learning System Supporting Students to Learn to Program. *ICIS 2019 Proceedings*.

Barry Peddycord Iii, Andrew Hicks, and Tiffany Barnes. 2014. Generating hints for programming problems using intermediate output. In *Educational Data Mining 2014*. Citeseer.

Johan Jeuring, L. Thomas van Binsbergen, Alex Gerdes, and Bastiaan Heeren. 2014. Model solutions and properties for diagnosing student programs in ask-elle. In *Proceedings of the Computer Science Education Research Conference*, CSERC '14, page 31–40, New York, NY, USA. Association for Computing Machinery.

Nam Ju Kim, Brian R Belland, and Andrew E Walker. 2018. Effectiveness of computer-based scaffolding in the context of problem-based learning for STEM education: Bayesian meta-analysis. *Educational Psychology Review*, 30:397–429.

Timotej Lazar, Martin Možina, and Ivan Bratko. 2017. Automatic extraction of ast patterns for debugging student programs. In *Artificial Intelligence in Education: 18th International Conference, AIED 2017, Wuhan, China, June 28–July 1, 2017, Proceedings 18*, pages 162–174. Springer.

Victor CS Lee, Yuen-Tak Yu, Chung Man Tang, Tak-Lam Wong, and Chung Keung Poon. 2018. Vida: A virtual debugging advisor for supporting learning in computer programming courses. *Journal of Computer Assisted Learning*, 34(3):243–258.

Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students struggle to explain their own program code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pages 206–212.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Mary Lou Maher, Celine Latulipe, Heather Lipford, and Audrey Rorrer. 2015. Flipped classroom strategies for cs education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 218–223.

Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A survey of automated programming hint generation: The hints framework. *ACM Computing Surveys (CSUR)*, 54(8):1–27.

Diba Mirza, Phillip T Conrad, Christian Lloyd, Ziad Matni, and Arthur Gatin. 2019. Undergraduate teaching assistants in computer science: a systematic literature review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 31–40.

Elham Mousavinasab, Nahid Zarifsanaiey, Sharareh R. Niakan Kalhori, Mahnaz Rakhshan, Leila Keikha, and Marjan Ghazi Saeedi. 2021. Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods. *Interactive Learning Environments*, 29(1):142–163.

National Council of Teachers of Mathematics. 2014. *Principles to actions: Ensuring mathematical success for all*. NCTM, National Council of Teachers of Mathematics, Reston, VA.

OpenAI. 2022. Introducing chatgpt.

OpenAI. 2023. Gpt-4 technical report. *arXiv*.

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, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Roy D Pea. 1986. Language-independent conceptual "bugs" in novice programming. *Journal of educational computing research*, 2(1):25–36.

David N Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229.

Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24.

Chris Quintana, Brian J. Reiser, Elizabeth A. Davis, Joseph Krajcik, Eric Fretz, Ravit Golan Duncan, Eleni Kyza, Daniel Edelson, and Elliot Soloway. 2004. A scaffolding design framework for software to support science inquiry. *Journal of the Learning Sciences*, 13(3):337–386.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67.

Brian J Reiser. 2004. Scaffolding complex learning: The mechanisms of structuring and problematizing student work. *The Journal of the Learning sciences*, 13(3):273–304.

Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27:37–64.

Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Eric Michael Smith, Y-Lan Boureau, and Jason Weston. 2021. Recipes for building an open-domain chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 300–325, Online. Association for Computational Linguistics.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ICER '22, page 27–43, New York, NY, USA. Association for Computing Machinery.

Erik Saule. 2018. Experiences on teaching parallel and distributed computing for undergraduates. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 361–368.

Kumar Shridhar, Jakub Macina, Mennatallah El-Assady, Tanmay Sinha, Manu Kapur, and Mrinmaya Sachan. 2022. Automatic generation of socratic subquestions for teaching math word problems. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4136–4149, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Katherine Stasaski, Kimberly Kao, and Marti A Hearst. 2020. Cima: A large open access dialogue dataset for tutoring. In *Proceedings of the Fifteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 52–64.

Anaïs Tack and Chris Piech. 2022. The AI teacher test: Measuring the pedagogical ability of blender and GPT-3 in educational dialogues. In *Proceedings of the 15th International Conference on Educational Data Mining*, pages 522–529, Durham, United Kingdom. International Educational Data Mining Society.

Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait Khayi, Priti Oli, and Vasile Rus. 2021. A comparative study of free self-explanations and socratic tutoring explanations for source code comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 219–225, New York, NY, USA. Association for Computing Machinery.

Gladys Tyen, Mark Brenchley, Andrew Caines, and Paula Buttery. 2022. Towards an open-domain chatbot for language practice. In *Proceedings of the 17th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2022)*, pages 234–249, Seattle, Washington. Association for Computational Linguistics.

Lev S Vygotsky. 2012. *Thought and language*. MIT press.

Steve Walsh. 2006. *Investigating classroom discourse*. Routledge.

Judith D Wilson. 1987. A socratic approach to helping novice programmers debug programs. *ACM SIGCSE Bulletin*, 19(1):179–182.

David Wood, Jerome S Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Child Psychology & Psychiatry & Allied Disciplines*.

Terry Wood. 1994. Patterns of interaction and the culture of mathematics classrooms. In *Cultural perspectives on the mathematics classroom*, pages 149–168. Springer.

Mike Wu, Noah Goodman, Chris Piech, and Chelsea Finn. 2021. Prototransformer: A meta-learning approach to providing student feedback. *arXiv preprint arXiv:2107.14035*.

Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. 2016. Expanding computer science education in schools: understanding teacher experiences and challenges. *Computer Science Education*, 26(4):235–254.

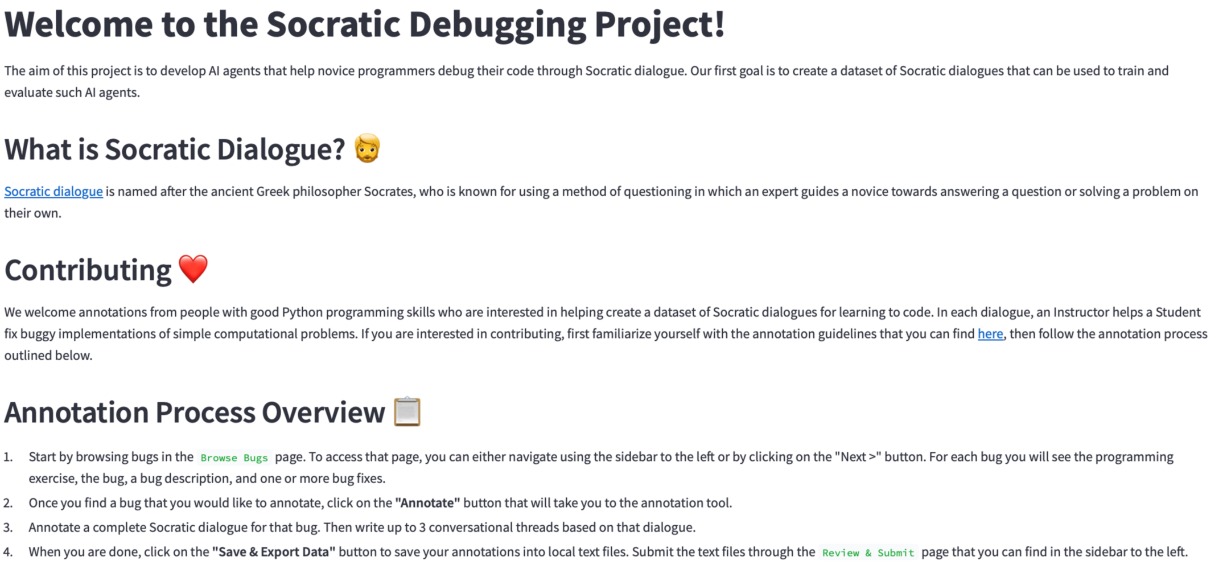
Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

A Data Contribution Web Application

We developed a 7-page data contribution web application tool using the Streamlit Python library[6] to collect dialogues and code snapshots. The application loads a repository of programming problems and bugs from a Google Spreadsheet using the Google Spreadsheet API through the gsheetsdb Python library[7]. The web app consists of the following pages:

- **Getting Started:** This page (Figure 1) orients the users on the task and provides a link to the guidelines document.
- **Browse Bugs:** Contributors browse and select a bug (Figure 2) to create a Socratic dialogue for.
- **4 Data Contribution Pages:** These pages contain a code editor and a chat area (Figure 3) where contributors create an initial conversation and up to 3 conversational threads.
- **Review and Submit:** This page (Figure 7) allows contributors to review their work and submit the exported dialogues for review.

During the data contribution process, contributors can add main and alternative utterances, undo added utterances or code snapshots, and edit the chat history text area and code in the code editor. When the contributor edits the code in the Code Editor, they can choose to compile and run the code within the web application and they can also add a code snapshot to the chat history by clicking the "Add Code to Chat History" button (Figure 4). Once the bug has been fixed, the contributor compiles and runs the code in the Code Editor, as demonstrated in Figure 5. Contributors can then use the import and export buttons shown in Figure 6 to save their work. The export button generates a standardized form of the dialogue and code states, while the import button allows contributors to load previously exported dialogues back into the tool. After completing their data contribution, contributors submit the exported dialogues for review.



Welcome to the Socratic Debugging Project!

The aim of this project is to develop AI agents that help novice programmers debug their code through Socratic dialogue. Our first goal is to create a dataset of Socratic dialogues that can be used to train and evaluate such AI agents.

What is Socratic Dialogue? 🧔

Socratic dialogue is named after the ancient Greek philosopher Socrates, who is known for using a method of questioning in which an expert guides a novice towards answering a question or solving a problem on their own.

Contributing ❤️

We welcome annotations from people with good Python programming skills who are interested in helping create a dataset of Socratic dialogues for learning to code. In each dialogue, an Instructor helps a Student fix buggy implementations of simple computational problems. If you are interested in contributing, first familiarize yourself with the annotation guidelines that you can find here, then follow the annotation process outlined below.

Annotation Process Overview 📋

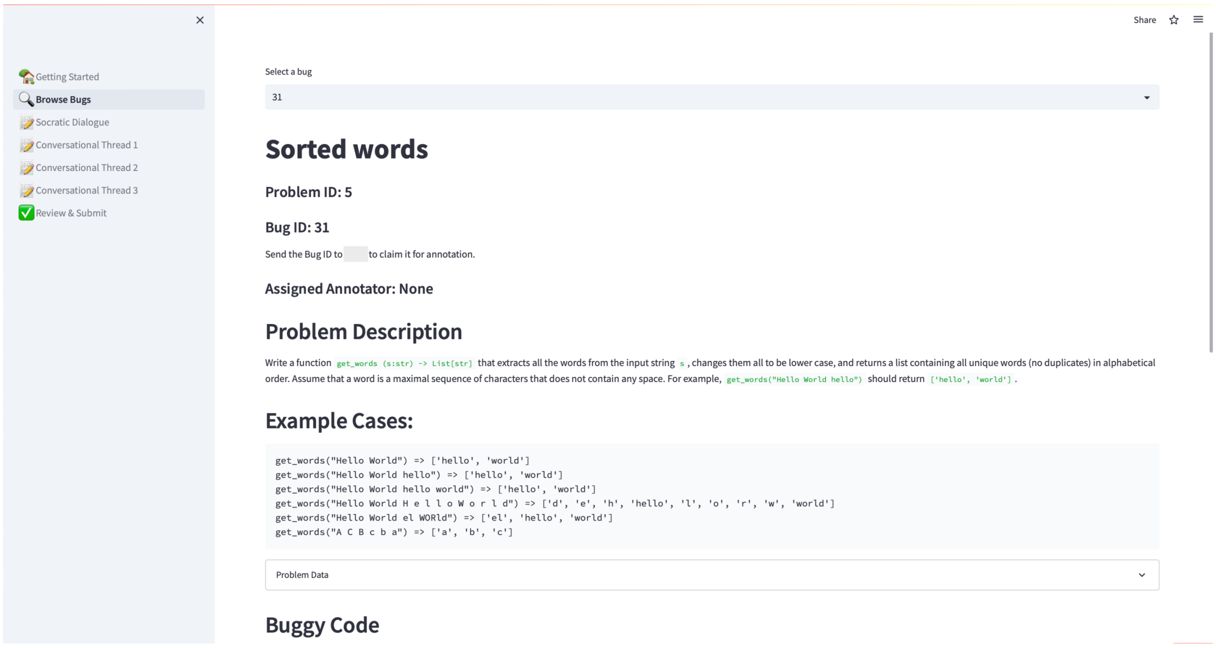
1. Start by browsing bugs in the `Browse Bugs` page. To access that page, you can either navigate using the sidebar to the left or by clicking on the "Next >" button. For each bug you will see the programming exercise, the bug, a bug description, and one or more bug fixes.
2. Once you find a bug that you would like to annotate, click on the **"Annotate"** button that will take you to the annotation tool.
3. Annotate a complete Socratic dialogue for that bug. Then write up to 3 conversational threads based on that dialogue.
4. When you are done, click on the **"Save & Export Data"** button to save your annotations into local text files. Submit the text files through the `Review & Submit` page that you can find in the sidebar to the left.

Figure 1: Screenshot of the web application's Getting Started page where contributors get familiarized with the task and go through the guidelines document.

---

[6] https://streamlit.io/

[7] https://github.com/betodealmeida/gsheets-db-api



Select a bug

31

Sorted words

Problem ID: 5

Bug ID: 31

Send the Bug ID to [redacted] to claim it for annotation.

Assigned Annotator: None

Problem Description

Write a function `get_words (s:str) -> List[str]` that extracts all the words from the input string `s`, changes them all to be lower case, and returns a list containing all unique words (no duplicates) in alphabetical order. Assume that a word is a maximal sequence of characters that does not contain any space. For example, `get_words("Hello World hello")` should return `['hello', 'world']`.

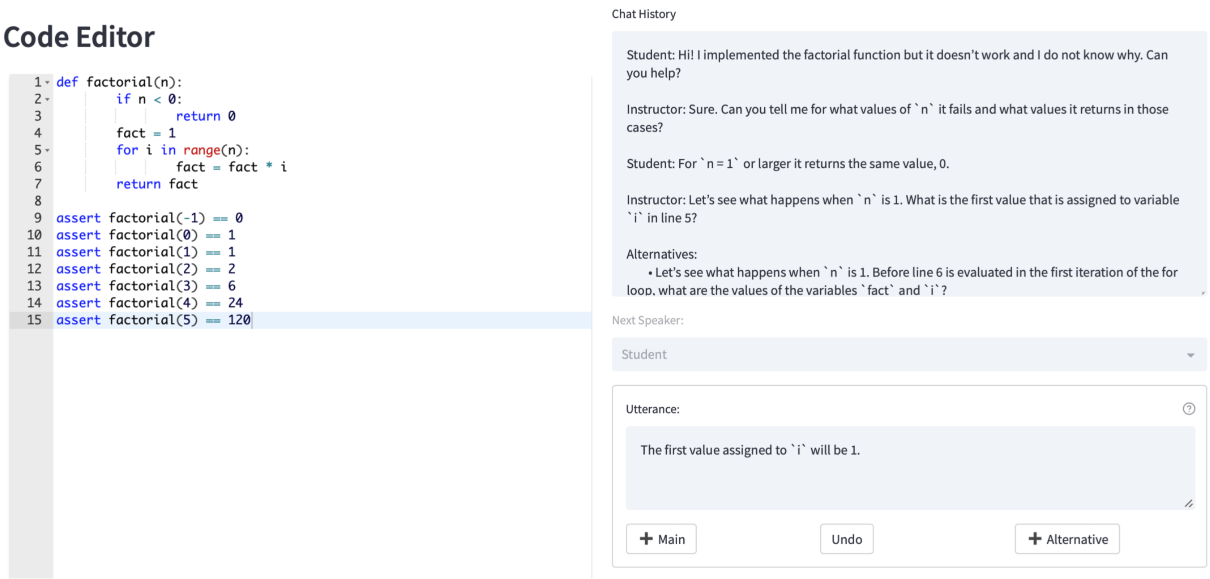
Example Cases:

```
get_words("Hello World") => ['hello', 'world']
get_words("Hello World hello") => ['hello', 'world']
get_words("Hello World hello world") => ['hello', 'world']
get_words("Hello World H e l l o W o r l d") => ['d', 'e', 'h', 'hello', 'l', 'o', 'r', 'w', 'world']
get_words("Hello World el WORld") => ['el', 'hello', 'world']
get_words("A C B c b a") => ['a', 'b', 'c']
```

Problem Data

Buggy Code

Figure 2: Screenshot of the interface. Contributors first browse a repository of bugs created from a set of programming problems. Each bug is displayed with the problem description, test cases, a buggy code, the bug description, and bug fixes. Contributors select a bug to create a dialogue for.



Code Editor

```
1  def factorial(n):
2      if n < 0:
3          return 0
4      fact = 1
5      for i in range(n):
6          fact = fact * i
7      return fact
8
9  assert factorial(-1) == 0
10 assert factorial(0) == 1
11 assert factorial(1) == 1
12 assert factorial(2) == 2
13 assert factorial(3) == 6
14 assert factorial(4) == 24
15 assert factorial(5) == 120
```

Chat History

Student: Hi! I implemented the factorial function but it doesn't work and I do not know why. Can you help?

Instructor: Sure. Can you tell me for what values of `n` it fails and what values it returns in those cases?

Student: For `n = 1` or larger it returns the same value, 0.

Instructor: Let's see what happens when `n` is 1. What is the first value that is assigned to variable `i` in line 5?

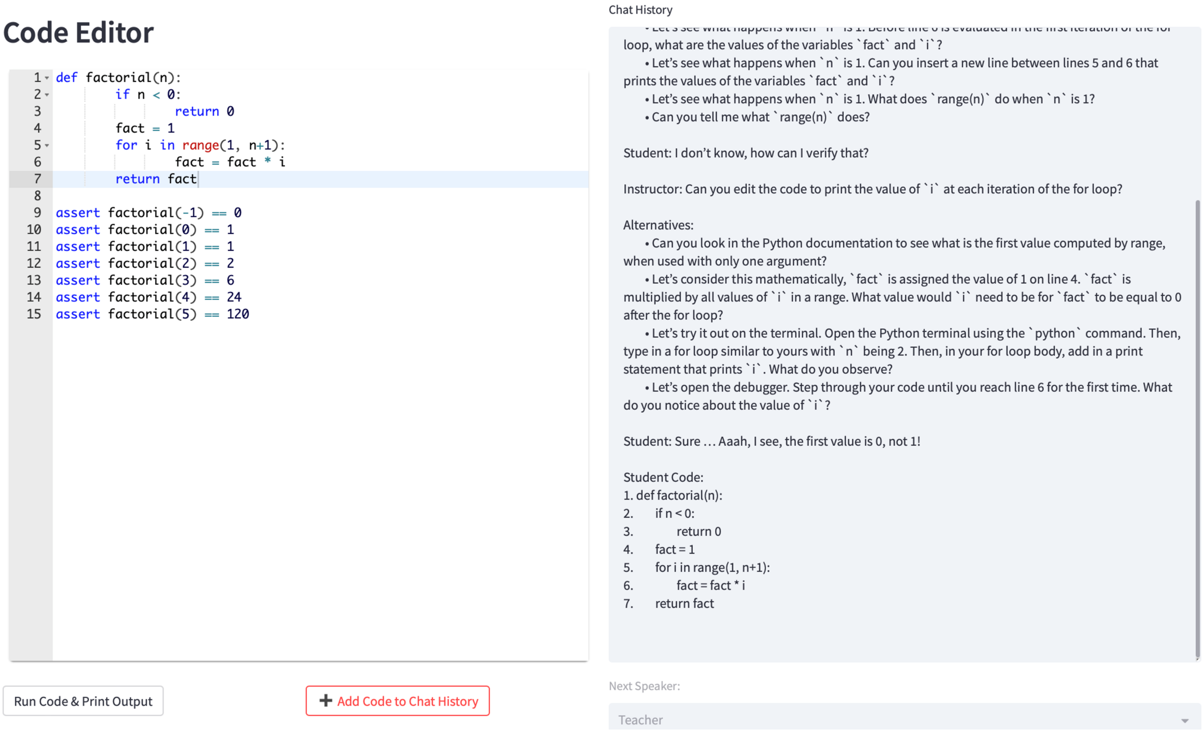
Alternatives:

- Let's see what happens when `n` is 1. Before line 6 is evaluated in the first iteration of the for loop, what are the values of the variables `fact` and `i`?

Next Speaker: Student

Utterance: The first value assigned to `i` will be 1.

Figure 3: Screenshot of the tool used to collect dialogues and code snapshots. Contributors are able to add a main utterance, an alternative utterance, and undo an adding utterance or a code snapshot. Additionally, the chat history text area is editable.



```
 1  def factorial(n):
 2          if n < 0:
 3                  return 0
 4          fact = 1
 5          for i in range(1, n+1):
 6                  fact = fact * i
 7          return fact
 8
 9  assert factorial(-1) == 0
10  assert factorial(0) == 1
11  assert factorial(1) == 1
12  assert factorial(2) == 2
13  assert factorial(3) == 6
14  assert factorial(4) == 24
15  assert factorial(5) == 120
```

loop, what are the values of the variables `fact` and `i`?

- Let's see what happens when `n` is 1. Can you insert a new line between lines 5 and 6 that prints the values of the variables `fact` and `i`?
- Let's see what happens when `n` is 1. What does `range(n)` do when `n` is 1?
- Can you tell me what `range(n)` does?

Student: I don't know, how can I verify that?

Instructor: Can you edit the code to print the value of `i` at each iteration of the for loop?

Alternatives:

- Can you look in the Python documentation to see what is the first value computed by range, when used with only one argument?
- Let's consider this mathematically, `fact` is assigned the value of 1 on line 4. `fact` is multiplied by all values of `i` in a range. What value would `i` need to be for `fact` to be equal to 0 after the for loop?
- Let's try it out on the terminal. Open the Python terminal using the `python` command. Then, type in a for loop similar to yours with `n` being 2. Then, in your for loop body, add in a print statement that prints `i`. What do you observe?
- Let's open the debugger. Step through your code until you reach line 6 for the first time. What do you notice about the value of `i`?

Student: Sure ... Aaah, I see, the first value is 0, not 1!

Student Code:

```
1. def factorial(n):
2.     if n < 0:
3.         return 0
4.     fact = 1
5.     for i in range(1, n+1):
6.         fact = fact * i
7.     return fact
```

Figure 4: Screenshot of the tool adding a code snapshot by clicking the Add Code to Chat History button.

```
 1  def factorial(n):
 2          if n < 0:
 3                  return 0
 4          fact = 1
 5          for i in range(1, n+1):
 6                  fact = fact * i
 7          return fact
 8
 9  assert factorial(-1) == 0
10  assert factorial(0) == 1
11  assert factorial(1) == 1
12  assert factorial(2) == 2
13  assert factorial(3) == 6
14  assert factorial(4) == 24
15  assert factorial(5) == 120
```

```
Program passes all unit tests!
```

Figure 5: Screenshot of the tool compiling and running the code in the Code Editor after the bug has been fixed.

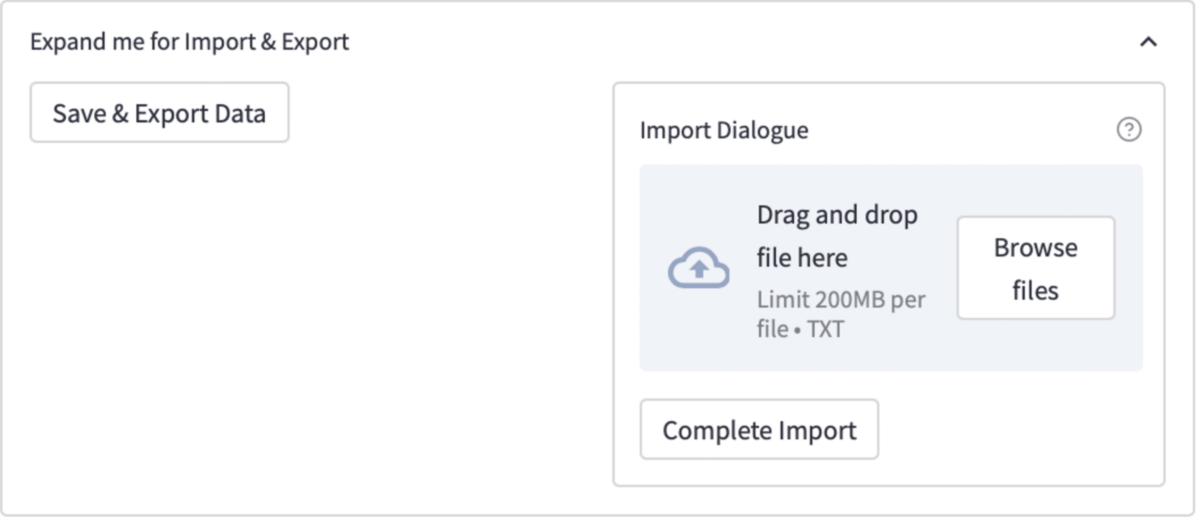
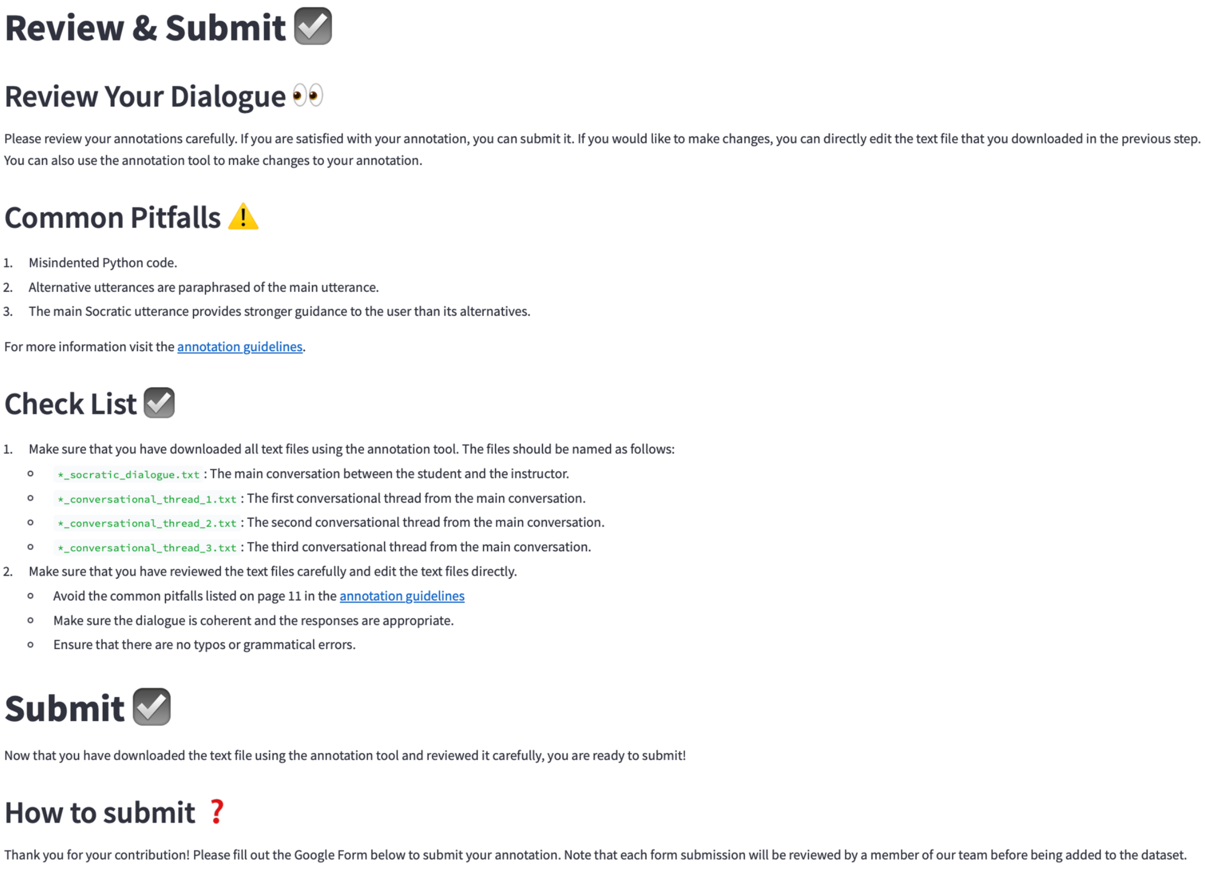


Figure 6: Screenshot of the tool's import and export buttons. Upon completing a dialogue contributors use the export button to export the dialogue and code states into a standardized form. Additionally, contributors can import any dialogue exported from this tool using the import button.



Review & Submit ☑

Review Your Dialogue 👀

Please review your annotations carefully. If you are satisfied with your annotation, you can submit it. If you would like to make changes, you can directly edit the text file that you downloaded in the previous step. You can also use the annotation tool to make changes to your annotation.

Common Pitfalls ⚠

1. Misindented Python code.
2. Alternative utterances are paraphrased of the main utterance.
3. The main Socratic utterance provides stronger guidance to the user than its alternatives.

For more information visit the annotation guidelines.

Check List ☑

1. Make sure that you have downloaded all text files using the annotation tool. The files should be named as follows:
   - `*_socratic_dialogue.txt` : The main conversation between the student and the instructor.
   - `*_conversational_thread_1.txt` : The first conversational thread from the main conversation.
   - `*_conversational_thread_2.txt` : The second conversational thread from the main conversation.
   - `*_conversational_thread_3.txt` : The third conversational thread from the main conversation.
2. Make sure that you have reviewed the text files carefully and edit the text files directly.
   - Avoid the common pitfalls listed on page 11 in the annotation guidelines
   - Make sure the dialogue is coherent and the responses are appropriate.
   - Ensure that there are no typos or grammatical errors.

Submit ☑

Now that you have downloaded the text file using the annotation tool and reviewed it carefully, you are ready to submit!

How to submit ❓

Thank you for your contribution! Please fill out the Google Form below to submit your annotation. Note that each form submission will be reviewed by a member of our team before being added to the dataset.

Figure 7: Screenshot of the web application's Review & Submit page where contributors are instructed to review their data contribution and submit their exported version.

B Language Model Prompt

This section describes the prompt template that was used for language models in this paper. {{text}} denotes a data point from the benchmark dataset. The steering prompt was adapted from the GPT-4 blog post[8]. The '1.' is added at the end of the instruction to prompt the language model to generate an itemized list of utterances that can then be parsed.

Steering Prompt:

> You are a tutor that always responds in the Socratic style. You *never* give the student the answer, but always try to ask just the right question to help them learn to think for themselves. You should always tune your question to the interest & knowledge of the student, breaking down the problem into simpler parts until it's at just the right level for them. Socratic utterances are utterances that guide the user and do not give them the solution directly. In each of your responses, provide a comprehensive list of Socratic responses that you can give to the user to help them solve the problem on their own, based on the conversation so far.

Prompt:

```
<problem>
{{Problem Description}}
</problem>
<bug_code>
{{Buggy Code}}
</bug_code>
<bug_desc>
{{Bug Description}}
</bug_desc>
<bug_fixes>
{{Bug Fixes}}
</bug_fixes>
<unit_tests>
{{Unit Tests}}
</unit_tests>

User: {{User Turn 1}}
Assistant: {{Assistant Turn 1}}
...
User: {{User Turn N}}
<code>
{{Code State at Turn N}}[a]
</code>

Respond to the user with all possible distinct Socratic utterances that guide the user to discover and fix the bug described between '<bug_desc>' and '</bug_desc>'. Student code is written between '<code>' and '</code>' throughout the conversation. Utterances that have the same meaning but different words are considered duplicates. Assume that the student has run the test cases.
1.
```

[a] Included only if turn N has a code state.

---

[8] https://openai.com/research/gpt-4