Need a Programming Exercise Generated in Your Native Language? ChatGPT's Got Your Back: Automatic Generation of Non-English Programming Exercises Using OpenAI GPT-3.5

Mollie Jordan
mcjordan@ncsu.edu
North Carolina State University

Kevin Ly
k9ly@ucsd.edu
University of California, San Diego

Adalbert Gerald Soosai Raj
asoosairaj@ucsd.edu
University of California, San Diego

ABSTRACT

Large language models (LLMs) like ChatGPT are changing computing education and may create additional barriers to those already faced by non-native English speakers (NNES) learning computing. We investigate an opportunity for a positive impact of LLMs on NNES through multilingual programming exercise generation. Following previous work with LLM exercise generation in English, we prompt OpenAI GPT-3.5 in 4 natural languages (English, Tamil, Spanish, and Vietnamese) to create introductory programming problems, sample solutions, and test cases. We evaluate these problems on their sensibility, readability, translation, sample solution accuracy, topicality, and cultural relevance. We find that problems generated in English, Spanish, and Vietnamese are largely sensible, easily understood, and accurate in their sample solutions. However, Tamil problems are mostly non-sensible and have a much lower passing test rate, indicating that the abilities of LLMs for problem generation are not generalizable across languages. Our analysis suggests that these problems could not be given verbatim to students, but with minimal effort, most errors can be fixed. We further discuss the benefits of these problems despite their flaws, and their opportunities to provide personalized and culturally relevant resources for students in their native languages.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**.

KEYWORDS

non-native English speakers; problem generation; large language models; introductory programming

ACM Reference Format:
Mollie Jordan, Kevin Ly, and Adalbert Gerald Soosai Raj. 2024. Need a Programming Exercise Generated in Your Native Language? ChatGPT's Got Your Back: Automatic Generation of Non-English Programming Exercises Using OpenAI GPT-3.5. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630897

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0423-9/24/03.
https://doi.org/10.1145/3626252.3630897

1 INTRODUCTION

English is the primary language of Computer Science (CS) in both education and industry. The most common programming languages are based in English, their documentation pages are in English, and Q&A sites (such as Stack Overflow) and other resources are also in English. Thus English fluency plays a large part in the ability to learn programming. Research has increased on the barriers **Non-Native English Speakers (NNES)** face in computing education, identifying obstacles such as lower confidence levels [26], increased anxiety related to programming assignments [13], difficulty choosing and understanding keywords and identifiers [15, 25], and an overall lack of resources in non-English languages [25, 31].

With the rise in artificial intelligence (AI) and large language models (LLM), it is unknown how their integration into CS education will impact NNES. LLMs have already been shown to perform well on introductory programming assignments [19, 24, 42, 43] and provide mostly thorough and correct code explanations [16, 32, 40]. Researchers indicate that these abilities point toward a shift in programming education to emphasize code reading and analysis over traditional writing from scratch [16, 20, 24]. As LLMs have been trained mostly on English data, these coming shifts may introduce additional barriers for NNES students. However, we believe LLMs could be used to positively impact NNES by addressing the lack of programming educational resources in non-English languages.

Sarsa et al. investigated the use of OpenAI Codex to generate introductory programming problems in English and found that problems were mostly sensible, novel, and some ready to use as-is [40]. We further this research by generating problems in 4 natural languages, English, Spanish, Vietnamese, and Tamil, using OpenAI GPT-3.5 to explore how large language models (LLM) can be used to create programming exercises for non-native English speakers in their native languages. In this study we ask the following research questions:

(1) How correct are the sample solutions and tests generated by GPT-3.5 and to what extent do the tests cover the generated code?
(2) To what extent are the exercises generated by GPT-3.5 sensible, easily understood, accurate to the prompt and culturally relevant?

2 RELATED WORK

2.1 NNES Learning Computer Programming

Much research concerning experiences of NNES in CS education has focused on the stress and confidence levels of students. One study conducted in CS1 courses in China and Saudi Arabia found a significant negative low correlation between overall Foreign Language Classroom Anxiety and students' academic performance on programming assignments [13]. Another study explored stress and sense of belonging among NNES in 3 introductory programming courses. While there was no significant disparity in stress between NNES and native English speakers (NES), NNES were more likely to report "self-doubt" and "embarrassment asking for help" [12]. Guzman et al. found that while NNES and NES received similar grades, NNES had significantly lower confidence levels than NES in 2 of 4 introductory CS courses at UC San Diego. In 3 courses NNES spent significantly more time studying outside of class [26].

An international survey by Guo identified barriers faced by NNES learning computing [25]. 840 responses (78% were NNES) were analyzed representing 74 native languages and 86 countries. The most commonly-reported barrier (34% of responses) was reading English instructional materials. NNES often said they had to rely on English materials since it was hard to find good materials in their language. Feijóo-García et al. also compared English and Spanish speakers' performance and experience learning using the programming educational tool Scratch (in English), and while there wasn't much difference in performance, most Spanish speakers felt they would do better if Scratch was in their native language [23].

2.2 Generative AI and Computing Education

The CS industry has seen a "boom" in the development of LLMs, AI models that generate code, provide code explanations and more. Their open access to students is bringing rapid shifts to CS education. Several studies have tested their performance on introductory programming assignments. OpenAI Codex scored 78.5% and 78.0% on the first two exams in a CS1 course, scoring in the top quartile of the class and better than the average student [24]. OpenAI's GPT-3.5 scored between 56%-67% on 3 beginner-to-intermediate Python coding classes [42], and GPT-4 improved this further, scoring 84.1% overall [41]. When given incorrect student code, GPT-3 correctly suggested improvements for 61.95% of submissions [14], and GPT-3.5 identified at least one issue in buggy code 90% of the time and all issues 57% of the time (when prompted in English) [27].

The remarkable abilities of LLMs bring advantages and disadvantages to CS education. Opportunities include low-cost generation of practice problem solutions for students [16, 20, 24, 36], use as a pair programmer [18, 44], easy access to code explanations [20, 32, 40], and alleviation of programmers' writer's block [16, 20, 37]. Challenges include threats to academic integrity [16, 20, 24, 30, 34, 37], licensing issues [16], biased/insecure generated code [16], and over-reliance by students [16, 24, 30, 37, 41]. However, Kazemitabaar et al. found students who learned programming topics with and without Codex scored similarly on retention tests, so over-reliance may not be as harmful as predicted [29].

2.3 Generation of Educational Resources for NNES using LLMs

Recent research has explored using LLMs for programming problem generation. This is better than simple translation from existing problems which may create incorrect wording and doesn't consider programming keywords that must not be translated. Sarsa et al. examined the sensibleness, novelty, topicality, and readiness for use of programming exercises generated by OpenAI Codex [40]. They primed Codex with 2 common introductory programming problems, the speeding and currency converter problems, and prompted it to create 240 similar exercises with different contexts. They conducted an automated evaluation of the 240 generated problems and manually evaluated 120/240. From the manual evaluation 75.0% were sensible, 81.8% novel, and 76.7% had a matching sample solution. 70.8% of all 240 exercises had auto-extractable tests, but only 30.9% passed their tests. These results indicated most exercises generated by Codex couldn't be given verbatim to students, but could be a much easier starting point for instructors. Parsers or other alterations (such as the "robosourcing" process of automated and manual review [21]) could make the exercises ready to use [40].

Sarsa et al. briefly explored the quality of exercises generated in Finnish, but their only report was the exercises were "generally good" [40]. We address this gap in research by replicating their work and introducing new natural languages into our generation.

3 METHODS

3.1 Prompt Engineering

Our final prompt is seen in Listings 1 and 2. We developed this prompt using OpenAI's ChatGPT web interface [1], then switched to the GPT-3.5 API for problem generation (see 3.2). We modified Sarsa et al.'s original prompt [40], adding an instructional statement to prime the model since it is chat-based unlike Codex, which we couldn't use as it was deprecated at the time of this study. Prompting the two parts separately, following OpenAI's suggestion to split up subtasks [2], gave us more consistent results. We added a natural language heading to the prompt and split Sarsa et al.'s "Keywords" heading into "Context" and "Programming Concepts." The prompt follows GPT best practices including providing examples and using delimiters to indicate distinct parts of input [2].

Listing 1: Instructional statement (given to the model first)

```
I will give you an introductory programming
problem. I will give you the following:

Natural Language
Context
Programming Concepts
Problem Statement
Sample Solution
Tests

I want you to respond with a similar problem that
uses the same programming concepts but with the
new context. Write Problem 2 in (INSERT LANGUAGE)
after I give you Problem 1.
```

Our instructional statement explained the task to ChatGPT before we gave it a modified version of Sarsa et al.'s prompt. We found the following strategies most successful in generating consistent responses in our desired format: writing clear, concise instructions, using the same words throughout instead of synonyms (e.g., only using "problem" to refer to what to generate, not "problem"/"exercise" interchangeably), and using specific, unambiguous language (e.g., using "context" to refer to the problem topic instead of "keywords" which may be interpreted as programming keywords). We came

Listing 2: Prompt (given to the model after the instructional statement in Listing 1)

```
"""Problem 1
--Natural Language--
English
--Context--
Cars
--Programming Concepts--
Function
Parameters
Conditional
--Problem statement--
Write a function called speeding_check that takes
a single parameter speed and prints out "You are
fined for $200" if the speed is above 120, "You
are fined for $100" if the speed is above 100 but
below 120 and otherwise prints "All good, race
ahead".
--Sample solution--
def speeding_check(speed):
    if speed > 120:
        return "You are fined for $200"
    elif speed > 100:
        return "You are fined for $100"
    else: return "All good, race ahead"
--Tests--
class Test(unittest.TestCase):
    def test_speeding_check(self):
        self.assertEquals(speeding_check(100),
            'All good, race ahead')
        self.assertEquals(speeding_check(101),
            'You are fined for $100')
        self.assertEquals(speeding_check(121),
            'You are fined for $200')
"""Problem 2
--Natural Language--
(INSERT LANGUAGE)
--Context--
(INSERT CONTEXT)
--Programming Concepts--
Function
Parameters
Conditional
--Problem Statement--
--Sample Solution--
--Tests--
```

to a largely successful initial version of our prompt, but it did not generate consistently in Tamil (only 17/100 problems generated in Tamil, the others were in English). We thought this was due to Tamil's non-Latin alphabet and the model may not have had much Tamil training data. However, we later directly prompted to write in a specific language (last sentence in Listing 1), after which all problems generated in the right language. We used this finalized instructional statement for our generation and evaluations.

3.2 Problem Generation

We initially chose to generate problems using ChatGPT's web interface [1] because it is easy to use for a teacher/student generating their own exercises. However, the web interface is not conducive to automation, so we automated generation using the ChatCompletions [3] function of OpenAI's `gpt-3.5-turbo` model through the OpenAI API. `gpt-3.5-turbo` is the most capable and cost-effective model in the GPT-3.5 family and is optimized for chat [4]. We chose it because ChatGPT's web interface is based on the GPT-3.5 model.

We generated 100 problems each in 4 languages: 1) English, 2) Spanish, 3) Vietnamese, and 4) Tamil. These languages were chosen because they were spoken by the researchers and represent a variety of language families. The speeding problem (taken from Sarsa et al.) was the only priming problem used to keep other factors constant while changing natural language. We prompted GPT-3.5 to use the same programming concepts as the priming problem so it would create problems with similar structure. 5 contexts were used (books, movies, music, health, relationships), generating 20 problems in each context. We chose these contexts because of their applicability in any language/culture. The temperature of the model (a parameter that controls how random the responses are) was constant at 1. Higher temperatures generate more diverse responses. We kept the default value (1) since this is the middle of the allowed range (0-2), and we wanted creative responses, but similar enough to be tested. This created a total of 400 generated problems (1 priming problem × 20 generations × 5 contexts × 4 languages = 400 total problems).

3.3 Automated Evaluation

Our automated evaluation tested the same criteria as Sarsa et al.: whether the sample solutions could run, whether they passed the generated tests, and to what extent the tests covered the sample solutions [40]. First, we prompted the `gpt-3.5-turbo` model using the OpenAI API. Then, we extracted the sample solution and tests from each response using the "--" in the headings of the responses as a delimiter and put the extracted code into a Python test file. If the generated response was not in our desired format and our parser failed, we manually entered the sample solution/tests. We used pytest [11] to run the tests and generate statement coverage (i.e., the percentage of successfully executed statements in the program).

3.4 Manual Evaluation

We chose 3 problems randomly from each context to manually evaluate in each language (15 problems total per language). We recruited 6 manual evaluators (2 evaluators each of Spanish, Vietnamese, and Tamil) to evaluate the problems because not all researchers were native speakers of these languages. The Tamil speakers were working professionals with 13+ years of experience in the tech industry. The Spanish and Vietnamese speakers were college students in CS programs. The participants were not informed the problems were AI-generated to not influence their responses. Two researchers (native English speakers) evaluated the English problems. The evaluators were given the problems alongside an evaluation table (see Table 1) modified from Sarsa et al.'s original. For each aspect participants chose Yes, No, or Maybe. They were instructed to describe their reasoning if they chose Maybe, and could write notes for any response. See [5] for an evaluation document.

We removed the novelty criteria from our evaluation after we found no similar problems online when searching for 5 problems from each language (1 in each context) on GitHub and Google. Also,

Table 1: Manual assessment rubric used by native speakers to evaluate problems

Aspect	Question	Options	Notes
Sensibleness	Does the problem statement describe a sensible problem?	Yes/No/Maybe	
Readability	Is the problem grammatically correct? Is it easy to read?	Yes/No/Maybe	
Translation	Does the problem statement sound like it was written by a native speaker?	Yes/No/Maybe	
Sample Solution Accuracy	Does the output from the sample solution match what is required from the problem statement? Does it follow all instructions from the problem statement?	Yes/No/Maybe	
Topicality: Programming Concepts	Do the problem statement and sample solution include all of the provided programming concepts (function, parameters, conditional)?	Yes/No/Maybe	
Topicality: Context	Does the problem statement context match the given priming context?	Yes/No/Maybe	
Cultural Relevance	Does the problem (statement, solution, or tests) use examples relevant to the culture of the language?	Yes/No/Maybe	

since Sarsa et al. found high novelty (81.8%) in English, we expected our problems to be even more novel due to the lack of non-English CS educational resources (and thus training data) online.

We used the criteria of **sensibility, sample solution accuracy** (though reworded this title to be more intuitive), and **topicality** (both programming concepts and context) from Sarsa et al.'s original rubric. We added 3 aspects to our evaluation. We added **readability** to see if the problems were easily understood by native speakers. We felt this was separate from sensibility as a problem statement could describe a sensible problem, but have confusing wording. We added **translation** to see how natural the problem statements sounded to native speakers. This is distinct from sensibility and readability as an exercise could be readable and describe a sensible problem, but use unnatural words in that context/language. We also added **cultural relevance** to understand the cultural relevance of the examples in the problems. During prompt engineering, we noticed some sample solutions and tests contained culturally relevant examples, such as famous Spanish books appearing in Spanish problems, or specific Indian music genres appearing in Tamil problems. Connecting students' everyday lives to CS education through culturally relevant pedagogy (CRP) has been shown to improve their performance and learning experience [33, 35], so we felt this criterion could provide new insight for NNES education.

Following Sarsa et al., when the responses for a language disagreed, two researchers analyzed the responses in tandem to form a consensus on a Yes/No answer. The researchers used the evaluators' notes and their knowledge of the languages to make an informed decision. When unsure, they finalized on the side of there being errors (e.g., if a Yes and No for Solution Accuracy, finalized on No).

Our study was approved by our institution's IRB and the protocol number is 201128.

4 RESULTS

4.1 RQ1: Automated Evaluation

In this section we refer to failures and errors. Failures were mistakes in the sample solutions/tests that caused the tests to fail, but they still ran. Errors caused the tests to not be able to be run.

Of the 400 total exercises, Spanish problems performed best with 87% passing test cases, followed by English (84%), Vietnamese (72%), and Tamil (48%). Tamil had the most errors (32%). Most were because the response was incomplete and missing code (21%). Vietnamese had the most failures (28%). Overall statement coverage was 93.86% and all languages had individual coverage above 89%. The context of music performed best (81.3% pass rate), followed by books (72.5%), movies (72.5%), health (70.0%), and relationships (67.5%). See Table 2 for automated evaluation data. Percentages of Passed, Errors, Failed, and Complete are out of 100 (100 problems in each language), and percentages in the Total row are out of 400 total problems.

Table 2: Summary of automated analysis of problems

Lang	Passed	Errors	Failed	Coverage	Complete
Eng	84%	2%	14%	96.10%	100%
Tam	48%	32%	20%	92.15%	79%
Viet	72%	0%	28%	89.82%	100%
Span	87%	1%	12%	96.67%	99%
Total	291(72.8%)	35(9.0%)	74(19.0%)	93.86%	378(94.5%)

Common failures and errors across languages are seen in Table 3. The "Total" column percentages are out of the total failures (74) and errors (35). Of 74 total failures, the most common failures were logic failures with 36 (48.6%). This was when either the sample solution logic was incorrect causing the tests to fail, or the test logic was incorrect so it expected the wrong output. Many logic failures were due to ambiguity in the problem statement about bounds where the problem statement asked to print a message if a parameter was "between" certain bounds, but didn't state whether it included those bounds. This caused the sample solutions and tests to use conflicting inclusive/exclusive bounds. The second most common failure was a print/return failure with 25 (33.8%), the most common error found by Sarsa et al.. In these cases, the tests expected the function to return a message, but the sample solution printed it. The next common failure was punctuation failures with 6 (8.1%), when the test used incorrect punctuation for the expected output (e.g., actual = "good." expected = "good!"). Next was data type failures with 4 (5.4%), when the solution used an incorrect data type for a command (e.g., subtracting strings). Next was name failures with

Table 3: Summary of failures and errors in automated tests

Failures	Eng	Tam	Viet	Span	Total
Logic	9	13	8	6	36 (48.6%)
Print/Return	2	1	20	2	25 (33.8%)
Punctuation	2	1	0	3	6 (8.1%)
Name	0	3	0	1	4 (5.4%)
Data Type	1	1	0	0	2 (2.7%)
Index	0	1	0	0	1 (1.4%)
Errors	**Eng**	**Tam**	**Viet**	**Span**	**Total**
Incomplete	0	21	0	1	22 (62.9%)
Syntax	2	11	0	0	13 (37.1%)

Table 4: Summary of participant responses from manual evaluation

	Eng	Tam	Viet	Span	Total
Sensibleness	15	4	15	12	46 (76.7%)
Readability	15	0	15	14	44 (73.3%)
Translation	15	0	7	10	32 (53.3%)
Solution Accuracy	6	2	6	1	15 (25%)
Programming Concepts	15	4	15	15	49 (81.7%)
Context	15	7	14	14	50 (83.3%)
Cultural Relevance	1	4	13	1	19 (31.7%)

2 (2.7%), when the solution used an undefined variable name. The index failure was when a solution called an out-of-bounds index.

Of the 35 total errors, the most common was incomplete errors at 22 (62.9%), when the problem was missing the sample solution or tests. The other 13 were syntax errors (37.1%). Most of these solutions combined pseudocode and actual code (especially in Tamil).

4.2 RQ2: Manual Evaluation

Of the 60 problems manually evaluated (15 per language), we found 46 (76.7%) were sensible, 44 (73.3%) were readable, 32 (53.3%) had natural translations, 15 (25%) had accurate sample solutions, 49 (81.7%) had the proper programming concepts, 50 (83.3%) had the proper context, and 19 (31.7%) were culturally relevant (see Table 4). English, Spanish, and Vietnamese were mostly sensible, readable, and contained the proper concepts. Spanish and Vietnamese scored lower on Translation (10 and 7 respectively), meaning many problems sounded unnatural to the native speakers. All languages scored low for Sample Solution Accuracy. Most inaccuracies were due to print/return failures. Some of these cases were related to the language, such as in Spanish. In 6/15 Spanish problems, the problem statement used the term "devuelve" for outputting, which translates to "give back." This made it unclear to the evaluators whether to print or return. Using "imprima" (print) or "retorne" (return) would be better. Vietnamese scored highest (13) for cultural relevance.

Tamil was rated worse in most aspects. Most problem statements were described by native speakers as completely non-sensible. No problems were easily readable nor sounded like they were written by a native speaker. Only 4 used the proper programming concepts. However, it should be noted our participants explained that the words for programming concepts in many responses didn't make sense to them because they learned programming concepts in English, so they may or may not have been correct.

5 DISCUSSION

In this study, we found that problems generated in English, Spanish, and Vietnamese were largely sensible, accurate, and understandable to native speakers. Our findings confirm those of Sarsa et al. that problems generated in English could not be given verbatim to students, but would require only minor changes. We further this claim for Spanish and Vietnamese. However, Tamil problems had a much lower passing test rate and were mostly non-sensible. Thus, we think the abilities of LLMs for programming exercise generation are NOT generalizable across different natural languages.

Our English problems performed better in most aspects than those of Sarsa et al.. This included higher passing test rates (our 98% vs. their 30.9%), similar statement coverage (our 96.10% vs. their 98.0%), and higher manual evaluation scores in all aspects except Sample Solution Accuracy. We believe these differences are due to our use of GPT-3.5, a newer model than Codex. An alternative explanation is our use of a different sample size (our 15 per each language vs. their 120) but we think the model change had more effect since we saw improvements in most categories.

We were surprised Spanish problems performed better than English in our automated evaluation, since most training data used by OpenAI is in English (though specifics on the data for GPT-3.5 are hard to find) [6, 17]. A possible explanation is there was less Spanish training data, but the English data had more errors.

Tamil performed worst in both the automated and manual evaluations. We think this is due to features of the Tamil script. It is a non-Latin alphabet and is known to be flawed in writing colloquial/informal Tamil [38]. This may lead to less accurate Tamil training data for the model. An alternate explanation is there is less training data because Tamil is not as widely used as other languages, but we do not think this is the case since Tamil has a similar number of speakers (66 million [7]) to Vietnamese (70 million [8]), which performed much better.

It should be noted that not all problems that passed the automated evaluation may be fully accurate. As seen in the manual evaluation, most sample solutions didn't do exactly as the problem statement asked (25% sample solution accuracy), such as returning a message when the problem statement asked to print it. See [9] for an example. In some cases the solution and tests may both have returned, so the test passed, but the problem statement asked to print. We think the prevalence of print/return and logical failures is due to errors in the model's training data. Confusion about printing and returning is common amongst novice programmers [22, 28], so much code uploaded online and used to train the model may have such errors in it. Our ambiguous inclusive/exclusive bounds (the cause of most logic failures) are likely due to our priming problem not explicitly stating inclusive or exclusive. We feel these errors indicate these problems could not be given verbatim to students as they could reinforce incorrect understanding, especially when the LLM confidently presents a solution as correct.

However, most failures and errors we found could be fixed with minor changes. Print/return failures could be fixed by requesting a specific output or by re-generating the sample solution/tests. We briefly explored prompting ChatGPT to fix a print/return failure in its response, which was successful. Many logic failures could be fixed by specifying inclusive/exclusive bounds in the problem statement. Translation and readability errors could be fixed by slight wording changes (in English, Spanish, and Vietnamese). We were strict in our final decisions when manual responses disagreed, and many problems deemed not easily readable/well-translated needed only small phrasing changes. This does not apply to Tamil, where most problem statements were entirely non-sensible.

We echo Sarsa et al. that although most generated problems could not be given to students verbatim, they still have great value. Giving students a flawed problem in the classroom allows for learning both why it is flawed and how to fix it. This skill will become more important as LLMs develop and more code is written by code generation tools [19]. Our Tamil performance change from our initial to final prompt (17% to 100% generating in the correct language) confirms past research that prompt engineering is critical to accurate generation [19, 39], and is vital for students to learn. These problems will also help educators create assignments quickly with minimal input, including personalized ones. Though only 19/60 (31.7%) of our manually evaluated problems were rated culturally relevant, the ability to generate them at all is significant as culturally relevant pedagogy helps students feel more connected to CS (which may be even more important for non-native English speakers who face barriers in the field). Explicitly prompting for cultural relevance or using a newer model like GPT-4 may increase their frequency of generation. This could help when creating non-English textbooks, where it would usually require much more effort to create culturally relevant problems. See [10] for some of our culturally relevant examples. However, it should be noted it would be challenging for teachers to review (and fix errors in) personalized problems in a student's native language they may not understand.

5.1 Threats to Validity

One threat to the validity of our data was in the cultural relevance aspect of our manual evaluation. In many cases, Vietnamese speakers cited cultural relevance but did not specify the relevant topic and it could not be identified by the researchers. The Tamil speakers also often disagreed on this aspect, with one evaluator interpreting cultural relevance broadly and citing it often while the other rarely cited it. In these cases we were strict in our consensus, mostly finalizing on a "No" answer. Though we elaborated on cultural relevance in our instructions to participants, the question in the evaluation table was vague. This may have caused them to interpret cultural relevance as less or more specific than we originally intended (the usage of unique examples that are relevant to a specific culture). However, we believe it is impactful if participants felt a problem was culturally relevant to them regardless of whether the researchers agreed, as this means they feel connected to the material, promoting a positive learning environment.

We also didn't consult evaluators when making a final consensus on a Yes/No answer. We used their notes to understand their reasoning, but we could not know their entire thoughts behind their answers. If the two native speakers had discussed their responses with each other, their conclusion might be different from ours. In future research, we hope to have in-depth discussions with native speakers to get better insight into their responses.

A threat to the generalizability of our results is our use of only one priming problem. Many failures/errors we found were related to the structure of the speeding problem. These errors would likely not occur for other problems, but new errors/failures may occur depending on the priming problem. The context used in relation to the priming problem may also impact performance. The speeding problem uses conditionals to specify ranges, so easily-quantified contexts performed better, seen in relationships (a topic difficult to quantify) having the worst performance of the 5 contexts. More research is needed to make definitive conclusions about performance using a wider variety of contexts and priming problems.

5.2 Limitations and Future Work

We manually evaluated only 15% of our total problems, thus these statistics may not apply to all problems generated in that language. The evaluators also had various levels of computing experience. Future work should evaluate a larger number of problems to find out how often different evaluation aspects can be expected to generate.

We also didn't fine-tune any problems. We wanted to create a template for users to generate their own programming exercises with no context, so we prompted once with no follow-up prompts. Many errors we found may have been fixed with further prompting. We anticipate users to fine-tune by prompting to fix specific errors, so more research is needed on how to effectively do this. Since GPT models perform better on simpler sub-tasks [2], it is also possible prompting for problem elements separately could improve accuracy.

We also generated problems in only 3 non-English languages, and two of them used the Latin alphabet. These 3 languages are not fully reflective of the many non-English languages, but it does open the floor for further research to be done with a larger set of languages. Future work should expand to additional languages to discover how LLMs can create exercises for NNES of all backgrounds.

6 CONCLUSION

In this paper, we explored the efficacy of OpenAI's GPT-3.5 for generating introductory programming exercises in 4 natural languages. Although not perfect, problems generated in English, Spanish, and Vietnamese are largely accurate and understandable to native speakers and could be given to students with minor modifications. Tamil problems were mostly non-sensical and had far fewer accurate solutions, showing that the abilities of GPT-3.5 for problem generation are not the same across languages. More research is needed to gauge performance in a wider variety of languages. Despite errors, these problems can provide easily-accessible and culturally relevant practice for students and are a starting point for professors when creating programming assignments. The success of non-English exercises shows LLM problem generation is one viable strategy to address the lack of resources for NNES, bringing us one step closer to the goal of CS for all.

REFERENCES

[1] [n. d.]. https://openai.com/chatgpt Accessed on 08/09/23.
[2] [n. d.]. https://platform.openai.com/docs/guides/gpt-best-practices Accessed on 7/27/2023.
[3] [n. d.]. https://platform.openai.com/docs/guides/gpt/chat-completions-api Accessed on 7/26/2023.
[4] [n. d.]. https://platform.openai.com/docs/models/gpt-3-5 Accessed on 8/7/2023.
[5] [n. d.]. https://bit.ly/ManualEvaluationSpanish
[6] [n. d.]. https://commoncrawl.github.io/cc-crawl-statistics/plots/languages Accessed on 8/16/2023.
[7] [n. d.]. https://www.britannica.com/topic/Tamil-language Accessed on 7/28/23.
[8] [n. d.]. https://www.britannica.com/topic/Vietnamese-language Accessed on 7/28/23.
[9] [n. d.]. https://bit.ly/PrintReturnError
[10] [n. d.]. https://bit.ly/CulturallyRelevantExamples
[11] 2015. https://docs.pytest.org/en/7.4.x/ Accessed on 7/27/2023.
[12] Vardhan Agarwal, Yada Chuengsatiansup, Elise Kim, Yuzi LYu, and Adalbert Gerald Soosai Raj. 2022. An Analysis of Stress and Sense of Belonging Among Native and Non-native English Speakers Learning Computer Science. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*. 376–382.
[13] Suad Alaofi and Seán Russell. 2022. The Influence of Foreign Language Classroom Anxiety on Academic Performance in English-based CS1 Courses. In *Proceedings of the 2022 Conference on United Kingdom & Ireland Computing Education Research*. 1–7.
[14] Rishabh Balse, Bharath Valaboju, Shreya Singhal, Jayakrishnan Madathil Warriem, and Prajish Prasad. 2023. Investigating the Potential of GPT-3 in Providing Feedback for Programming Assessments. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 292–298.
[15] Brett A Becker. 2019. Parlez-vous Java? Bonjour La Monde!= Hello World: Barriers to Programming Language Acquisition for Non-Native English Speakers.. In *PPIG*.
[16] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.
[17] 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 (2020), 1877–1901.
[18] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023), 111734.
[19] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.
[20] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing Education in the Era of Generative AI. *arXiv preprint arXiv:2306.02608* (2023).
[21] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources–Leveraging Large Language Models for Learnersourcing. *arXiv preprint arXiv:2211.04715* (2022).
[22] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*. 83–89.
[23] Pedro Guillermo Feijóo-García, Keith McNamara, and Jacob Stuart. 2020. The Effects of Native Language on Block-Based Programming Introduction: A Work in Progress with Hispanic Population. In *2020 Research on Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT)*, Vol. 1. IEEE, 1–2.
[24] 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 *Proceedings of the 24th Australasian Computing Education Conference*. 10–19.
[25] Philip J Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–14.
[26] Carmen Nayeli Guzman, Anne Xu, and Adalbert Gerald Soosai Raj. 2021. Experiences of Non-Native English Speakers Learning Computer Science in a US University. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 633–639.
[27] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. *arXiv preprint arXiv:2306.05715* (2023).
[28] Cruz Izu and Peter Dinh. 2018. Can Novice Programmers Write C Functions?. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, 965–970.
[29] 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*. 1–23.
[30] Sam Lau and Philip J Guo. 2023. From" Ban It Till We Understand It" to" Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. (2023).
[31] Yinchen Lei and Meghan Allen. 2022. English Language Learners in Computer Science Education: A Scoping Review. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*. 57–63.
[32] 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. *arXiv preprint arXiv:2304.03938* (2023).
[33] Tia C Madkins, Alexis Martin, Jean Ryoo, Kimberly A Scott, Joanna Goode, Allison Scott, and Frieda McAlear. 2019. Culturally relevant computer science pedagogy: From theory to practice. In *2019 research on equity and sustained participation in engineering, computing, and technology (RESPECT)*. IEEE, 1–4.
[34] Kamil Malinka, Martin Peresíni, Anton Firc, Ondrej Hujnák, and Filip Janus. 2023. On the educational impact of ChatGPT: Is Artificial Intelligence ready to obtain a university degree?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 47–53.
[35] Marlon Mejias, Ketly Jean-Pierre, Legand Burge, and Gloria Washington. 2018. Culturally relevant cs pedagogy-theory & practice. In *2018 Research on Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT)*. IEEE, 1–5.
[36] Eng Lieh Ouh, Benjamin Kok Siew Gan, Kyong Jin Shim, and Swavek Wlodkowski. 2023. ChatGPT, Can You Generate Solutions for my Coding Exercises? An Evaluation on its Effectiveness in an undergraduate Java Programming Course. *arXiv preprint arXiv:2305.13680* (2023).
[37] 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. *arXiv preprint arXiv:2304.02491* (2023).
[38] Sankaran Radhakrishnan. [n. d.]. Tamil Diglossia. https://sites.la.utexas.edu/tamilscript/category/people-and-culture/tamil-diglossia Accessed on 8/7/2023.
[39] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 299–305.
[40] 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*. 27–43.
[41] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. *arXiv preprint arXiv:2306.10073* (2023).
[42] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses? *arXiv preprint arXiv:2303.09325* (2023).
[43] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).
[44] Tongshuang Wu, Kenneth Koedinger, et al. 2023. Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming. *arXiv preprint arXiv:2306.05153* (2023), 1–12.