Using GitHub Copilot to Solve Simple Programming Problems

Michel Wermelinger
School of Computing and Communications
The Open University
Milton Keynes, United Kingdom
michel.wermelinger@open.ac.uk

ABSTRACT

The teaching and assessment of introductory programming involves writing code that solves a problem described by text. Previous research found that OpenAI's Codex, a natural language machine learning model trained on billions of lines of code, performs well on many programming problems, often generating correct and readable Python code. GitHub's version of Codex, Copilot, is freely available to students. This raises pedagogic and academic integrity concerns. Educators need to know what Copilot is capable of, in order to adapt their teaching to AI-powered programming assistants. Previous research evaluated the most performant Codex model quantitatively, e.g. how many problems have at least one correct suggestion that passes all tests. Here I evaluate Copilot instead, to see if and how it differs from Codex, and look qualitatively at the generated suggestions, to understand the limitations of Copilot. I also report on the experience of using Copilot for other activities asked of students in programming courses: explaining code, generating tests and fixing bugs. The paper concludes with a discussion of the implications of the observed capabilities for the teaching of programming.

CCS CONCEPTS

• **Computing methodologies** → Natural language processing; • **Software and its engineering** → *Automatic programming*; • **Social and professional topics** → **CS1**.

KEYWORDS

code generation, test generation, code explanation, programming exercises, programming patterns, novice programming, introductory programming, academic integrity, OpenAI Codex

ACM Reference Format:
Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569830

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569830

1 INTRODUCTION

Program synthesis is an active research field with a long history. Recent developments include OpenAI's Codex[1] [1], DeepMind's AlphaCode[2] [8], Amazon's CodeWhisperer[3] and Tabnine[4], four systems that can translate a problem description to code. Currently (August 2022), AlphaCode and CodeWhisperer aren't publicly available and Tabnine's free version only does code suggestions. This paper considers GitHub's Copilot[5], a version of Codex accessible through a plugin for popular IDEs.

In June 2022, GitHub released Copilot to individual customers and included it in the free Student Pack. Codex, publicly available since November 2021, is a paid service, accessed by writing a program that calls OpenAI's API. These are barriers to adoption by students and Copilot removed them. Whether we like it or not, many students will use the free IDE plugin for exercises and assignments, without having to learn an API or disrupt their workflow.

With a suitable configuration, the most performant Codex model (Davinci) can solve typical CS1 problems [4]. (I use 'solve' in the sense of 'generate an answer that passes the tests'.) However, Davinci is also the slowest and most expensive of the models and we know that "a distinct production version of Codex powers GitHub Copilot" [1]. Given the recent release of Copilot and its expected widespread use by students, it is timely to check if it performs as well as Davinci, and if not, what limitations it has.

OpenAI's Codex examples[6] include fixing a bug and generating documentation strings, step-by-step explanations and code summaries. Sarsa et al. showed that Davinci can create exercises (as variations of a given problem), sample solutions, explanations and tests [12]. However, Copilot's FAQ states that it "is not intended for non-coding tasks like data generation and natural language generation, like question and answering". We can thus expect Copilot to perform less well than Davinci in generating tests and code explanations, two tasks often asked of students.

The demo videos on the Codex site illustrate the incremental creation of programs: the user's initial request generates a minimal program or function that is modified by each subsequent request. Codex and the user engage in a 'dialogue' in which the user's English sentences elicit a reply in a programming language. Copilot makes this interaction stronger and more seamless: it makes suggestions as we type in the IDE, we can ask for alternative suggestions with one keystroke and we can edit Copilot's suggestions. It's no surprise that GitHub dubs Copilot as "your AI pair programmer",

[1] https://openai.com/blog/openai-codex
[2] https://www.deepmind.com/blog/competitive-programming-with-alphacode
[3] https://aws.amazon.com/codewhisperer
[4] https://www.tabnine.com
[5] https://copilot.github.com
[6] https://beta.openai.com/examples?category=code

even though the interaction is far more limited than with a human; notably, Copilot does not provide a rationale for its suggestions.

In summary, Copilot is currently the most likely AI assistant to be adopted by students, but only Codex has been evaluated, in a quantitative way, mostly with automated tests to obtain the percentage of correct code suggestions for a given problem statement. While Copilot could be evaluated in the same way, a qualitative look at Copilot's answers may be more insightful. This paper is a report of my experience in using Copilot as if I were a student tasked with writing code (possibly with an explanation) and tests for a given CS1 problem. The guiding questions for this exploration are:

- How does Copilot perform, compared with Davinci, in terms of the correctness and variety of the generated code, tests and explanations?
- If a suggestion is incorrect, can Copilot be interactively led to a correct one?

2 RELATED WORK

The OpenAI team wrote 164 problems[7] with tests, to evaluate if early models of Codex generate correct Python code from English [1]. Almost 29% of the problems were solved by Codex's first suggestion. A fine-tuned model, trained only on correct Python code, solved 38% of the problems on its first attempt. If that model is allowed 100 attempts per problem, then at least one of them was correct for 78% of the problems. Judging from the randomly selected attempts in the paper's appendix, most are incorrect.

Finnie-Ansley et al. [4] investigated the performance of the Davinci model on two sets of CS1 problems in Python, setting the temperature parameter to 90% to elicit more creative answers.

The first set had 23 problems from past tests of their CS1 course. Codex had ten attempts at each problem. It solved 10 problems on the first attempt, it needed manual output correction for some problems, and it couldn't solve 4 problems that required ASCII formatting or restricted what students could use, e.g. 'you must use a while-loop but not the `split()` method'.

The second set consisted of 7 variants of the rainfall problem [3–7, 13, 14]. Codex was asked for 50 solutions to each problem. For all but one variant, at least one attempt passed all tests. The 50 attempts passed on average 19% of the tests for the Ebrahimi et al. variant [3], up to 63% of the tests for Soloway's [14].

Sarsa et al. [12] also evaluated Davinci, but from the perspective of a teacher who wants varied but related exercises in different domains to better engage their students. The researchers submitted to Codex a Python docstring containing: keywords describing the domain (e.g. 'football') and the programming context (e.g. 'list'); a base problem in English; a sample solution; some tests. This was followed by different domain and programming keywords and a prompt to generate another exercise (statement, solution and tests). The authors read half of the 240 exercises generated and 75% of those were reasonable, i.e. they made sense, incorporated the prompted domain or programming concept, and included appropriate code. However, 30% of the exercises had no solution or no tests. When both existed, only 31% of the solutions passed the tests.

[7] https://github.com/openai/human-eval

```
"""
Write a program that will read the amount of rainfall for each day.
A negative value of rainfall should be rejected, since this is invalid and inadmissible.
The program should print out the number of valid recorded days,
the number of rainy days, the rainfall over the period, and
the maximum amount of rain that fell on any one day.
Use a sentinel value of 9999 to terminate the program.
"""

def main():
    """
    Main function
    """
    rainy_days = 0
    total_rainfall = 0
    max_rainfall = 0
    valid_days = 0
    rainfall = int(input("Enter the rainfall for a day: "))
    while rainfall != 9999:
        if rainfall >= 0:
            valid_days += 1
            total_rainfall += rainfall
            if rainfall > max_rainfall:
                max_rainfall = rainfall
            if rainfall > 0:
                rainy_days += 1
        rainfall = int(input("Enter the rainfall for a day: "))
    print("The number of valid days is:", valid_days)
    print("The number of rainy days is:", rainy_days)
    print("The total rainfall is:", total_rainfall)
    print("The maximum rainfall is:", max_rainfall)
```

Figure 1: An inline suggestion for Ebrahimi's rainfall problem variant.

The authors also asked Codex to generate line-by-line explanations for three functions and a class: only two thirds of the generated lines were correct. Codex often got the operator wrong, e.g. `speed > 100` was explained as 'speed is less than 100'.

In summary, previous research shows that while Codex solves a good percentage of problems on the first attempt, it requires many attempts for most problems, and it never solves quite a number of problems. None of the related work captures how most students will use Codex: interactively in an IDE.

3 USING COPILOT

Copilot is accessible via a plugin for several editors. I used Visual Studio Code within the 60-day free Copilot trial period GitHub users have. It's unlikely results would be different with other IDEs as they are only interfaces to Copilot.

In VS Code, Copilot attempts to provide an inline suggestion when the user pauses typing or presses Alt-\ or Enter. The suggestion, in italic grey font at the cursor position, may complete the current line or may be several lines long (Figure 1). The user can cycle through alternative suggestions with Alt-[or Alt-]. Pressing Tab accepts the current suggestion. Pressing Ctrl-Enter requests Copilot to generate up to 10 suggestions and display the unique ones in a separate panel. Each suggestion has an acceptance button, which copies it to the editor.

The following sections detail the prompts used, i.e. the code and comments in the editor at the time a suggestion is asked for. All code is in Python, to allow comparing with previous work and because it is the introductory programming language at our institution.

Copilot can be configured in the Settings page of the user's GitHub account. Users can choose whether to see suggestions that match public code and whether to allow GitHub and OpenAI to use the submitted prompts for training. I declined both. The plugin doesn't indicate which suggestions match public code, so students

are likely to hide them to avoid unintended plagiarism. Although unlikely, allowing to use the prompts for training might have changed the results over the course of experiencing Copilot.

4 GENERATING CODE

I used Copilot with two CS1 problem sets. This section presents the qualitative results, illustrated with selected examples. I highlight Copilot's mistakes with `# Wrong` and, due to limited space, I replace repetitive generated code with an ellipsis.

4.1 Programming patterns

The use of programming and design patterns is grounded in cognitive theories of how people construct and organise knowledge and become problem-solving experts. Muller et al. [10] introduced 30 programming patterns, showing that they improve the students' ability to solve problems. Patterns are step-by-step algorithmic templates and thus I wanted to check if they are suited to incremental program construction as demonstrated by OpenAI.

For our introductory programming module, I created a small set of patterns [11] and 8 exercises delivered via the CodeRunner plugin [9] for Moodle. Exercises include printing how many images fit in a disk (formula pattern), the strength of an earthquake given its Richter magnitude (case analysis), printing the list of values outside a range (list filtering), and printing the percentage of such values (pattern combination). For the last exercise, students are expected to combine the list filtering and formula patterns. For every other exercise, the CodeRunner editor shows the needed pattern as Python comments. Students write the corresponding code line(s) after each comment. CodeRunner runs some tests on their code and presents the results. Students have unlimited attempts: the exercises aren't graded. They just help students practice the instantiation of patterns and the basics of programming.

4.1.1 Method. I copied each exercise to VS Code, as in Figure 2: I put the problem statement as-is within a docstring, keeping the instructions to students and the references to our materials. On CodeRunner, each problem description includes a table of example inputs and corresponding outputs. I translated the HTML table to Markdown, within the docstring. The CodeRunner editor content (the comments with the pattern) was copied unchanged.

I pressed Enter at the end of each comment line to make Copilot suggest the code for that step of the pattern (see line 22 in Figure 2), until the whole program was generated. If it was incorrect or could be improved, I added a prompt (a comment with an English instruction) after the code, like in OpenAI's Codex examples.

4.1.2 Results. Copilot solved 4 of the 8 problems on the first attempt. It extracted the relevant information from the docstring, translated the general pattern into problem-specific code, found the right formula even if it was not given, and used appropriate variable names, like `kelvin_values` instead of `output_list` in Figure 2. When Copilot used the generic name from the pattern, prompt 'Rename ... to ...' led to the desired improvement.

Copilot generated the correct algorithm for one exercise but it failed the automated test because Copilot added one word to the output message. Copilot failed another exercise because it wrote code with an undefined variable:

```
1  """This question is about Block 2 Section 2.1.4 (Transform).
2
3  Consider again a list of hourly temperature readings from a fridge, in degrees Celsius,
4  as described in Question 9. Write a program that prints the list of temperatures,
5  but in degrees Kelvin. Find the appropriate formula online.
6
7  The answer box contains a simplification of Pattern 2.4 (List transformation) as comments,
8  to get you started. Write the corresponding code beneath each comment. Use variable names
9  appropriate to the problem at hand instead of the generic names used by the pattern.
10 The first step has been done for you.
11
12 For example:
13
14 Input | Result
15 -|-
16 [0, 5.85, -2.5] | [273.15, 279.0, 270.65]
17 """
18
19 # initialise the input_list with the given values
20 celsius_values = get_input()
21 # initialise the output_list to the empty list
22 kelvin_values = []
23 # for each input_value of the input_list:
24     # transform the input_value into an output_value
25     # append the output_value to the output_list
26 # print the output_list
```

Figure 2: The list transformation exercise, with an inline suggestion (line 22).

```
# if input values fall into the first case:
if magnitude < 4:
#   compute outputs according to the first case
  print("That is a minor earthquake.")
...
# print the outputs
print(output)   # Wrong
```

Replacing 'compute outputs' with 'compute the description' made Copilot assign each string to a `description` variable and print it at the end, as our students are intended to use the pattern. Impressively, after I changed the last comment to 'print the full message' and gave the prompt 'Now let the description be just the adjaective.' (note the typo), Copilot generated:

```
# if input values fall into the first case:
if magnitude < 4:
#   compute the description according to the first case
  description = "minor"
...
# print the full message
print("That is a {} earthquake.".format(description))
```

The hardest exercise asks students to generate a pseudo-random number sequence until the seed is repeated. The problem statement tells students they must change the given pattern. Copilot follows the pattern and produces a while-loop (`while value != seed`) that is never entered because the first value is the seed. After several attempts, prompt 'Modify the above program so that it computes and prints the second value before entering the loop' did generate the required additional code, but in the separate suggestions, which required editing to integrate them in the program.

For the list transformation exercise (Figure 2), Copilot called a function `transform` on each value. Prompted with `def trans` after the code, Copilot wrote a correct Celsius to Kelvin transformation function, with a good docstring.

4.1.3 Analysis. One the one hand, it is notable how Copilot sifts the chaff from the grain in the problem statement, including extracting the output messages from Markdown tables, and how it defines auxiliary functions as needed.

On the other hand, it fails to solve half of these simple problems. While Copilot can be instructed to incrementally fix or improve a program, the user must know exactly what they want and find the right words to communicate their intentions. It's easier and quicker to edit the code directly.

Copilot sometimes uses constructs novice programmers might not know about, like list comprehensions. How students will react to such suggestions remains to be seen.

4.2 The rainfall problem

4.2.1 Method. To directly compare Copilot and Davinci, I asked Copilot to solve the same rainfall problem variants as Finnie-Ansley et al. For each one, the prompt was a docstring with the problem description given in Table 2 of [4], followed by `def`, to make Copilot suggest the function header and the body. Since the problem variants are similar, I didn't accept Copilot's suggestions, to reduce any learning effect that might increase the correctness or reduce the variety of the suggestions for subsequent variants.

The rainfall problem asks for the mean of the valid numbers (what is valid depends on the variant) up to the first sentinel value. If there are no valid numbers, the mean is undefined. Most variants don't state what to return in that case or if we can assume the input to have a valid number. I took the latter stance. Finnie-Ansley et al. tested Davinci's suggestions with the empty input and with only invalid numbers, but it's unclear what outcome their tests expect.

4.2.2 Results. Copilot made for every variant 1–3 inline suggestions and less than 10 unique separate suggestions, but often they were essentially the same, differing in the names of variables, the docstring (or its absence), the messages in `input()` and `print()`, and minor coding variations, e.g. iterating with a repeat-until loop or an infinite while-loop with a `break`.

Soloway's original problem [14] is the simplest: read numbers from the input until the sentinel occurs and compute their mean. All of Copilot's suggestions are correct.

Simon [13] asks to treat negative values as if they were zeroes. Copilot does not do that: it adds all values until the sentinel.

Guzdial et al. [6] ask for the mean of the positive values; there is no sentinel. All Copilot suggestions are correct, except two. One does `if value < 0: continue` instead of `<= 0` to skip non-positive values. The other uses `len` to divide the sum by the length of the list, even though this is the only variant stating that one must divide by the number of valid values.

Lakanen et al. [7] ask for the mean of the positive numbers up to the first value exceeding 998. The description uses the word 'sentinel', but it is not a fixed value like in the other variants. This was the only variant Davinci couldn't solve, and neither can Copilot: one suggestion breaks the loop on 999 only; the others don't check for a sentinel. All suggestions (except the one with the `break` statement) have an unnecessary `else: continue` branch, a striking example of the lack of variety.

Fisler [5] asks for the mean of the non-negative values up to the sentinel, which is optional. The statement is 'Design a program [...] that consumes a list of numbers [...] entered by a user. The list may contain [the sentinel]'. Contrary to all other variants, this leads Copilot to generate two kinds of algorithms: while-loops that use `input()` and break upon reading the sentinel, and for-loops that iterate over a list parameter. Only one for-loop suggestion and two while-loop suggestions are correct: the others use `len()` or don't check whether a value is negative. This variant elicited the most varied suggestions: some use a while-loop to read the values up to the sentinel into a list, followed by a for-loop to add and count the non-negative values; others use `sum() / len()` and one uses slicing.

Ebrahimi [3] asks for a program that reads rainfall amounts (negative values are invalid) until the sentinel occurs and then outputs the number, total and maximum of the valid values and 'the number of rainy days', which isn't further explained. This was the variant on which Davinci performed worst, with suggestions passing on average 2 of the 10 tests. Surprisingly, Copilot's first inline suggestion (Figure 1) and most other ones are correct. The incorrect suggestions compute the mean or use the wrong variable when printing. Some suggestions unnecessarily store the values read in a list that isn't further used. The prompt 'The next code doesn't create a rainfall list' removed it.

Finnie-Ansley et al. [4] ask for the mean, rounded to one decimal place, of the non-negative values up to the sentinel or the end of the list, whichever occurs first. If the input is `None` (instead of a list of numbers) or the mean is undefined, then the output should be -1.0. Copilot's suggestions are all incorrect. They use `len()`, don't round the result, don't stop at the sentinel, or they check if the sum (rather than the count) of valid values is zero to return -1. The first inline suggestion is almost correct: it doesn't round.

4.2.3 Analysis. For every variant, Finnie-Ansley et al. found that Davinci generated a variety of attempts, including doing one or two passes over the list, with a for-loop or a while-loop. Copilot generates one approach, possibly combined with `len()`, per variant: a single-pass for-loop if the input is a list and a single-pass while-loop if the standard input is read. It generates both approaches only for Fisler's variant, likely due to both 'entered by a user' and 'list' occurring in the problem description.

Looking more closely at the suggestions for each variant, they seem to be obtained largely by combining, say, two different names for one variable, two docstrings, two ways of using the while-loop, etc. Even though the code of each suggestion is unique, algorithmically they are essentially the same.

There are at least two reasons why Copilot's suggestions are much less varied than Davinci's. First, Copilot generates about 10–12 suggestions per variant, whereas Davinci was asked for 50. Second, Copilot's temperature is probably low, because lower temperatures tend to give more precise results while higher temperatures, as used with Davinci, tend to produce more random results.

The lack of variety led to more polarised correctness results than those reported by Finnie-Ansley et al. The suggestions tend to be mostly (or exclusively) correct or incorrect. Davinci found a solution for 6 of the 7 variants; Copilot solved 4 variants.

Also to note, the separate suggestions, which take a few seconds to appear, frequently do not improve on the inline suggestions, which are almost immediate. This may be on purpose, to provide a good user experience and avoid further requests to the servers.

5 GENERATE TESTS

Students are sometimes asked to write tests in order to think about edge cases. Can Copilot generate comprehensive tests?

The programming pattern exercises didn't ask students for tests, so I requested tests for the rainfall problems, using the prompt `def test_`, sometimes after the code Copilot had generated, sometimes after the problem description, to mimic test-driven development, in which tests are written before the code.

Typical suggestions, both inline and separate, both after the problem and after the code, look like Listing 1. While some tests are correct, most are wrong and extend a data sequence in a repetitive way. Copilot stops midway through a list, which is a syntax error.

Using a different prompt and requesting the tests line by line (with Enter) starts off well...

```
# Write unit tests below this line
def test_harvest_none():
  assert harvest(None) == -1.0
def test_harvest_empty():
  assert harvest([]) == -1.0
def test_harvest_one():
  assert harvest([1]) == 1.0
def test_harvest_two():
  assert harvest([1, 2]) == 1.5
def test_harvest_three():
  assert harvest([1, 2, 3]) == 2.0
```

... but Copilot again settles into a repetitive testing pattern. Describing the required unit tests brings more variety:

```
# A unit test with positive and negative values
def test_harvest_positive_negative():
  assert harvest([1, 2, -3]) == 1.5
def test_harvest_negative_positive():
  assert harvest([-1, 2, 3]) == 2.0       # Wrong
def test_harvest_negative_negative():
  assert harvest([-1, -2, -3]) == -1.0
def test_harvest_positive_positive():
  assert harvest([1, 2, 3]) == 2.0
```

After leaving a blank line, continuing to press Enter and Tab generates this:

```
# A unit test with a sentinel
def test_harvest_sentinel():
  assert harvest([1, 2, 3, -999]) == 2.0
def test_harvest_sentinel_positive():
  assert harvest([1, 2, 3, -999, 4]) == 2.5   # Wrong
def test_harvest_sentinel_negative():
  assert harvest([1, 2, 3, -999, -4]) == 1.5  # Wrong
```

The generated comment suggests that Copilot 'knows' that it has yet to test for the sentinel value, but the wrong tests show that it has no understanding of the sentinel's effect.

6 GENERATE EXPLANATIONS

Explaining code allows students to 'think aloud' about their code and educators to check the students' understanding. Codex can translate code to natural language. How does Copilot fare?

I typed `#` at the end of lines of code to request a comment from Copilot. Figure 3 shows the comments generated for a wrong suggestion. (The file also contains the problem description, not shown in the figure.) Most comments are literal translations of the code, but some explain the role of a number or variable: -999 is the sentinel, the 1 in the last line is the number of decimal places and `count` is the number of non-negative values. Whether these comments are enough to make students realise that the code counts all numbers instead of the valid ones remains to be seen.

```
def harvest(list):
    if list is None:      # if the list is None, return -1.0
        return -1.0
    else:
        sum = 0           # initialize the sum to 0
        count = 0         # initialize the count to 0
        for i in list:    # for each value in the list
            if i == -999: # if the value is the sentinel, break
                break
            else:         # otherwise, add the value to the sum and increment the count
                sum += i
                count += 1
        if count == 0:    # if there are no non-negative values, return -1.0
            return -1.0
        else:             # otherwise, return the average of the non-negative values
            return round(sum/count, 1) # round the average to 1 decimal place
```

Figure 3: Comments generated for the Finnie-Ansley variant.

Generated comments may explain unknown constructs to students. Here are the comments generated for a suggestion for Soloway's variant. I moved some comments one line up to improve the layout.

```
total = 0
count = 0
while True:
    try: # try to read in a rainfall value
        number = int(input("Enter a number: "))
        # if the value is 99999, break out of the loop
        if number == 99999:
            break
        total += number
        count += 1
    # if the value is not an integer, print an error
        message
    except ValueError:
        print("Invalid input")
        # continue to the next iteration of the loop
        continue
# print the average of the rainfall values
print("The average is: {}".format(total / count))
```

Copilot explains the `try ... except`, `break` and `continue` constructs, but not the `format` syntax.

The prompt 'Explain the code above:' (the colon seems crucial) also generates text. If the file includes the problem statement, Copilot just repeats it as the explanation. Without the problem statement, Copilot attempts to summarise *what* the code does, without explaining *how* it works. For example, for a correct solution to Ebrahimi's variant, Copilot generated:

> The code above is a program that asks the user to enter rainfall for a day. The user will enter 9999 to indicate that they are finished entering rainfall. The program will then print the number of valid days, the number of rainy days, the total rainfall, and the maximum rainfall.

If we type '1.' after the prompt to explain the code, Copilot, like Codex [12], generates a list of statements. For the same 23-line program, the generated 14-step list formed this text:

> The code above is a program that asks the user to enter rainfall for a day. The rainfall is stored in a list. The rainfall is checked to see if it is valid. If the rainfall is valid, the rainfall is added to the list. The rainfall is checked to see if it is the maximum rainfall. If the rainfall is the maximum rainfall, the maximum rainfall is updated. The rainfall is checked to see if it

Listing 1: A test function for the rainfall variant of Finnie-Ansley et al.

```
def test_harvest():
  """
  Test the harvest function with various inputs.
  """
  assert harvest([]) == -1.0
  assert harvest([1, 2, 3, 4, 5]) == 3.0
  assert harvest([1, 2, 3, 4, 5, -999]) == 3.0
  assert harvest([1, 2, 3, 4, 5, -999, 6, 7, 8, 9, 10]) == 5.5       # Wrong
  assert harvest([1, 2, 3, 4, 5, -999, 6, 7, 8, 9, 10, -999]) == 5.5 # Wrong
  assert harvest([1, 2, 3, 4, 5, -999, 6, 7, 8, 9, 10, -999, 11, 12, 13, 14, 15]) == 7.5 # Wrong
  ...
  assert harvest([1, 2, 3, 4, 5, -999, 6, 7, 8, 9, 10, -999, 11, 12, 13, 14, 15, -999, 16, 17, 18, 19, 20, -999, 21, 22,
        23, 24, 25, -999]) == 13.5 # Wrong
  assert harvest([1,
```

> is greater than 0. If the rainfall is greater than 0, the rainy days is updated. The total rainfall is updated. The number of valid days is updated. The number of rainy days is updated. The total rainfall is updated. The maximum rainfall is updated. The code prints the number of valid days, the number of rainy days, the total rainfall, the maximum rainfall, and the average rainfall.

The step-by-step description is repetitive, sometimes unclear, misses the check for the sentinel in the code, puts some steps in the wrong order, and wrongly states that the code prints the average rainfall.

7 CONCLUDING REMARKS

The first guiding question was about Copilot's performance compared with Davinci's, in terms of correctness and variety of answers. Comparing my results to those cited about Davinci, it seems clear that Copilot fares less well on both accounts: as observed in the code, tests and explanations generated, Copilot's suggestions often are wrong, include unnecessary elements or are mainly 'variations on a theme', possibly due to a low default temperature.

Sometimes Copilot seems to have an uncanny understanding of the problem, able to extract the relevant details from text and tables of examples, while ignoring student instructions and references to materials. Other times, Copilot looks completely clueless, generating gibberish, like irrelevant lists of imports.

The second guiding question probed if Copilot can be interactively led to a correct answer. As in the first question, the examples provide a sobering reminder that, in spite of all the hype, using tools like Copilot can be a frustrating 'hit and miss' affair. Copilot most often does not understand our instructions to fix or improve the code it generated unless we formulate them in a very specific way. I felt like Gandalf trying to open the Doors of Durin.

Neither this nor previous work has studied how students will use Codex and Copilot in practice. Even without such a study, some educated guesses of what the future holds can be made.

Finn-Ansley et al. note that Codex poses challenges to academic integrity that can't be 'wished away': educators must adapt to the new reality. This is even more so with Copilot's free IDE plugin.

While less performant than Davinci, Copilot does generate code (and with some editing, tests and explanations) that could have been written by humans. Detecting and punishing the use of Copilot is impossible and futile. A more fruitful approach is to adopt these tools and educate students and colleagues about their advantages and limitations. Knowing what Copilot is good at and what it can't do helps students and educators understand what they need to learn, teach and assess in an age where up to 40% of code is written by Copilot, when it is used [2].

Copilot can provide a first helpful attempt at a problem, but students still need to know a language's syntax and semantics well, in order to spot and modify Copilot's often incorrect suggestions.

Copilot's 'explanations' may help students understand unfamiliar constructs and detect errors in the code, but they mostly describe *what* the code does at a low level of detail and sometimes omit important aspects. Students still need to learn how to write clear, high-level, synoptic documentation and they still need to figure out *why* some code doesn't work, in order to fix it.

Copilot can provide several suggestions, but to be productive, students must be able to quickly read and understand code without running it, in order to choose the correct suggestion or to combine the correct parts from different suggestions.

Copilot quickly completes comments, code lines and individual tests with hardly any syntax errors. Students will spend less time typing code and understanding compiler errors, and work through more exercises. Lecturers and TAs can focus more on documentation, testing, debugging and program comprehension.

Educators should stop 're-dressing' old toy problems that only have 1–2 sensible solutions, because Copilot's first suggestion will likely be correct, readable and the simplest one, thus restricting the learning opportunities for coding, debugging and algorithmic thinking, compared to problems with interesting 'wrinkles'.

Finnie-Ansley et al. note that students will likely get partial credit if they submit any of Davinci's suggestions, since almost each one passes some tests. This also happens with Copilot. Educators may need to increase the use of all-or-nothing grading, to make students analyse and combine partially correct solutions.

As this experience shows, Copilot is a useful springboard to productively solve CS1 problems, but the algorithmic thinking, program comprehension, debugging and communication skills are as needed as ever. As Jean-Baptiste Karr observed: the more things change, the more they stay the same.

REFERENCES

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[2] Thomas Dohmke. 2022. GitHub Copilot is generally available to all developers. GitHub blog. https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers

[3] Alireza Ebrahimi. 1994. Novice Programmer Errors: Language Constructs and Plan Composition. *Int. J. Hum.-Comput. Stud.* 41, 4 (Oct. 1994), 457–480. https://doi.org/10.1006/ijhc.1994.1069

[4] 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* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[5] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/2632320.2632346

[6] Mark Guzdial, Rachel Fithian, Andrea Forte, and Lauren Rich. 2003. *Report on Pilot Offering of CS1315 Introduction to Media Computation With Comparison to CS1321 and COE1361.* Technical Report. Georgia Tech.

[7] Antti-Jussi Lakanen, Vesa Lappalainen, and Ville Isomöttönen. 2015. Revisiting Rainfall to Explore Exam Questions and Performance on CS1. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 40–49. https://doi.org/10.1145/2828959.2828970

[8] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

[9] Richard Lobb and Jenny Harlow. 2016. Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (Feb. 2016), 47–51. https://doi.org/10.1145/2810041

[10] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-Oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland) *(ITiCSE '07)*. Association for Computing Machinery, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[11] Paul Piwek, Michel Wermelinger, Robin Laney, and Richard Walker. 2019. Learning to Program: From Problems to Code. In *Proceedings of the 3rd Conference on Computing Education Practice* (Durham, United Kingdom) *(CEP '19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 4 pages. https://doi.org/10.1145/3294016.3294024

[12] 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* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. Association for Computing Machinery, New York, NY, USA, 27–43. https://doi.org/10.1145/3501385.3543957

[13] Simon. 2013. Soloway's Rainfall Problem Has Become Harder. In *Learning and Teaching in Computing and Engineering*. IEEE, 130–135. https://doi.org/10.1109/LaTiCE.2013.44

[14] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594