Assessing CodePilot's Breadth

Justin Zhang*
JustinZhang@cmail.carleton.ca
Carleton University
Ottawa, Ontario, Canada

Kaya Gouin*
KayaGouin@cmail.carleton.ca
Carleton University
Ottawa, Ontario, Canada

Abstract

...

Keywords

Software engineering, prompt engineering, artificial intelligence, few-shot

ACM Reference Format:
Justin Zhang and Kaya Gouin. 2024. Assessing CodePilot's Breadth. In *Mining Software Repositories, December 2024, Ottawa, ON*. ACM, Ottawa, ON, Canada, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

LLMs, code coverage, CodePilot. Why we care.

2 Assessing CodePilot's Breadth

Introduce our own work, and the motivation behind it, by talking about the fact that CodePilot is very new and has only been tested with Python, so very limited. Here we want to explore the breadth of CodePilot by testing it on different types of programming languages. We are interested in assessing both the quality of the plan that the model outputs, the quality of the code coverage prediction that the model outputs, and the link between the two (i.e. does it look like the model uses its own plan to predict code coverage). These are our research questions.

3 Methodology

Five different programming languages, each belonging to a different type of language class. Onephase. Two-shot, so we give two exemplars to the model. Two tests for each language. We match the exemplar and test language. We wanted small exemplars with some level of complexity. The two exemplars for all languages are implementations of the same algorithm, one with a certain level of complexity. The two tests for all languages are implementations of the same algorithm, one with a certain level of complexity. Talk about the exact algorithms. We run each experiment once. Temperature of 0.6. GPT instruct 3.5. The template that we use (the same as in CodePilot's original paper).

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit

```
Input:

For the given code snippet, predict the code coverage.
The code coverage indicates whether a statement has been
executed or not.

> if the line is executed
! if the line is not executed

Example output:
> line1
! line2
> line3
...
> linen

You need to develop a plan for step by step execution of
the code snippet.

Do not answer unless instructed to do so

DISCLAIMER: Lines that are not executed are to be denoted
with a SINGLE '!' whereas lines that are executed are to be
denoted with a single '>'

Below are a couple examples of the process you need to follow
to predict the code coverage of a given code snippet and its
plan.

[example 1]

[example 2]

In a similar fashion, develop a PLAN of step by step execution
of the below code snippet and predict the CODE COVERAGE.

[code snippet]

Output:

[output plan]
[output code coverage]
```

Figure 1: Template

*Both authors contributed equally to this research.

COMP 5117, December 2024, Ottawa, ON
2024. ACM ISBN XXX-X-XXXX-XXXX-X/24/12
https://doi.org/XXXXXXX.XXXXXXX

```
maxProfit :: [Int] -> Int
maxProfit prices = maxProfitHelper prices (maxBound :: Int) 0
  where
    maxProfitHelper [] _ maxProfit = maxProfit
    maxProfitHelper (p:ps) minPrice maxProfit =
      let newMinPrice = min minPrice p
          profit = p - newMinPrice
          newMaxProfit = max maxProfit profit
      in maxProfitHelper ps newMinPrice newMaxProfit
main = do
    let prices = [7, 1, 5, 3, 6, 4]
    let result = maxProfit prices
    putStrLn $ "Maximum profit: " ++ show result
```

Figure 2: Test

```
Given PLAN:
STEP 1: Define a function maxProfit which takes an array of
integers as input and produces an integer as output.
STEP 2: In the main function, let the "prices" variable hold
the input array of integers [7, 1, 5, 3, 6, 4].
STEP 3: In the main function, let the "result" variable hold
the output of maxProfit given the input array of integers.
STEP 4: In the maxProfit function, initiate the conditional
checks with parameters prices, (maxBound :: Int), and 0.
STEP 5: Define two checks. Check 1 verifies whether the first
parameter is an empty list and the second parameter is the
maximum bound of the integer type. Check 2 verifies whether
the first parameter is a non-empty list. These checks are
executed in the order in which they appear. In the case of the
[7, 1, 5, 3, 6, 4] input array, given the ordering of the checks,
only check 2 is executed.
STEP 6: In the maxProfit function, define a helper function
maxProfitHelper which takes three parameters: an array of
integers, an integer representing the minimum price, and an
integer representing the maximum profit. This helper function
recursively calculates the maximum profit by updating the
minimum price and maximum profit values based on the current
element of the input array.
STEP 7: In the main function, print the resulting maximum profit
to the screen.

So the CODE COVERAGE for the given code snippet will be:
> maxProfit :: [Int] -> Int
> maxProfit prices = maxProfitHelper prices (maxBound :: Int) 0
>   where
>     maxProfitHelper [] _ maxProfit = maxProfit
>     maxProfitHelper (p:ps) minPrice maxProfit =
>       let newMinPrice = min minPrice p
>           profit = p - newMinPrice
>           newMaxProfit = max maxProfit profit
>       in maxProfitHelper ps newMinPrice newMaxProfit
> main = do
>     let prices = [7, 1, 5, 3, 6, 4]
>     let result = maxProfit prices
>     putStrLn $ "Maximum profit: " ++ show result
```

Figure 3: Output

congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

4 Results and Discussion

Achieved results, and our interpretation of them.

```
Given CODE SNIPPET:
isValidParentheses :: String -> Bool
isValidParentheses str = check str 0
  where
    check [] 0 = True
    check [] _ = False
    check (')':xs) n = n > 0 && check xs (n - 1)
    check ('(':xs) n = check xs (n + 1)
    check (_:xs) n = check xs n
main = do
  let result = isValidParentheses "(()"
  print result

Given PLAN:
STEP 1: Define a function isValidParentheses which
takes a string as input and produces a boolean as output.
STEP 2: In the main function, let the "result" variable
hold the output of isValidParentheses given the string
"(()".
STEP 3: In the isValidParentheses function, initiate the
conditional checks with parameters str and 0.
STEP 4: Define five checks. Check 1 verifies whether the
parameters are an empty list and the value 0. Check 2 verifies
whether the parameters are an empty list and any value other
than 0. Check 3 verifies whether the first character of the
first parameter is a right parenthesis. Check 4 verifies
whether the first character of the first parameter is a left
parenthesis. Check 5 verifies whether the first character of
the first parameter is any character other than a right or a
left parenthesis. These checks are executed in the order in
which they appear. We stop executing the next check as soon as
we find one which satisfies the correct condition. In the case
of the "(()" input string, given the ordering of the checks,
all checks are executed.
STEP 5: In the main function, print the resulting array to the
screen.

So the CODE COVERAGE for the given code snippet will be:
> isValidParentheses :: String -> Bool
> isValidParentheses str = check str 0
>   where
>     check [] 0 = True
>     check [] _ = False
>     check (')':xs) n = n > 0 && check xs (n - 1)
>     check ('(':xs) n = check xs (n + 1)
>     check (_:xs) n = check xs n
> main = do
>   let result = isValidParentheses "(()"
>   print result
```

Figure 4: Exemplar

5 Conclusion and Future Directions

Summary of main contributions. Why this study matters, how can our findings be used. Generally speaking we see that prompt and exemplar engineering is necessary in the sense that it really has an impact on the model's output, and they must be tailored to the specific task, the specific language.