Do Large Code Models Understand Programming Concepts? Counterfactual Analysis for Code Predicates

Ashish Hooda [1 2] **Mihai Christodorescu** [2] **Miltiadis Allamanis** [3] **Aaron Wilson** [2] **Kassem Fawaz** [1] **Somesh Jha** [1 2]

Abstract

Large Language Models' success in text generation has also made them better at code generation and coding tasks. While a lot of work has demonstrated their remarkable performance on tasks such as code completion and editing, it is still unclear as to why. We help bridge this gap by exploring to what degree auto-regressive models understand the logical constructs of the underlying programs. We propose Counterfactual Analysis for Programming Concept Predicates (CACP) as a counterfactual testing framework to evaluate whether Large Code Models understand programming concepts. With only black-box access to the model, we use CACP to evaluate ten popular Large Code Models for four different programming concepts. Our findings suggest that current models lack understanding of concepts such as data flow and control flow.

1. Introduction

Language Language Models (LLMs) have demonstrated remarkable performance on a variety of automated programming tasks, such as code completion (Austin et al., 2021; Fried et al., 2022), code repair (Jiang et al., 2021; Joshi et al., 2023), and code translation (Pan et al., 2023; Chen et al., 2023b). Automating a programming task is a complex problem that requires understanding many concepts in the underlying code. These concepts include how variables are stored, accessed, and modified in memory; how execution proceeds across various constructs; and how different parts of the code compose sequentially or in parallel to perform a computation. We refer to these concepts as *Programming-Concept Predicates* (PCPs). Despite their remarkable performance, the degree to which LLMs understand the PCPs in the programs they manipulate remains unclear.

Empirical evaluations on benchmark datasets such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and CodeContests (Li et al., 2022) drive the current understanding of the code capabilities of LLMs. While task-driven evaluation measures the end-to-end performance, it does not reveal the LLM's fine-grained understanding of PCPs. As a result, we often cannot attribute the failures in these coding tasks to specific aspects of the underlying code — was the code completion wrong due to confusing variable names, unusual control flow, inherent algorithmic complexity, or code size? Such a fine-grained attribution would allow practitioners to better reason about these models' limits and highlight the avenues to improve their performance.

In this work, we consider the problem of *evaluating a given model's understanding of programming concepts*. We focus on four PCPs that represent classical concepts in the program analysis literature (Allen, 1970; Fosdick & Osterweil, 1976; Lin & Wu, 2008; Dart & Zobel, 1992):

- ☐ *Control Flow:* The output of the automated coding task does not change with the ordering of independent code statements.
- ☐ *Data Flow:* The automated coding task uses only variables that are in scope (and live) within the coding task.
- ☐ *Data Types:* The automated coding task satisfies the constraints of the type system.
- ☐ *Identifier Naming:* Functionality of the automated coding task does not depend on the names of the variables or functions.

We introduce Counterfactual Analysis for Programming Concept Predicates (CACP), a counterfactual analysis framework for evaluating whether large code models understand PCPs. As the name suggests, CACP builds on counterfactual analysis to cast concept understanding as the problem of determining how controlled input changes result in model output changes. There are two main components of CACP– (1) Generating counterfactuals for code that only perturb specific PCPs, and (2) Using them to analyze the model's performance. Specifically for a given PCP, we define code

[1]UW-Madison [2]Google Research [3]Google DeepMind.
The majority of this work was completed while Ashish Hooda was an intern at Google Research.. Correspondence to: Ashish Hooda <ahooda@wisc.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

Input	Counterfactual Input

def max_path_sum(tri, m, n):
for i in range(m-1, -1, -1):
for j in range(i+1):
if (tri[i+1][j] > tri[i+1][j+1]):
	def max_path_sum(tri, m, n):
	for i in range(m-1, -1, -1):
	for j in range(i+1):
	if (tri[i+1][j] <= tri[i+1][j+1]):

Completion [Correct]	**Completion [Incorrect]**
tri[i][j] += tri[i+1][j] else: tri[i][j] += tri[i+1][j+1] return tri[0][0]	tri[i][j] += tri[i+1][j] else: tri[i][j] += tri[i+1][j+1] return tri[0][0]

Figure 1. In this example the counterfactual input is generated by negating the relational expression in the **if** statement. Starcoder (Li et al., 2023) generates an incorrect completion for the input on the right. This suggests that LLMs have incomplete understanding of programming concepts such as *control-flow*.

perturbations (called *mutations*) that are minimal in that they influence only one PCP, but not others. The challenge lies in defining these minimal mutations and predictably evaluating their impact on the model output. The minimality of mutations allows us to explain failures concerning specific PCPs that are not well understood by the model.

We apply our CACP framework on code completion (the most popular code task for language models) and show how to benchmark predicate understanding with only hard-label black-box access to a model. This allows us to quantify the model's coding capability through an end-to-end automated measurement of understanding of PCPs related to the task, without having to adapt the model to those predicates (e.g., without fine-tuning or using additional training data). We develop four mutations that instantiate the PCPs described above: flipping if-else conditions, swapping independent statements, breaking def-use chains, and changing variable names. Building on these mutations, we create a new benchmark dataset to evaluate how LLMs understand PCPs.

Our evaluations of ten popular LLMs reveal that state-of-art completion models have gaps in understanding PCPs, where some mutations result in more than 20% of the tasks completed with incorrect code. Figure 1 shows an example generated by our framework, where flipping an if-condition results in an incorrect code output.

In summary, our work makes the following contributions:

1. We propose CACP, a counterfactual testing framework for evaluating understanding of Programming Concept Predicates (PCPs). We show how to overcome challenges of generating counterfactual programs.

2. We apply CACP to the code completion task and test four types of PCPs. To this end, we extend three popular code datasets—HumanEval, MBPP, and CodeContests—and create a new benchmark dataset for evaluating PCP understanding in LLMs.

3. Using CACP, we evaluate ten popular LLMs and provide insights on how the model's understanding depends on different model and data parameters. We highlight the gaps in the state-of-art models' understanding of coding concepts.

2. Background and Related Work

Programming Concept Predicates and LLMs for Code. Programming Concept Predicates describe properties of specific elements of the program (variables, functions, data values, execution paths, etc.) either by themselves or in relation to other elements (Hoare, 1969). For example, a predicate may describe the range of values a variable v may take at a program location l, or whether some execution from location l_1 in function f_1 could reach location l_2 in function f_2 (these are a type of *control-flow predicates*), or whether the value assigned to variable w at location l_1 could be the value used when w is later accessed at location l_2 (a type of *data-flow predicate*). We say a program satisfies a predicate if in every possible execution of that program the predicate evaluated over the actual values of the relevant program elements is true[1].

Large language models (LLMs) have shown strong performance on a variety of code tasks, from code completion (Austin et al., 2021; Fried et al., 2022), to code translation (Pan et al., 2023; Chen et al., 2023b), and to code repair (Jiang et al., 2021; Joshi et al., 2023). A code LLM takes as input a sequence of natural-language instructions and a sequence of code statements (i.e., a partial program) and outputs another partial program (depending on the task). We consider the general case where the task of interest has an associated function (called the *attribution function*) that determines whether the output of the model satisfies the input instruction. For generative tasks for code such as code completion or code repair, it is common to use program testing as attribution function, where the output program is executed against a test suite.

The core problem we investigate is how to estimate a model's understanding of PCPs. Such an estimation can be useful to validate a model's suitability for a particular task, where the task is expected to depend (or not depend at all) on a particular predicate. For example, the task of code completion is useful only when it is sensitive to the order of program statements and thus it is expected to depend on control-flow predicates. In turn, a model trained for code completion should yield different outputs on programs with statements in different orders. If a task depends on a predicate, we want any model trained for that task to have high understanding of the predicate.

[1]For our purposes, describing PCPs as holding over all program executions is without loss of generality, as the predicate itself may limit its scope to some subset of executions.

Robustness of Code Models. Recent work has studied the robustness of code models against both natural and adversarial perturbations. Shirafuji et al. (2023) & Wang et al. (2022) study robustness of code completion against different representations for the problem description as well as the input program. Henkel et al. (2022); Jha & Reddy (2023) demonstrate that function name prediction models can be attacked using semantic preserving transformation applied to the input program. Chen et al. (2023a) have similar findings for the code summarization task. In this work, we focus on evaluating the understanding of specific programming concepts. Our approach is based on causal analysis, which involves carefully designing counterfactuals and attributing their effect. Similar to robustness, our approach also involves mutating programs and performing inference. However, our mutations are aimed at generating counterfactuals and need to ensure that the input for the original prompt and the counterfactual prompt differ only along the concept to be tested.

Counterfactual Analysis. For ML models, counterfactual analysis proceeds by performing interventions on the inputs and observing the changes in the model outputs. This can be achieved via counterfactual (CF) inputs generated by changing an input x such that only a specific concept C_k of the input is changed to a different value i.e. $x_{C_k=c'}$ is a counterfactual for input $x_{C_k=c}$ for concept C. Now, the effect of the concept on the model can be estimated by observing how the model output differs from the counterfactual. To be effective, CFs are designed to achieve three main properties (Abid et al., 2022) — (1) Correctness: CF perturbations should lead to a predictable change in the ground-truth output, (2) Validity: CFs should pertain to real world constraints, and (3) Specificity: CFs should only perturb individual properties in order to evaluate understanding of specific concepts.

In contrast to tabular and image data, generating counterfactuals has been relatively unexplored for programs. Past work on counterfactual explanations for code has looked only into syntactic perturbations and has primarily focused on finding the minimum perturbations that change the output (Cito et al., 2022). Since these perturbations do not change isolated concepts, they are more useful in explaining model behaviour for individual inputs rather than evaluating understanding of specific concepts. In contrast, we focus on both syntactic and semantic perturbations that only change programs along specific PCPs.

Independently, there has been work on counterfactual analysis of output token probabilities of large code models (Palacio et al., 2023a;b). These methods only work for the next predicted token and do not apply to outputs with multiple tokens. They also require access to the probability distribution of the output token prior to sampling. In contrast, our method works for the entire output and works in the hard label black box setting with access only to the final output.

3. Counterfactual Analysis for Programming Concept Predicates

In the following, we describe CACP, starting with the basic notation. Second, we discuss the requirements associated with counterfactual analysis for PCPs. Third, we describe how CACP addressed these challenges for four PCPs. Finally, we describe how CACP estimates the model's understanding.

3.1. Notation

Let M be a code LLM such that

$$\mathsf{M} : \mathcal{H} \times \mathcal{X} \rightarrow \mathcal{Y},$$

where $\mathcal{H}$ is the space of instructions and $\mathcal{X}, \mathcal{Y} \in \mathcal{P}$ with $\mathcal{P}$ being the space of programs. For code completion, $\mathcal{H}$ is the docstring or the problem specification in natural language, and $\mathcal{X}$ and $\mathcal{Y}$ are program prefixes and completions, respectively. An attribution function $\mathsf{A} : \mathcal{H} \times \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\}$ evaluates if the model output satisfies the instruction. Also, let $O_{h \times x} = \{y \mid y \in \mathcal{Y}, \mathsf{A}(h, x, y) = 1\}$ be the set of correct outputs for a given instruction-input pair, where $x \in \mathcal{X}, h \in \mathcal{H}$. For code completion, a common attribution function evaluates if the completed program passes the unit tests specified by the problem.

3.2. Requirements

We now describe the requirements, and related challenges, for generating counterfactual programs (Abid et al., 2022).

1. **Correctness:** A counterfactual needs to correctly solve the original task. For programs, this would mean that the perturbed program should still be able to solve the task described by the instructions. We use the task's attribution function to verify this condition. Specifically, for a model M, a counterfactual pair $x, x' \in \mathcal{X}$, associated problem description $h \in \mathcal{H}$ and corresponding attribution function A, we ensure that $|O_{h \times i}| > 0 \;\; \forall i \in \{x, x'\}$.
2. **Validity:** The generated counterfactuals also need to be valid, i.e., they need to pertain to real-world constraints. This means that the perturbed programs should be syntactically correct. Furthermore, they should be "natural," i.e., in distribution with programs seen in the software development pipeline (Hindle et al., 2016).
3. **Specificity:** Counterfactual perturbations should only change specific attributes/concepts in the input, which is especially challenging for programs. Formally, let $Preds(x)$ be the infinite set of all PCPs that a program $x \in \mathcal{X}$ satisfies. Note that $Preds(x)$ is infinite because

for any predicates p_1 and p_2 in $Preds(x)$, the predicates $\mathsf{p}_1 \vee \mathsf{p}_2$ and $\mathsf{p}_1 \wedge \mathsf{p}_2$ are also in $Preds(x)$. This implies that any mutation applied to the program x cannot affect exactly one predicate $\mathsf{p} \in Preds(x)$, but rather it affects a subset of $Preds(x)$. Therefore, for programs, we relax this requirement by considering counterfactuals that affect only a minimal set of PCPs.

3.3. Mutations for Counterfactual Programs

Now, we discuss how CACP generates counterfactual programs that satisfy the above requirements. CACP automates the CF generation process using *mutations*. These are transformation functions that perturb programs with respect to specific concepts, i.e., $\mathsf{T}_{\mathsf{p}_k} : \mathcal{X} \rightarrow \mathcal{X}$ where p_k is the target PCP. A PCP can have more than one associated mutation. Given an input program $x \in \mathcal{X}$, the mutation function is then used to generate a counterfactual $x_{\mathsf{p}_k} = \mathsf{T}_{\mathsf{p}_k}(x) \in \mathcal{X}$. Our comprehensive review of the program analysis literature revealed four themes of studied program predicates: control flow (Allen, 1970; Yang et al., 2015), data flow (Fosdick & Osterweil, 1976; Nilsson-Nyman et al., 2009), identifier names (Lin & Wu, 2008), and data types (Dart & Zobel, 1992; Allamanis et al., 2020). As we study weakly typed programs (for instance, Python), we consider four distinct PCPs that cover the first three themes. Next, we show how CACP automates the generation of these four distinct PCPs (also illustrated in Figure 2).

If-Else Flip: We use a mutation that swaps the branches of an **`if-else`** statement and negates the condition to test for the PCP: *Inverting the relational expression of an if-else block flips the ordering of the then and else bodies*. It involves two steps: Negating the test condition of the **`if-else`** statement using DeMorgan's law and swapping the `then` body with the **`else`** body. This mutation satisfies – (1) Correctness: The counterfactual still solves the task since it is semantically equivalent to the input; (2) Validity: We negate the relational expression by using complementary operators, for example, we substitute `x==y` with `x!=y`; (3) Specificity: We ensure that we do not affect other PCPs by only applying this perturbation to relational expressions that do not include any method calls that might change the state of the program.

Independent Swap: Next, we evaluate the PCP: *Code Completion is invariant to the ordering of independent statements*. This mutation swaps pairs of independent statement blocks in the program. We use data-flow analysis to identify pairs of independent blocks. This mutation satisfies – (1) Correctness: Since we only swap independent blocks, the perturbed program is semantically identical and still solves the problem; (2) Validity: Ordering of independent statements does not change the "naturalness" of the program; (3) Specificity: Our data-flow analysis ensures that we only swap statements where the ordering does not affect any other PCP.

Def-Use Break: We design a mutation that breaks def-use chains to evaluate the PCP: *Breaking a def-use chain alters the scope of variables*. Def-Use chains capture the relationship between the definitions of variables (where a variable is assigned a value) and their subsequent uses (where that value is accessed or modified). To break a def-use chain, we substitute a variable's second chain with a new name (a random string of 5 characters), i.e., we simply rename the second definition and all subsequent uses. For example, in Figure 2, we rename the second chain of variable `list1`. This mutation satisfies – (1) Correctness: we ensure that the counterfactual is semantically equivalent and still solves the problem by consistently substituting all subsequent occurrences; (2) Validity: Random strings are often used as identifiers in obfuscated or minified versions of programs (Tran et al., 2019); (3) Specificity: We use def-use analysis to identify and perturb individual chains.

Variable-Name Invariance: Next, we evaluate the PCP: *Variable names do not affect the semantics of a program*. Here, we generate counterfactuals by renaming variables. We consider two variants of this mutation — renaming to random strings and permuting or shuffling existing names between variables. For the first variant, we substitute variable names with randomly generated strings of five characters. For the second variant, we shuffle names among the variables defined in the program. This mutation satisfies – (1) Correctness: we ensure that the counterfactual is semantically equivalent by consistently substituting each variable; (2) Validity: We only substitute user-defined variables and do not rename reserved keywords; (3) Specificity: We do not substitute function parameters as their names decide the order in which arguments are passed during invocation.

3.4. Measuring Counterfactual Effect

We need a way to analyze the effect of mutations on the observed output. For a single program $x \in \mathcal{X}$, instruction $h \in \mathcal{H}$, attribution function A, and model M, we formulate the mutation effect (ME) as:

$$\mathsf{ME}^{\mathsf{M}}_{(\mathsf{p}_k,h,x)} = |\mathsf{A}(h, x_{\mathsf{p}_k}, \mathsf{M}(h, x_{\mathsf{p}_k})) - \mathsf{A}(h, x, \mathsf{M}(h, x))|$$

For code completion, a model that understands: *Variable names do not affect the semantics of a program* would generate a correct completion even for the renamed program, leading to a mutation effect of 0. A model that relies on variable names might generate erroneous completions, leading to a mutation effect of 1. To compute the ME across all programs, we define the Average Mutation Effect (AME):

$$\mathsf{AME}^{\mathsf{M}}_{\mathsf{p}_k} = \mathop{\mathbb{E}}_{h,x \in \mathcal{H},\mathcal{X}} \left[\mathsf{ME}^{\mathsf{M}}_{(\mathsf{p}_k,h,x)}\right]$$

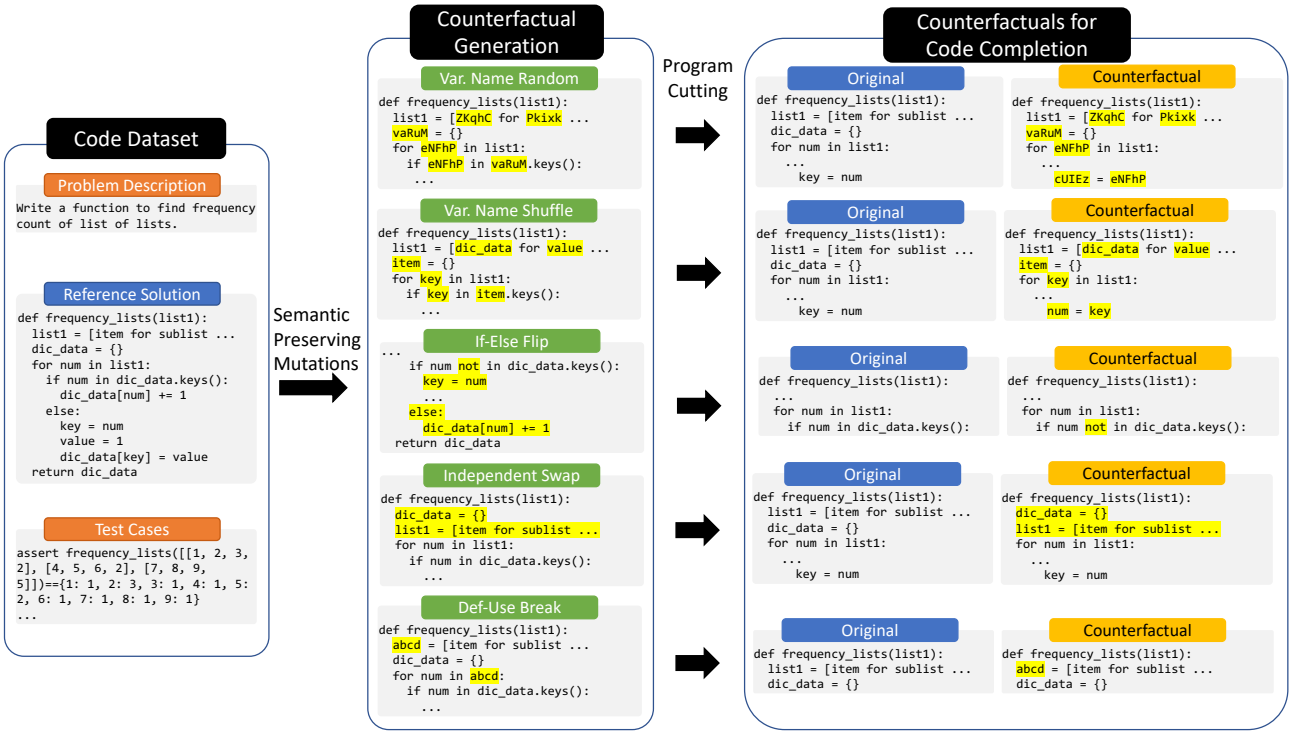

Figure 2. The counterfactual generation pipeline of CACP. It consists of two stages. First, the reference solution for the problem is perturbed using predicate-specific mutations. Second, both the original and the perturbed solution are cut at the same location to generate a pair of counterfactual inputs.

AME with a small magnitude indicates a better understanding of the PCP. On the other hand, a large magnitude indicates poor understanding since the model performs worse after the mutation. Note that this formulation is similar to the Average Treatment Effect used in counterfactual analysis (Pearl, 2009). The treatment Effect is defined for the output of the model, whereas we compute the Mutation Effect using the attribution function.

4. CACP for Code Completion

In this section, we instantiate CACP for the Code Completion task. We first briefly describe the code completion task. Then, we demonstrate how CACP generates counterfactuals for code completion for the four PCPs. Finally, we describe how we measure the mutation effect.

4.1. Large Language Models for Code Completion

Code completion tasks, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), have become instrumental in evaluating the capabilities of code completion models. These tasks challenge models with an array of programming tasks designed to test different aspects of coding proficiency. In these benchmarks, problems are presented as Python *function skeletons* with accompanying *descriptions* that specify what the function should accomplish, along with *unit tests* to validate the correctness of the generated code. Each problem in these benchmarks is also accompanied by a *reference solution* that acts as a gold standard, allowing for direct comparison between model-generated code and the expected output.

While HumanEval and MBPP excel in testing a model's ability to generate syntactically and semantically correct code, they do not assess the model's understanding of PCPs. To address this gap, CACP extends these datasets by using reference solutions as a base and generating counterfactuals that can be used to evaluate the understanding of specific PCPs.

4.2. CACP Counterfactual Generation

CACP generates counterfactuals for code completion using a two-step procedure: (1) Reference solutions are transformed using mutations specified in Section 3 to generate mutated solutions, and (2) Reference and mutated solutions are cut at the same location to create partial programs which act as counterfactual inputs. Additionally, we test these mutated solutions by compiling and executing them to confirm that

they pass the required test cases. Below, we describe how we cut the solutions for each mutation (also illustrated in Figure 2):

If-Else Flip: We cut both the reference solution as well as the perturbed solution at the beginning of the `then` body. As shown in Figure 2, this generates partial programs which end at a statement of the form - `if` <condition> and the relational condition for the counterfactual is the negation of the original.

Independent Swap: We only consider mutations where both the swapped statements are part of the initial 75% of the program. Then, we cut the trailing 25%, and the remaining acts as the input for the code completion task. Note that the cutting for both the original and the counterfactual happens at the same location since the ordering of statements after the swapped pair does not change.

Def-Use Break: We only consider mutations where the perturbed chain is at least partially present in the initial 75% of the program. Then, we cut trailing 25% for both the original and the counterfactual. This ensures that counterfactual input is not identical to the original. Note that the cutting happens at the same location since renaming the variable does not affect the line numbers of statements.

Variable-Name Invariance: We only consider mutations where at least one variable appearance is renamed in the initial 75% of the program. This ensures that counterfactual input is not identical to the original. We cut off the trailing 25% and use the rest as the counterfactuals.

4.3. CACP Effect Measurement

There are two primary approaches to evaluating the generations of a code-completion task—testing and exact string matching. Exact string-matching techniques like CodeBLEU (Ren et al., 2020) and chrF (Popović, 2015) evaluate generations by computing the distance from the reference solution. However, such match-based metrics are unable to account for the large space of programs that are functionally equivalent to, yet syntactically distinct from, a reference solution and thus underestimate the capabilities of a model that understands programming concepts. Testing provides a more direct evaluation, where a generation is deemed correct if it passes all the unit tests for that code-completion instance. Therefore, we use *unit-test correctness* as the attribution function for computing the AME. We generate candidate solutions by querying the model on both the original input as well as the counterfactual. Then, we execute the candidate solutions against the test cases, resulting in one of two outcomes: passing all test cases or at least one failure. Note that we only consider problems where the model generates a successful completion (i.e. passing all test cases) for the original (non-perturbed) input, the perturbed input, or both. The cases where the model fails for both the original and perturbed inputs are not necessarily informative about the impact of the PCP, and we discard them. In that case, the perturbed inputs are not considered as counterfactual.

Table 1. Number of valid counterfactual pairs per mutation type.

Mutation	Counterfactual Pairs		
	HumanEval+MBBP	CC	Total
Var. Name Random	724	1000	1724
Var. Name Shuffle	724	1000	1724
If-Else Flip	103	1000	1103
Independent Swap	624	1000	1624
Def-Use Break	22	277	299

5. Experiments

Using CACP, we evaluate ten popular Large Language Models against five different mutations. Our evaluation answers the following questions.

Q1: How are leading LLMs affected by counterfactual mutations?
We evaluate ten popular LLMs and show that they suffer significant drops in unit test correctness for mutations on Variable-Names, IfElse-Flip, and DefUse-Break, leading to AMEs as high as 34%. The effect is smaller in magnitude for Independent-Swap. Overall, these results suggest that current models lack understanding of program predicates.

Q2: How does the Average Mutation Effect (AME) depend on LLM parameters?
We observe that understanding of predicates seems to improve with model size. Training or fine-tuning on code-specific data also seems to improve understanding, specifically for variable name-related predicates.

Q3: Are the errors related? What do they depend on?
We analyze the correlation between pairs of mutations and show that all pairs exhibit low correlation apart from the two Variable Names mutations. In the case of StarCoder (Li et al., 2023), our analysis suggests a relation between AME for the IfElse-Flip mutation and the frequency of appearance of different relational operators in the model's training data.

5.1. Experimental Setup

We use the following settings to demonstrate how CACP evaluates understanding of programming concepts.

Datasets and mutations. We instantiate CACP using three popular code generation benchmarks — HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and CodeContests (Li et al., 2022). All of the problems in these datasets include a reference solution, which is used to generate counterfactual pairs as described in section 4. Since not every

Table 2. We compute the AME using the Pass/Fail attribute function as described in subsection 4.3. We only consider problems where the model achieves non zero accuracy on either the original or the counterfactual setting.

			Average Mutation Effect (AME)				
Dataset	**Model**	**Original Accuracy**	**Variable-Names Random**	**Variable-Names Shuffle**	**IfElse-Flip**	**Independent-Swap**	**DefUse-Break**
HumanEval + MBPP	Starcoder (13B)	66.04 %	16.86 %	19.42 %	21.07 %	07.47 %	05.00 %
	Llama 2 (7B)	43.20 %	24.58 %	29.08 %	25.18 %	13.45 %	21.88 %
	Llama 2 (13B)	48.40 %	21.14 %	26.84 %	20.00 %	09.12 %	15.88 %
	Llama 2 (70B)	63.37 %	14.37 %	19.81 %	20.83 %	05.54 %	06.50 %
	Llama Code (7B)	60.10 %	19.84 %	21.44 %	17.71 %	10.88 %	05.00 %
	Llama Code (13B)	66.61 %	12.56 %	18.06 %	16.62 %	05.04 %	09.50 %
	Llama Code (34B)	72.65 %	12.55 %	15.14 %	17.09 %	04.76 %	07.62 %
	PaLM 2 (64B)	45.74 %	23.75 %	22.58 %	25.00 %	12.96 %	19.38 %
	PaLM 2 (340B)	66.98 %	14.71 %	17.70 %	19.72 %	06.13 %	17.00 %
	PaLM 2-S^* (24B)	70.01 %	12.31 %	19.74 %	16.09 %	06.51 %	11.90 %
	GPT4 (gpt-4-1106)	88.94 %	06.43 %	07.21 %	05.95 %	01.57 %	04.76 %
Code Contests	Starcoder (13B)	43.75 %	16.90 %	21.18 %	30.93 %	06.43 %	22.92 %
	Llama 2 (7B)	24.75 %	29.14 %	25.38 %	29.72 %	13.24 %	34.07 %
	Llama 2 (13B)	29.48 %	23.78 %	23.86 %	29.52 %	09.26 %	23.98 %
	Llama 2 (70B)	40.18 %	17.19 %	18.20 %	28.58 %	09.14 %	26.04 %
	Llama Code (7B)	38.74 %	22.16 %	21.62 %	26.95 %	09.21 %	20.23 %
	Llama Code (13B)	40.66 %	21.45 %	22.52 %	32.53 %	07.48 %	29.40 %
	Llama Code (34B)	49.55 %	16.53 %	18.09 %	32.02 %	07.04 %	26.60 %
	PaLM 2 (64B)	38.75 %	18.18 %	21.53 %	26.43 %	08.06 %	23.11 %
	PaLM 2 (340B)	47.27 %	15.57 %	17.90 %	27.31 %	07.58 %	18.56 %
	PaLM 2-S^* (24B)	47.28 %	13.22 %	15.59 %	29.37 %	05.48 %	18.25 %
	GPT4 (gpt-4-1106)	67.83 %	11.25 %	16.48 %	14.89 %	05.05 %	21.58 %

mutation applies to all reference solutions, the final number of counterfactual pairs differs based on the mutation type. As shown in Table 1, mutations related to Variable Names can be applied to almost all solutions, whereas mutations related to control-flow or def-use are more selective. In this evaluation, we focus on Python programs, but our methodology applies to any programming language. We use `libCST` (LibCST Developers) for parsing and manipulating source code for our mutations.

Models. We use CACP to evaluate popular models, including Llama 2 (Touvron et al., 2023) and PaLM (Anil et al., 2023). We also evaluate counterparts of these models that are fine-tuned for coding tasks – Code Llama (Roziere et al., 2023) and PaLM 2-S^* (Anil et al., 2023). Finally, we also evaluate the popular open source code LLM StarCoder (Li et al., 2023). We set the sampling temperature to 0 for all models to have deterministic results.

5.2. Average Mutation Effect

Table 2 shows the AME for the three datasets, five mutations, and ten models. The table shows that the original unit test correctness rates vary across models. AME values are non-zero, which suggests that models do not fully understand the evaluated PCPs. In the case of the Variable-Names and IfElse-Flip perturbations, AME values are as high as 33%. On the other hand, the Independent-Swap mutation is the most well-understood. While most mutations have similar effects across the two kinds of datasets, the DefUse-Break perturbation shows a relatively lower effect on the HumanEval and MBPP datasets. This is likely due to the small number of valid problems — only 22.

Across Models: For Variable-Name related perturbations, we first observe that smaller models perform worse and larger models do better. This is evident in Figure 3, which shows the AME as a function of the model size. Secondly, models trained on code (StarCoder) or fine-tuned on code (Llama Code, PaLM 2-S^*) perform better than models that are not. Perturbations related to control flow and data flow follow a similar trend for model size, but code fine-tuning does not always seem to improve performance. GPT4 performs much better on HumanEval and MBPP, but is similar to the other models for the CodeContests dataset.

Correlation across Mutations: Until now, we have seen the average effect of the perturbations across the datasets. Figure 4 shows the correlation between different perturbation types. As expected, the two Variable-Names perturbations correlate highly. Other perturbations have fairly low correlation, suggesting that our mutations are predicate-specific and have minimal correlated errors.

Errors due to Memorization: We performed an additional

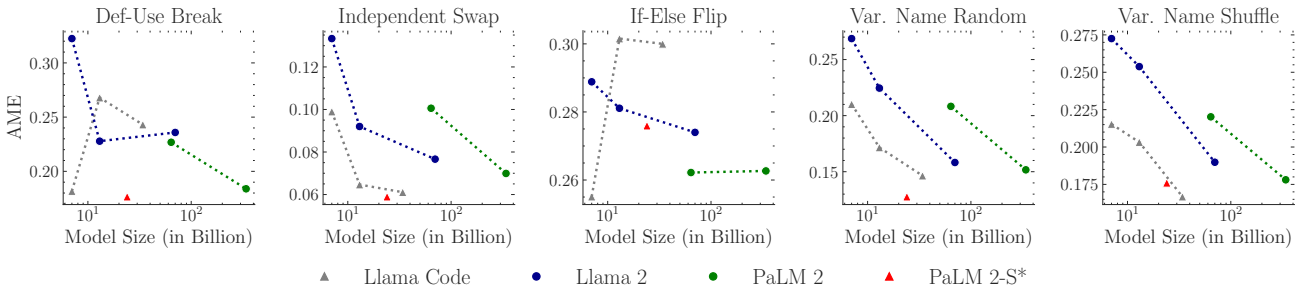

Figure 3. AME as a function of model size (number of parameters in Billions). The different model classes are depicted using different colors.

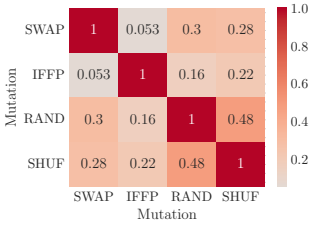

Figure 4. Correlation between AME values across pairs of mutations. The number of samples used to compute each value depends on the size of the intersection of the two mutation types. Independent-Swap: SWAP, IfElse-Flip: IFFP, Variable Names Random: RAND, Variable Names Shuffle: SHUF

Table 3. Memorization Analysis for the `If-Else` mutation for Starcoder. We parse Starcoder's training data and show the relative frequency of appearance of pairs of complementary relational operators. We also show the average change in unit test correctness computed over all valid programs in HumanEval, MBPP and CodeContests.

Op A	Op B	Ratio	Δ(A→B)	Δ(B→A)
==	! =	3.9	13.21 %	07.37 %
>	<=	3.8	16.92 %	01.48 %
<	>=	2.2	05.00 %	0.00 %

experiment to gain some insights on whether memorization (Carlini et al., 2022) contributes to the observed mutation effects. For the If-Else perturbation, we analyze the connection between the frequency of appearance of relational operators in the training set and their respective change in unit test correctness. We perform this analysis with StarCoder's training data (Husain et al., 2019). More specifically, in Table 3, we show the relative frequency of complement relational operators and the change in correctness values when substituted. We can see that operators that appear more frequently in the training set face a significantly higher drop in correctness when they are being substituted.

Effect of cutoff point: In subsection 4.2, we describe that we keep 75% of the program as the prefix for generating counterfactuals for the Independent-Swap, DefUse-Break and Variable Name Invariance mutations. To study the effect of the cutoff point, we evaluated counterfactuals generated using the same set of programs but cut at different places. In Table 4, we present the original accuracy and the AME for the Starcoder model. This does not include the IfElse-Flip mutation since in that case the cut depends on the location of the If block. We find that an earlier cut leads to a decrease in the original accuracy as well as a higher AME. This is expected since cutting earlier increases the complexity of the completion task. However, we observe that AME is relatively more stable than the original accuracy. This suggests that AME is a good measure of the model's understanding, irrespective of the complexity of the coding task.

Code Repair Task: We also evaluate CACP for the code repair task. We use the HumanEvalPack (Muennighoff et al., 2023) dataset which is an extension of HumanEval to also include the Code Repair task. This dataset is constructed by manually adding a bug to each solution in HumanEval. For this task, the model is tasked with fixing the bug and generating the correct solution. In this case, we generate counterfactuals by applying mutations on the buggy solution. In Table 5, we show the performance of Octocoder (Muennighoff et al., 2023) which is an instruction-tuned version of Starcoder. Similar to code completion, we observe a high average mutation effect which suggests a lack of understanding for the Code Repair task as well.

6. Future Work

Automating Semantic Preserving Perturbations. Currently, crafting these perturbations requires a significant amount of manual effort and deep domain knowledge to ensure they do not alter the underlying logic of the program and only change specific predicates. Developing automated tools and techniques that can reliably generate such perturbations will not only streamline the evaluation process but also enhance the scalability of our testing framework.

Perturbation-based Data Augmentation. A promising

Table 4. AME for different cutoff settings when evaluating Starcoder. A lower prefix ratio implies an earlier cut. Independent-Swap: SWAP, Variable Names Random: RAND, Variable Names Shuffle: SHUF, DefUse-Break: DUBR

Prefix Ratio	Orig. Acc.	RAND	SHUF	SWAP	DUBR
0.4 - 0.6	32.8 %	18.7 %	21.4 %	08.0 %	26.0 %
0.6 - 0.8	50.1 %	16.7 %	19.6 %	25.0 %	06.0 %
0.8 - 1.0	60.1 %	12.4 %	17.7 %	19.6 %	04.0 %

Table 5. AME for the code repair task. We evaluate OctoCoder (Muennighoff et al., 2023) on countefactuals generated on the code repair benchmark from HumanEvalPack. Independent-Swap: SWAP, Variable Names Shuffle: SHUF, IfElse-Flip: IFFP

Original Accuracy	SHUF	IFFP	SWAP
15 %	36 %	33 %	15 %

area of future work is the application of perturbations to data augmentation to reduce the mutation effect observed in models. By systematically introducing perturbed data during the training phase, models could potentially develop a more nuanced understanding of code, reducing their susceptibility to errors. This approach requires careful consideration to balance the augmentation process without introducing bias or overly diluting the training data.

Expanding Counterfactual Analysis with Diverse Code Datasets. Our framework would benefit from adding more code datasets including ones that may not support test-based attribution functions (Lu et al., 2021; Husain et al., 2019). This would also help increase the number of input samples for more selective perturbations like def-use chains. However, in absence of test cases, this would require the development of specialized attribution functions. Moreover, careful attention must be paid to the provenance of the data to avoid contamination of the evaluation set with examples that may have been part of the model's training set.

7. Conclusion

In conclusion, we explore whether Large Code Models understand programs and propose CACP, a counterfactual testing framework for evaluating understanding of program predicates. CACP builds upon existing code datasets and requires only hard-label, black-box access to the model. We use CACP to evaluate ten popular large code models and demonstrate that current models suffer from accuracy drops up to 33% due to lack of understanding of program predicates related to control-flow and data-flow.

Acknowledgements

We thank Saswat Anand, Sajjad Arshad and Fengguo Wei for their help with evaluating Google models. We also thank John Cyphert, Jordan Henkel, Zi Wang and Neal Mangaokar as well as the anonymous reviewers for their valuable feedback.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here

References

Abid, A., Yuksekgonul, M., and Zou, J. Meaningfully debugging model mistakes using conceptual counterfactual explanations. In *International Conference on Machine Learning*, pp. 66–88. PMLR, 2022.

Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pp. 91–105, 2020.

Allen, F. E. Control flow analysis. *ACM Sigplan Notices*, 5 (7):1–19, 1970.

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., et al. PaLM 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Carlini, N., Ippolito, D., Jagielski, M., Lee, K., Tramer, F., and Zhang, C. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, N., Sun, Q., Wang, J., Gao, M., Li, X., and Li, X. Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023a.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023b.

Cito, J., Dillig, I., Murali, V., and Chandra, S. Counterfactual explanations for models of code. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 125–134, 2022.

Dart, P. W. and Zobel, J. Efficient run-time type checking of typed logic programs. *The Journal of Logic Programming*, 14(1-2):31–69, 1992.

Fosdick, L. D. and Osterweil, L. J. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8 (3):305–330, 1976.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Henkel, J., Ramakrishnan, G., Wang, Z., Albarghouthi, A., Jha, S., and Reps, T. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 526–537. IEEE, 2022.

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Jha, A. and Reddy, C. K. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 14892–14900, 2023.

Jiang, N., Lutellier, T., and Tan, L. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173. IEEE, 2021.

Joshi, H., Sanchez, J. C., Gulwani, S., Le, V., Verbruggen, G., and Radiček, I. Repair is nearly generation: Multilingual program repair with llms. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):5131–5140, 2023.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

LibCST Developers. LibCST documentation. URL https://libcst.readthedocs.io/en/latest/.

Lin, J.-C. and Wu, K.-C. Evaluation of software understandability based on fuzzy matrix. In *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*, pp. 887–892. IEEE, 2008.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., Von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Nilsson-Nyman, E., Hedin, G., Magnusson, E., and Ekman, T. Declarative intraprocedural flow analysis of java source code. *Electronic Notes in Theoretical Computer Science*, 238(5):155–171, 2009.

Palacio, D. N., Cooper, N., Rodriguez, A., Moran, K., and Poshyvanyk, D. Toward a theory of causation for interpreting neural code models. *arXiv preprint arXiv:2302.03788*, 2023a.

Palacio, D. N., Velasco, A., Rodriguez-Cardenas, D., Moran, K., and Poshyvanyk, D. Evaluating and explaining large language models for code using syntactic structures. *arXiv preprint arXiv:2308.03873*, 2023b.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., and Jabbarvand, R. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109*, 2023.

Pearl, J. Causal inference in statistics: An overview. *Statistics Surveys*, 3(none):96 – 146, 2009. doi: 10.1214/09-SS057. URL https://doi.org/10.1214/09-SS057.

Popović, M. chrF: character n-gram F-score for automatic MT evaluation. In Bojar, O., Chatterjee, R., Federmann, C., Haddow, B., Hokamp, C., Huck, M., Logacheva, V., and Pecina, P. (eds.), *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pp. 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/W15-3049. URL https://aclanthology.org/W15-3049.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Shirafuji, A., Watanobe, Y., Ito, T., Morishita, M., Nakamura, Y., Oda, Y., and Suzuki, J. Exploring the robustness of large language models for solving programming problems. *arXiv preprint arXiv:2306.14583*, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Tran, H., Tran, N., Nguyen, S., Nguyen, H., and Nguyen, T. N. Recovering variable names for minified code with usage contexts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1165–1175, 2019. doi: 10.1109/ICSE.2019.00119.

Wang, S., Li, Z., Qian, H., Yang, C., Wang, Z., Shang, M., Kumar, V., Tan, S., Ray, B., Bhatia, P., et al. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*, 2022.

Yang, S., Yan, D., Wu, H., Wang, Y., and Rountev, A. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pp. 89–99. IEEE, 2015.