Toward a Theory of Causation for Interpreting Neural Code Models

David N. Palacio, *Student Member, IEEE,* Alejandro Velasco, *Graduate Student Member, IEEE,* Nathan Cooper, *Member, IEEE,* Alvaro Rodriguez, Kevin Moran, *Member, IEEE,* Denys Poshyvanyk, *Fellow, IEEE*

Abstract—Neural Language Models of Code, or Neural Code Models (NCMs), are rapidly progressing from research prototypes to commercial developer tools. As such, understanding the capabilities and limitations of such models is becoming critical. However, the abilities of these models are typically measured using automated metrics that often only reveal a portion of their real-world performance. While, in general, the performance of NCMs appears promising, currently much is unknown about how such models arrive at decisions. To this end, this paper introduces do_{code}, a post hoc interpretability method specific to NCMs that is capable of explaining model predictions. do_{code} is based upon causal inference to enable programming language-oriented explanations. While the theoretical underpinnings of do_{code} are extensible to exploring different model properties, we provide a concrete instantiation that aims to mitigate the impact of *spurious correlations* by grounding explanations of model behavior in properties of programming languages. To demonstrate the practical benefit of do_{code}, we illustrate the insights that our framework can provide by performing a case study on two popular deep learning architectures and ten NCMs. The results of this case study illustrate that our studied NCMs are sensitive to changes in code syntax. All our NCMs, except for the BERT-like model, statistically learn to predict tokens related to blocks of code (*e.g.,* brackets, parenthesis, semicolon) with less confounding bias as compared to other programming language constructs. These insights demonstrate the potential of do_{code} as a useful method to detect and facilitate the elimination of confounding bias in NCMs.

Index Terms—Causality, Interpretability, Neural Code Models.

1 Introduction

The combination of large amounts of freely available code-related data, which can be mined from open source repositories, and ever-more sophisticated deep learning architectures for code, which we refer to as Neural Code Models (NCMs), have fueled the development of software engineering (SE) tools with increasing effectiveness. NCMs have (seemingly) illustrated promising performance across a range of different SE tasks [1], [2], [3], [4], [5], [6], [7]. In particular, *code generation* has been an important area of SE research for decades, enabling tools for downstream tasks such as code completion [8], program repair [9], and test case generation [1]. In addition, industry interest in leveraging Large Language Models (LLMs), a scalable version of NCMs, has also grown as evidenced by tools such as Microsoft's IntelliCode [10], Tabnine [11], OpenAI's Codex [12], and GitHub's Copilot [13]. Given the prior popularity of code completion engines within IDEs [14] and investment in commercial tools, NCMs for code generation will almost certainly be used to help build production software systems in the near future if they are not being used already.

Recently, there has been increased interest in evaluating NCMs for code generation. A recent study from Chen et al. [15] illustrates that certain issues, such as alignment failures and biases, do exist for large-scale NCMs. Most of the conclusions from Chen et al.'s study were uncovered through manual analysis, *e.g.,* through sourcing counterexamples, making it difficult to rigorously quantify or to systematically apply such an analysis to research prototypes [16]. Given the rising profile and role that NCMs for code generation play in SE and the current limitations of adopted evaluation techniques, it is clear that *new methods are needed to provide deeper insight into NCMs' prediction performance*.

Much of the quality assessment work on NCMs has primarily concentrated on accuracy-based metrics (*e.g.,* Accuracy, BLEU, METEOR, ROUGE) as opposed to multi-metric evaluations (*e.g.,* robustness, fairness, bias, efficiency). Moreover, skepticism within the NLP research community is growing regarding the efficacy of current accuracy-based metrics, as these metrics tend to overestimate model performance [17], [18], [19]. Even benchmarks that span multiple tasks and metrics have been shown to lack robustness, leading to incorrect assumptions of model comparisons [20]. Notable work has called for a more systematic approach that aims to understand a given model's behavior according to its linguistic capabilities [17], while others have suggested the need for holistic evaluations of Language Models [21].

In addition to limitations with current methods of model quality assessment, some of the most popular NCMs have been adapted from the field of Natural Language Processing (NLP), and thus may inherit the various limitations often associated with such models — including biases, memorization, and issues with data inefficiency, to name a few [22]. However, perhaps the most problematic aspect of current

- *David Nader Palacio, Alejandro Velasco, Nathan Cooper, and Denys Poshyvanyk are with the Department of Computer Science, William & Mary, Williamsburg, VA, 23185.*
  E-mail: danaderpalacio,svelascodimate,nacooper,dposhyvanyk{@wm.edu}
- *Kevin Moran is with the Department of Computer Science, George Mason University, Fairfax, VA, 22030.*
  E-mail: kpmoran@gmu.edu
- *Alvaro Rodriguez is with the Department of Computer Science, Universidad Nacional de Colombia, Colombia, Bogota.*
  E-mail: aldrodriguezca@unal.edu.co

Manuscript received January 16th, 2023;

neural language models is the fact that they are incapable of explaining their reasoning [23], [24], [25], [26]. NCMs and, in general, *deep learning architectures* seemingly trade effectiveness for transparency, as the same complexity that allows for impressive learning and generalization leads models to operate in a *black-box* fashion. That is, we are uncertain how neural models — including NCMs— *arrive at decisions*; a phenomenon described as *incompleteness in problem formalization* [27]. Such incompleteness manifests as an inability to explain models' predictions in human-understandable terms. If neural models fail at justifying their outputs, *Can we trust these models? Will these models work in deployment? How brittle are neural models in practical software engineering settings?* Researchers and practitioners require neural models to be robust not only at making predictions but also in the **interpretability** of those predictions [26]. Despite the increasing popularity and apparent effectiveness of code generation tools based on NCMs, there is still much that is unknown about their behavior.

A Motivating Example for Interpretability. Consider the scenario in which we *observe* that a given NCM is underperforming when predicting code from code inputs that are buggy. That is, we observe a negative correlation between *buggy code* in the prompt and *accuracy* after an initial exploratory analysis on this NCM. However, a simple observation of a correlation between *buggy code* and *accuracy* is insufficient evidence to explain the NCMs prediction performance in this scenario, as different *properties* of the buggy input code could be the *cause* of the observation. To assert that *input code being buggy* is an explanation or *interpretation* for having a worsened accuracy, we must first establish a **causal relationship** between these two random variables.

Statistical dependencies are induced by an underlying causal process (*i.e.,* Reichenbach's common cause principle [28]). In consequence, the correlation between *buggy code* and *accuracy* has three possible causal explanations[1] depicted as causal graphs in Fig. 1: (a) *accuracy* is caused by *buggy code*, (b) *buggy code* is caused by *accuracy*, and (c) the reason for a correlation is a third variable. This variable, known as a confounder, causally influences *buggy code* and *accuracy*. Based on our *domain knowledge*, we may posit that option (c) is most likely to represent our causal assumptions, as many factors such as the *number of tokens or subwords* can affect both variables of interest.

If a causal connection indeed exists between *buggy code* and *accuracy*, we can claim that the setting *buggy code* is an *interpretation* for code predictions. However, in the above example, we would essentially be guessing, and as such the question remains: *how can we quantify the causal effect of buggy code on NCM's accuracy after controlling for the influence of hidden confounders?* Unfortunately, when attempting to understand the prediction performance of NCMs, no causal interpretability formalism is available to articulate or even answer the previous questions.

In this paper, we cast this problem of achieving a more complete understanding of Neural Code Models as a *Causal Interpretability* task and posit that we can leverage the *theory of causation* as a mechanism to explain NCMs prediction performance. This mechanism includes a formalism to articulate and answer causal queries. We hypothesize that this mechanism can serve as a useful verification tool to fulfill desiderata that we might want of NCMs. These desiderata, which are originally suggested in NLP literature, comprise notions such as facilitating debugging, detecting biases, providing recourse, and eventually, increasing the reliability and trust of NCMs [27], [30], [31]. As such, this paper introduces do_{code}, a novel global post hoc interpretability method specifically designed for understanding the effectiveness of NCMs using causal queries. Furthermore, do_{code} intends to establish a robust and adaptable methodology for *interpreting predictions* of NCMs in contrast to simply *measuring the accuracy* of these same NCMs.

do_{code} asks causal questions as to why prediction performance is affected by software engineering-based *interventions*. These interventions are generally data or model properties (*e.g.,* Bugginess of Code, Number of Inline Comments, Number of Model Layers) that a domain expert might think are affecting a given NCM. More specifically, do_{code} consists of four major conceptual steps, (1) **modeling** a *structural causal graph*, (2) **identifying** causal estimand, (3) **estimating** causal effects, and (4) **validating** the causal process by refuting the obtained effect estimate and vetting the graph creation. Through the introduction of this interpretability method, we aim to help SE researchers and practitioners by allowing them to understand the potential limitations of a given model, work towards improving models and datasets based on these limitations, and ultimately make more informed decisions about how to build automated developer tools given a more holistic understanding of ***what*** NCMs are predicting and ***why*** the predictions are being made.

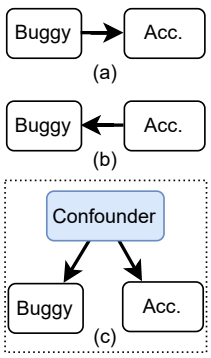


Fig. 1: Common Cause Principle in NCMs [28]

To showcase the types of insights that do_{code} can uncover, we perform a comprehensive study on seven practical *interpretability scenarios* (see Sec. 9) on different variations of popular deep learning architectures for the task of code generation, namely RNNs [32] and Transformers [33] trained on the CodeSearchNet dataset [34]. We instantiated our study (see Fig. 9) using configuration criteria such as type of architecture (*e.g.,* RNN, GPT, BERT), evaluation dataset (*e.g.,* codexglue, codesearchnet, galeras), intervention modality (*e.g.,* binary, linear), hyperparameter interventions (*e.g.,* layers, units), data interventions (*e.g.,* bug fixing, inline comments, semantic preserving), potential outcomes (*e.g.,* cross-entropy, next token predictions, Jaccard), causal inference metric (*e.g.,* Pearson, Jensen Shannon, Average Treatment Effect (ATE)), refutation testing (*e.g.,* placebo, unobserved common cause, random common cause), and syntactic clustering. This syntactic clustering comprises ten keyword-based code categories derived from Java and eleven grammar-based categories derived from Python (see Fig. 11).

1. Pearl poses an additional causal explanation based on conditional independence in colliders used for the "back-door criterion" [29].

Our case study design resulted in several notable findings illustrating the usefulness of our interpretability technique. Our most relevant findings are as follows:

- The presence or absence of buggy code does not appear to causally influence (or explain) the prediction performance of our NCMs even under measured high correlation;
- The presence of Type II Clones in training data impacts (or causally explains) the effectiveness of NCMs in terms of cross-entropy. Our observed correlations and measured causal effects suggest an influence of interventions related to the insertion or removal of Type II clones in training data for different programmatic constructs, including: identifiers, literals, types, white spaces, layout, and comments;
- We observed strong correlations between the number of layers and model performance in terms of Next Token Prediction (NTP), which might suggest a causal interpretation of code predictions. However, do_{code} revealed that the reported Average Treatment Effects were close to zero for such phenomena – demonstrating the presence of confounding bias suggesting multiple effects are at play;
- Finally, the intervention of masking random tokens has more impact on code predictions than masking grammar-based categories. This suggests that our BERT-like models do not entirely capture the structural information of programming languages.

Our contributions. Previous findings suggest the need for integrating causal interpretability in post-hoc analysis of neural language models. We hope to provide researchers with an approach for integrating causal analysis into their research. To that end, in addition to our case study showing the power such an analysis can provide, we have additionally created a checklist that summarizes the steps required to apply Causal Interpretability to Neural Code Models (see Fig. 15). In summary, this paper makes the following contributions:

1. A formal four-step pipeline for interpreting NCMs that enables causal analyses of different settings for language model use; **2.** A comprehensive study on seven practical scenarios rooted in software engineering practice in which do_{code} is applied to popular deep learning models for code generation; **3.** A set of findings for these models that help to explain their behavior in different settings challenges some current notions of code generation models; **4.** A checklist to help researchers apply Causal Interpretability to Neural Code Models; **5.** And a replication package [35] to both encourage the use of our technique in future work on developing and evaluating code generation models and for replicating our experiments.

2 Background & Related Work

Interpretable Machine Learning is a research field aimed at understanding how opaque models give rise to predictions. This process is centered around human understanding. Although the research community has not achieved a consensus on a precise definition of interpretability, researchers usually refer to this field either as "Interpretable Machine Learning" or "Explainable AI" [24], [25]. The main goal of the field is to create methods that explain models' reasoning and then verify whether such reasoning is sound [27]. These interpretability methods can be classified into different criteria depending on the researcher's concentration. The most common criteria are post hoc vs. intrinsic, model-specific vs. model-agnostic, global vs. local explanation scope, feature importance vs. rule-based unit of explanation, and causal levels [25], [36]. We depict in Fig. 2 a summary of the most relevant methods published thus far. Fig. 2 also positions our method do_{code} within the scope of *Causal Interpretability*, which is based on the concept of Ladder of Causation by Pearl introduced in Sec. 3 [37].

> *Causal Interpretability* is a post hoc global approach by which Neural Code Models (NCMs) are interpreted or explained from a causal assumption encoded in a Structural Causal Model (SCM). By using the formalism of Pearl's Ladder of Causation, researchers can *estimate* a quantifiable *causal effect* of proposed interventions.

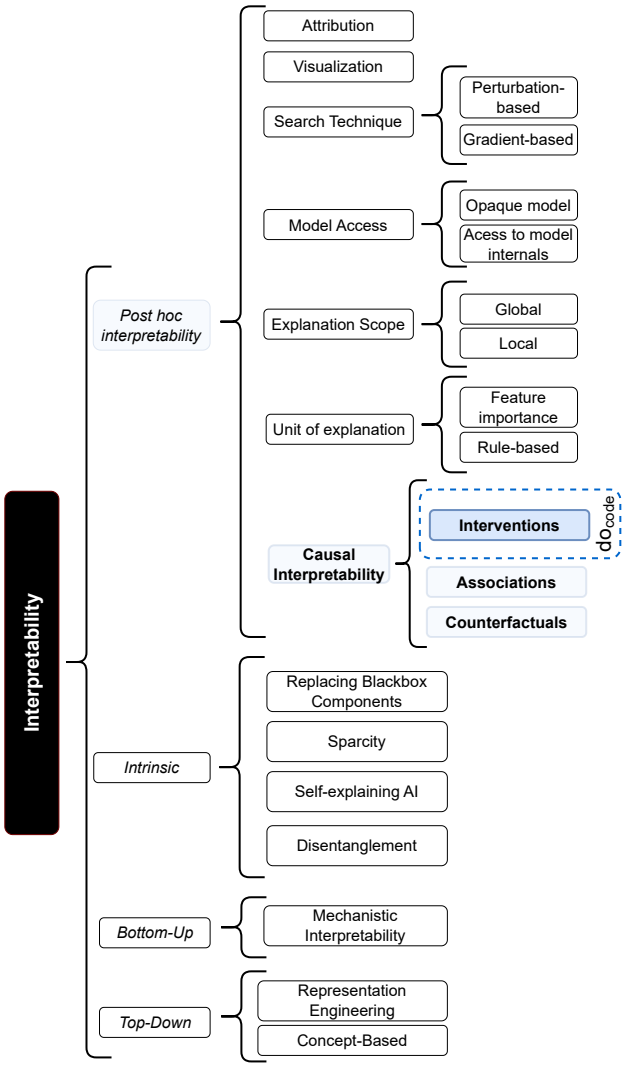


Fig. 2: A classification of key methods in interpretability including *Causal Interpretability*.

Applications of NCMs in SE: NCMs in SE have a rich history, stemming from the seminal work by Hindle

et al. who proposed the concept of the *naturalness* of software using *n*-gram models [38]. Then, with the rise of Deep Learning, researchers began exploring the possibility of adapting NLMs to code, as initially demonstrated by White et al. [2]. Since then, NCMs have been used for a number of SE tasks such as code completion [3], [15], [39], [40], [41], [42], [43], [44] and translation [4], [9], [45], [46], [47], [48], [49]. Researchers have also investigated various representations of NCMs for code [50] as well as graph-based representations based on ASTs [51] and program dependency graphs [52]. It has also been shown that using a NCM for code representation and then fine-tuning for specific SE tasks can achieve *state-of-the-art* performance across tasks [53], [54], [55].

Interpretability of NCMs: A growing body of research has investigated the potential of evaluating and understanding LMs using various techniques, which can be classified into two basic groups: (i) *intrinsic techniques* such as probing for specific linguistic characteristics [56] and examining neuron activations [57]; and (ii) *post hoc techniques* such as benchmarks [15], [34], [58], [59], [60], attention-based analysis [61], [62], and intervention analyses [17], [63], [64], [65]. Karpathy et al. were among the first to interpret NLMs for code using general Next Token Predictions [66] as an interpretability method as well as investigating individual neurons. Our work extends Karpathy et al.'s interpretability analysis using causal inference and a more statistically rigorous methodology that grounds its evaluation in SE-specific features and settings.

Complementary to our work, the work by Polyjuice paper [67] aims to automatically generate perturbations to language, to see how a given model performs in different scenarios. Our work is differentiated by the fact that we both define a mapping of source code tokens to categories and define a methodology for linking causal relationships of model changes to these various categories. As such, our work functions alongside techniques such as Polyjuice, providing deeper insights into why model performance varies across different perturbations. Our work is also complementary to the work by Cito et al., [68] on Counterfactual explanations for models of source code. This work takes a significant step toward defining perturbations of code for counterfactual explanations of generative models. Our technique in conjunction with Cito et al.'s approach can provide deeper explanations as to *why* a given NCM's performance changed for a given perturbation.

Our method is different from the work on adversarial robustness via one key point, similar to the work of Cito et al. *Our different testbeds are not intentionally designed to fool a model.* On the contrary, our different testbeds represent completely natural distributions of code tokens collected at scale. As such, our research has a completely different aim as compared to the work on adversarial robustness. Other related works from Explainable Artificial Intelligence (XAI) have focused on the usage of causal theory applied to generate post hoc explanations about the logical structure of a neural network [69] or explore the causal relationships between the explanation and the prediction [70]. Both causal XAI methods differ from our work since we employ the notion of Pearl's Ladder of Causation directly on the inputs and outputs of NCMs.

3 Why do we Need Causal Interpretability for Deep Learning Models applied to Software Engineering?

Our research leverages Pearl's theory of Causal Inference and grounds it in interpreting Neural Code Models (NCMs). Given the wealth of metrics and benchmarks that exist for NCMs it is natural to ask *why should we study causation in Deep Learning for Software Engineering?* Causation has two main goals in science (i) discovering causal variables and (ii) assessing counterfactual interventions [37]. The field of *Deep Learning for Software Engineering (DL4SE)* can take advantage of the latter when dealing with uncertainty and *confounding bias* of NCMs. Estimating counterfactual interventions is a powerful tool to generate explanations of the model's performance. Our method, do_{code}, can be applied to a wide range of SE models for detecting and eliminating confounding bias. We do not intend to pose do_{code} as a tool to enhance prediction performance but as an adaptable approach to analyze why NCMs obtain their predictions. However, quantifying the effects of interventions requires establishing a causal structure underlying the target data.

Randomized controlled experiments were the general choice to explore causality before Pearl's definitions of graphical models. Nonetheless, it is not practical to force developers to perform interventions (such as removing comments from a training corpus) or even train hundreds of NCMs to test various treatments. Instead the $do-operator$ and causal graphs (*i.e.,* Structural Causal Models) are better tools for performing causal estimations from observational data. Reconstructing such graphical representations is challenging since it not only requires formalizing causation in the field of SE (*i.e.,* defining potential outcomes, common causes, and treatments) but also tracing and connecting software data to causal models (see the pipeline in Sec. 4). In addition, formulating interventions is not an easy process. We must hypothesize feasible transformations or interventions that can occur in code to simulate real-world settings for NCMs, a concept that we synthesize as **The Causal Interpretability Hypothesis** (see Sec. 3.3). To that end, we have proposed a pipeline to help aid in adapting the process of causal inference to the interpretation of Neural Code Models (NCMs). This pipeline has been inspired by Pearl's notion of the *Ladder of Causation*, introduced below, and the *doWhy* library [71].

3.1 Pearl's Ladder of Causation

According to Pearl & Mackenzie [37], Causal Inference (CI) seeks answers to questions of **association** (*what is?*), counterfactual **interventions** (*what if?*), and pure **counterfactuals** (*why?*). The authors introduce the concept of *Ladder of Causation* to match distinct levels of cognitive ability with concrete actions: seeing (*level one*), doing (*level two*), and imagining (*level three*). Our proposed analysis is primarily concerned with levels one & two. Particularly, our method do_{code} is an extension of the intervention level.

Causal Association. In level one causation, $p(Y|T)$ is estimated by using typical correlation methods (*e.g.,* Pearson, Spearman, or Covariance) in addition to functional associations such as $y = g(t)$, which can be predicted with regressions or ML methods. For binary treatments

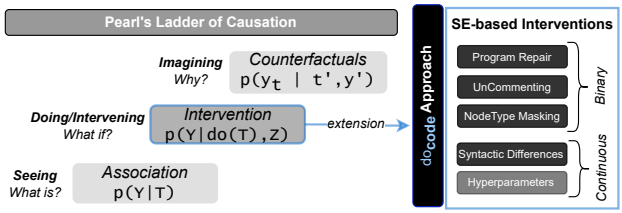


Fig. 3: Ladder of Causation: do_{code} is an extension of the intervention level.

similar to the ones we used in $T_{[data]}$ (*e.g.*, Buggy/Fixed and Commented/Uncommented), we opt to employ Pearson correlations ρ_{YT} and Jensen-Shannon distance as association estimand for level one causation.

Causal Intervention. If we want to go beyond *"what is"* type questions, we must move past simple correlations and associations. This requires the $do-operator$ found in level two, causation $p(y|do(t))$. It is relevant to identify cases of *spurious correlations* (*i.e., Confounding Bias*) or cases where $p(Y|T) \neq p(Y|do(T))$. Typically, *association is not causation* due to the influence of a common cause or confounding variable Z. Such a variable is the one that is being controlled for or adjusted employing the concept of *adjustment formula* in Eq. 1. Nonetheless, we can still compute the correlation ρ_{YZ} to assess the variables affecting potential outcomes.

An Example of Confounding Bias. Consider an intervention that simulates the software engineering task of *ProgramRepair* where a treatment T is predicting tokens following either Buggy/Fixed code, Y is the cross-entropy of each method of a dataset of both buggy and fixed code (which we introduce in detail later in the paper), and Z is the *Number of Subwords* for each method. Using do_{code} we find there exists a spurious correlation after estimating the association distribution $p(Y|T) \approx 0.67$ with *Jensen-Shannon Distance* (see Def. 6) and intervention distribution $p(Y|do(T)) \approx -2E-4$ with *Average Treatment Effect* (ATE) (see Def. 5). One possible explanation is that the common cause Z (the number of subwords) confounds the relationship between the treatment and the outcome. Fig. 4 depicts the influence of Z on the potential outcome Y for BuggyCode ($p(Y|Z, T = Buggy) \approx 0.87$) and FixedCode ($p(Y|Z, T = Fixed) \approx 0.86$). Blue and orange points in the plot are code snippets from the dataset. These points are equally distributed, which suggests that the *ProgramRepair* intervention has a negligible impact on model effectiveness as measured by cross-entropy.

3.2 Software Engineering-Based Interventions

In order to enable our analysis based upon Pearl's Ladder of Causation, we need to design *interventions*, or changes in the input data distribution that represent a meaningful concept (*i.e.*, commented vs. uncommented code), to attribute cause from the changes in the model's performance in predicting code tokens in different settings. That is, if we observe that a model is less effective at predicting operators after changing "test" data sequences to which the model is applied, we may be able to confirm that the change *caused* this drop in effectiveness if we properly control for confounders. We design do_{code}'s interventions based on the fact that NCMs are often not applied to the same types of code corpora upon

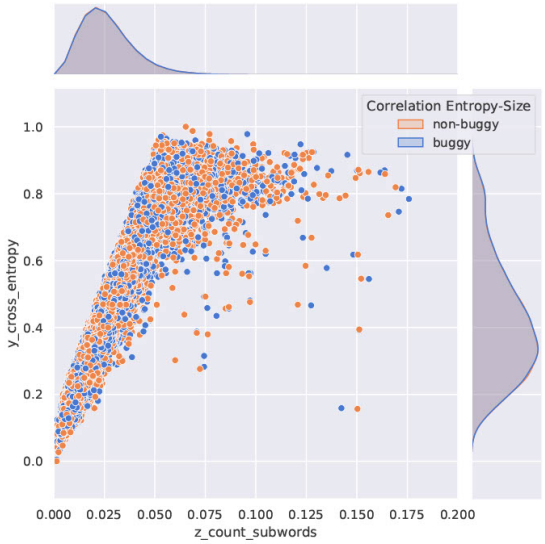


Fig. 4: *Spurious Correlation* between the *Number of Subwords* common cause and Cross-Entropy values ($p(Y|Z) \approx 0.87$) for the *ProgramRepair* intervention generated from GPT-$2_{6,12}$.

which they are trained. For instance, if a model trained on a well-commented dataset is applied to predict segments of poorly commented code, this could potentially impact performance. As such we define parallel code corpora which contain programming language-specific changes across the datasets, and specifically introduce four different initial interventions depicted on the right side of Fig. 3.

We define *SE-based Interventions* to better understand model performance across different settings. We formulate these settings as parallel code corpora with differing specified semantic properties. For instance, a testbed aimed at simulating a debugging environment may consist of two parallel corpora: the buggy code, and the (corresponding) fixed code. Therefore, these datasets describe some high-level SE properties, which we employ in do_{code}'s causal analysis. We define four types of SE application settings, adapted from both our prior work and community datasets: (i) buggy/non-buggy [72], (ii) commented/non-commented [60], (iii) type II, and (iv) type III clone pairs' differences [73], and Abstract Syntax Trees node masking [74]. do_{code} is extensible, meaning that researchers can define their own code corpora interventions. In addition, we have also defined *Model Hyper-parameter Interventions* to extend the causal analysis beyond data or code corpora perturbations (see Sec. 9).

3.3 The Causal Interpretability Hypothesis

The following hypothesis aims to answer the question of why causal interpretability is necessary for the field of Deep Learning for Software Engineering (DL4SE). We claim that the prediction generated by any NCM can be causally explained by employing the notion of the Ladder of Causation. Our proposed method do_{code} uses the process of causal inference to enable practitioners to answer their particular causal queries.

Hypothesis: do_{code} is a causal interpretability method that aims to make DL4SE systems (*i.e.,* Neural Code Models) and their decision-making process understandable for researchers and practitioners.

4 An Overview of the do_{code} Approach

do_{code} comprises a set of statistical and causal inference methods to *generate (and evaluate) post hoc interpretations* of NCMs. In the simplest of terms, our proposed interpretability approach asks *Why does a NCM make a given code prediction?* – and provides a framework for answering this broad question by ① **modeling** the inference problem to encode causal assumptions, which rely on *domain knowledge* of NCMs and *observable software data*, in a graph representation; ② **identifying** the causal estimand based on the previous graph representation; ③ **estimating** the causal effect based on probabilistic and machine learning methods that operate on observable software data; and ④ **validating** the causal process by, firstly, refuting obtained estimate using different sensitivity and robustness techniques (*e.g.,* placebo, random common cause, data subsets validation); and secondly, vetting the creation of the graph using correlational analyses on SCMs' variables. Fig. 5 depicts the four steps that consolidate our post hoc interpretability approach (*i.e.,* after models have been trained). We provide a high-level explanation of how each step of do_{code} pipeline functions (modeling: Sec. 5, identifying: Sec. 6, estimating: Sec. 7, and validating: Sec. 8) before describing the case study design in detail in Sec. 9.

In addition to the previous pipeline and to help bridge the gap between a given NCMs low-level code prediction and human-understandable categories, we propose a **syntax clustering** criterion that aims to group well-known categories of programming languages from individual code tokens. This criterion is not explicitly defined in the pipeline but it is required for posterior evaluations. Syntax clustering is introduced in Sec. 9 and validated in Sec. 11.3.

While our syntax clustering provides the building blocks for explaining global model behavior when applied to predict different types of software data, it may be difficult to determine whether any correlations in performance are *causal*, *i.e.,* definitely resulting from the intervention, or *spurious*, *i.e.,* potentially caused by confounding factors. To enable an analysis of whether a change in model performance is truly causal or not, allowing for the generation of *accurate explanations*, we propose an analysis based on causal inference which we describe below.

Preliminaries. *Software Data*, *Domain Knowledge*, and *Exploratory Causal Analysis* are indispensable elements before starting the do_{code} pipeline. We refer to *Software Data* as any artifact generated by a software engineering-based process (*e.g.,* source code, requirements, documentation). Because we concentrated on Neural Code Models, our target data is human-generated code for causal interpretability. *Domain knowledge* enables software researchers to discern whether the causal assumptions about data are plausible on scientific grounds. This ability to qualitatively encode causal assumptions in a graphical model is known as *Transparency*, depicting the process in which researchers recognize cause-and-effect relationships in a domain [75]. Although *Software Data* and *Domain Knowledge* are sufficient elements to create a graphical form of perceived causal assumptions, we scrutinized the graph creation process using statistical tools. These tools aim at detecting correlational evidence among SCMs' variables to support the qualitative graph encoding. We present the methodology and results of these correlations in *Exploratory Causal Analyses* (see Sec. 11).

Step One: Modeling Causal Inference Problem. A variable T is a *cause* of Y if the variable Y depends on T to determine its value. However, there are also variables U that represent *unmodeled* factors [28]. This causal relationship can be precisely formulated via a causal model that uses directed acyclic graphs (DAGs) to describe direct parent-child relationships, instead of probabilistic dependencies, among random variables such as T and Y. It also allows us to introduce a graphical definition of causation [28], [76]. Therefore, Structural Causal Models (SCMs) are graphical models responsible for enabling us to quantitatively estimate the results of an action or intervention simulated in the graph. In other words, SCMs provide a framework for *counterfactual reasoning*.

Step Two: Identifying Causal Estimand. In this step, do_{code} formulates a causal estimand, (*i.e.,* mathematical expression) subject to the previously defined Structural Causal Model (SCM). do_{code} employs graph-based criteria and *the adjustment formula* for identifying an expected causal effect. These graph-based criteria determine which set of confounders Z in the SCM should be conditioned on when seeking the causal relationship between T and Y. The doWhy library is used to *explore* the identification of different causal estimands with criteria such as back-door criterion, front-door criterion, instrumental variables, and mediation analysis [71].

Step Three: Estimating Causal Effects. After obtaining an expression using any identification criteria, our method uses a suitable or proper estimation method to compute the causal effect. The estimation method depends on the nature of the SCM's variables (*e.g.,* binary, discreet, continuous). The library doWhy supports methods based on estimating the treatment assignment, the outcome model, the instrumental variable equation, and machine learning estimators [71]. Our study concentrates on estimating *propensity scores* for binary treatments and *linear regression* for continuous treatments.

Step Four: Validating Causal Process. An important step in causal analysis is validating our causal process. This validation comprises two parts: *refuting effect estimate* and *vetting the causal graph*. To perform the first part, we employ *refutation methods* that calculate the robustness of the causal estimate by making the assumptions *falsifiable*. These methods test the robustness of our assumptions through various types of perturbations on the graph or software data to see the impact on our Average Treatment Effect (ATE) estimation. There are multiple refutation methods, but in this paper, we focus on four: adding a random common cause or covariate $\mathcal{R}_1$, adding an *unobserved* common cause or covariate $\mathcal{R}_2$, replacing the treatment with a random variable or placebo $\mathcal{R}_3$, and removing a random subset of the data $\mathcal{R}_4$. Depending on the method, the resulting value should either be the same as the original ATE (*i.e., invariant transformation*) or close to zero (*i.e., nullifying transformation*). To perform the second part, we inspect the causal graph

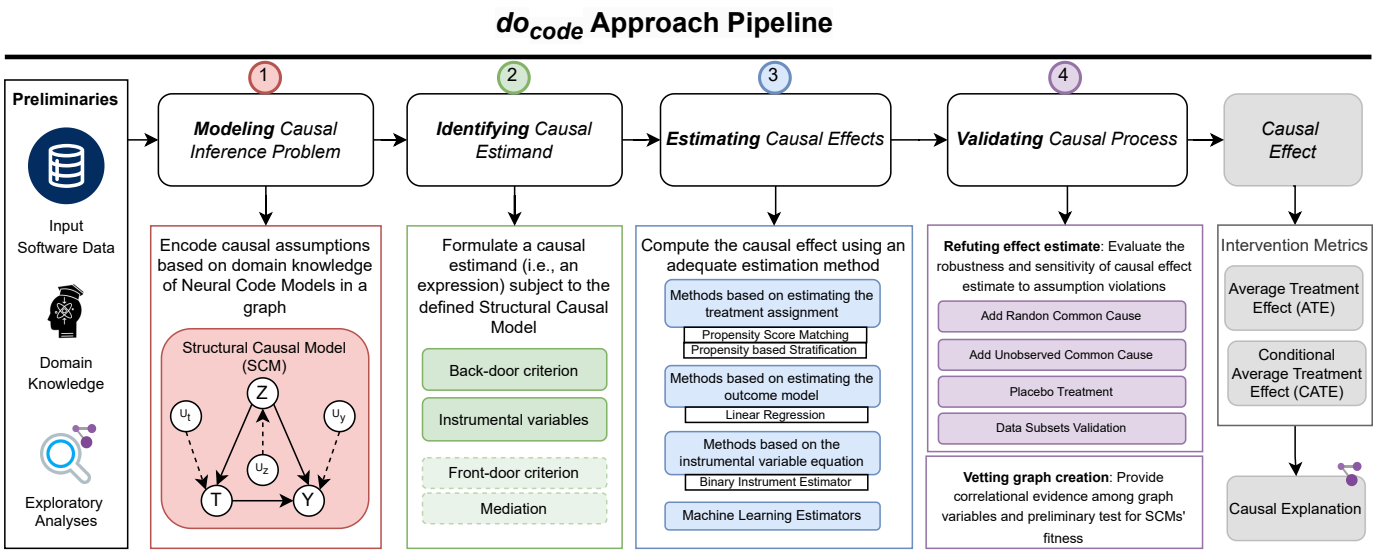


Fig. 5: Overview of the do_{code} Approach: Each numeral represents a step in the process of generating causal interpretations. First, a Structural Causal Model (SCM) that frames the explanation hypothesis is formulated. Second, do_{code} executes graph surgery on the SCM to isolate a targeted estimand of the causal effect. Third, the causal effect is assessed based on the targeted estimand. Finally, the estimated effect and the SCM are scrutinized through refutation techniques and exploratory analysis to confirm their validity.

at two different times: 1) *graph creation*, which indicates supporting the rationale behind the construction of the graph using correlational analyses on SCMs' variables, and 2) *graph correctness*, which applies the concept of *Testability* after executing an identification method [37].

Enabling Causal Interpretability for NCMs. Consider the scenario in which we want to understand how a given model operates between buggy and fixed code. To do this do_{code} constructs a SCM composed of an intervention variable $T = Buggy$ and potential outcome $Y = Acc$ (representing a measure of model effectiveness or performance in terms of our defined code categories). In addition, we identify confounding factors Z for interventions and potential outcomes. Next, we want to perform the intervention wherein the model is applied to fixed code instead of buggy code. do_{code} then constructs a parallel graph for this intervention. Using notions from causal inference theory (*i.e.,* modeling, identifying, estimating, refuting), do_{code} is then able to determine how the intervention affected model performance in terms of code categories, and whether the change in performance is a true causal effect, or spurious, resulting from effects of covariates.

While the above scenario illustrates the intuition of our causal analysis, it is important to define how causal effects, and ultimately interpretations, will be generated. First, we need methods by which we can measure model performance. To do this we use Cross-Entropy loss, or the difference in distribution of the model's prediction and the ground truth for given token sequences, and average Next Token Predictions, or probabilities assigned by models to individual tokens, across token sequences as these measures are relevant values produced at inference time that reflect the prediction effectiveness. By relating these values to SE-based interventions T, we can gain an understanding of how well a studied model is *generating code* under these treatments. To calculate causal effects for binary treatments similar to the ones we used in $T_{[data]}$ (*e.g.,* Buggy/Fixed and Commented/Uncommented), we first calculate association correlations (Def. 6) and then causal effects (Def. 5).

If we want to estimate the extent buggy/non-buggy SE-based interventions affect the accuracy of a NCM, we need before to *identify* the effect of the treatment using the *adjustment formula* of causal inference (see Def. 4). After the causal effect is calculated, we aim to generate **causal explanations** from our analysis via structured templates that relate the code token category to the intervention: *e.g., `[code token category]` performed worse by `[change in performance]`, due to a change in model application from `[intervention]` to `[intervention]`, with a causal analysis Average Treatment Effect of `[ATE value]`.*

5 Step One: Modeling Causal Problem

In the first pipeline step, assumptions about relationships among data are defined in a Structural Causal Model (SCM) similar to Fig. 6. This SCM is later modified to compute the interventional distribution $p(Y|do(T))$ in step two. Causal assumptions must be made explicit in the graph, which entails defining the interventions T (*i.e.,* binary: buggy/non-buggy, discrete: layer modifications, continuous: syntactic differences), potential outcomes Y (*i.e.,* Cross-Entropy or NTPs performance), and the confounders (or common causes) Z. For all SCMs proposed in this study, we assumed that confounders are SE quality metrics since they have the potential to influence models' code predictions, *i.e.,* a NCM may be influenced by more or less for loops, as well as influence interventions, *i.e.,* more code is correlated with more bugs [77].

Before interpreting the prediction performance of a given Neural Code Model (NCM) using causal explanations, we introduce the formalism of a NCM as defined below.

Definition 1. **Neural Code Model (NCM)**: A NCM is a probability distribution $P(w_t|w_1, w_2, \ldots, w_{t-1})$ in which the output w_t at time step t, given the input sequence $(w_1, w_2, .., w_{t-1})$, is inferred through the conditional probability $P(w_t|h_t)$ [2], [40]. The hidden state h_t encapsulates the properties of the preceding context. The training corpora of NCMs primarily contain source code. Consequently, input sequences are split into subwords of code (*i.e.,* tokens w from a vocabulary $\mathcal{V}$). Furthermore, NCMs are typically pre-trained for code generation, and fine-tuned to perform specific SE downstream tasks such as code summarization, test case generation, and bug fixing.

The following example showcases a practical scenario in which code predictions can be causally explained by SE-based interventions on program repair.

Example 1. Consider a Bayesian network created to explain code predictions of a given NCM, depicted as an arrow that links buggy input code to the model's accuracy $Buggy \rightarrow Acc$. Accuracy is a measure of agreement between current vs. expected predictions. We can use accuracy as a measure that contains and represents information about code predictions. The variables $Buggy$ and Acc are dependent. These variables embody a *program repair intervention*. Therefore, if we want to define the joint distribution $p(Buggy, Acc)$ to represent the network, we must specify by Bayes' rule using the prior $p(Buggy)$ and the conditional probability $p(Acc|Buggy)$. However, this joint distribution can be computed in the opposite direction $Acc \rightarrow Buggy$ using the same Bayes' rule for a prior $p(Acc)$ and conditional $p(Buggy|Acc)$. The fact that the joint distribution can be represented with both networks is non-intuitive since we know from experience that the model performance cannot give rise to a bug in the original input. In other words, the relationship between these variables is asymmetric or *causal*. Hence, we expect that buggy snippets affect the prediction performance of NCMs, not the other way around.

A first attempt to address the relationship in Ex. 1 would be computing a correlation coefficient $\rho_{TY} \approx p(Y|T) = p(Acc|Buggy)$, where T is a binary *intervention* that represents the *debugging* process and Y is a *potential outcome* that corresponds to the prediction performance of the model. This coefficient, however, is still symmetric: if T is correlated with Y, then Y is equally correlated with T. *Causal networks* allow us to model causal asymmetries where directionality goes beyond probabilistic dependence. These causal models represent the mechanism by which data *were generated* [29]. Instead of testing whether $Buggy$ and Acc are conditionally dependent, *causation* asks which variable *responds* to the other one: $Buggy$ to Acc or Acc to $Buggy$? [29]. Therefore, we can formally introduce a definition of causation:

Definition 2. **Causation.** A variable T is a *cause* of Y if the variable Y depends on T to determine its value. Formally, the value of Y was *assigned* based on what is known about T. In other words, the value of Y is determined by a ***structural equation*** $Y = f_y(T, U_y)$ and the arrow $T \rightarrow Y$. The U variables in these equations represent *unmodeled* variables that are exogenous to the causal network but disturb the functional relationship between the outcome and its treatment [28].

Similarly, we can define a structural function for the treatment $T = f_t(U_t)$ that depends only on U disturbances assuming that no *common causes* exists between potential outcomes and interventions. A common cause (or confounder) is a random variable Z that causally influences two variables that are initially perceived as statically dependent ($T \not\perp\!\!\!\perp Y$). However, this dependency can be explained by the underlying influence of Z on the effects, making the effects conditionally independent ($T \perp\!\!\!\perp Y|Z$). Therefore, there exist more complex causal relationships between treatments and outcomes that we can model with structural equations that correspond to a ***Structural Causal Model*** (SCM). We introduce a definition for Structural Causal Models as graphical models that capture causal assumptions:

Definition 3. **Structural Causal Models.** These directed acyclic graphs (DAGs) describe direct parent-child relationships, instead of probabilistic dependencies, among random variables X_i. The value x_i of each variable X_i is defined by the structural equations $x_i = f_i(PA_i, U_i)$ where $PA_i = X_j : X_j \rightarrow X_i$ denotes the set of parents or direct causes of X_i. This model allows us to introduce a graphical definition of causation [28], [76].

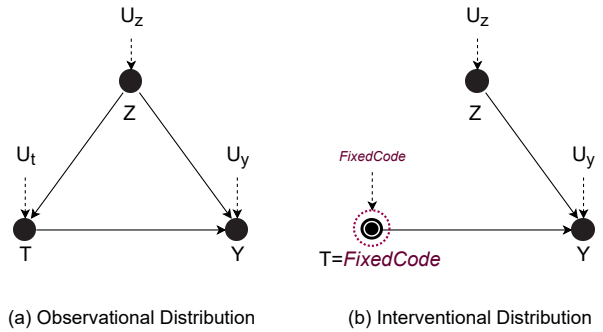


Fig. 6: (a) *Structural Causal Model* representing cause-effect relationships of program repair in NCMs. (b) The *SCM* after intervening the treatment with *Fixed Code*.

In summary, this step consists of *setting down assumptions* about the causal relationships of software data employed to interpret NCMs. SCMs help us to describe relevant features of software data and how these features interact with each other. In the following subsection, we define each component of the Structural Causal Models (*i.e.,* interventions, potential outcomes, and common causes or confounders).

5.1 SE-Based Counterfactual Interventions

NCMs are notorious for behaving differently across distinct datasets [78]. For example, if a model trained on a well-commented dataset is applied to predict segments of poorly commented code, this mismatch could potentially impact its performance. As such, we assert that observing model performance across datasets with different characteristics can aid in understandability and interpretability. Hence, we defined *SE-Based Counterfactual Interventions* T to better understand model performance across different settings. We

formulate these interventions based on domain knowledge from observable testbeds (*i.e.,* datasets) organized in sample pairs **treatment** $T = 0$ (*i.e.,* BuggyCode) and **control** $T = 1$ (*i.e.,* FixedCode). Note that we define testbeds according to different applications often described in SE research. The general process comprises of identification of some specific *intervention* (*i.e.,* program repair) and 2) construction or collection of the necessary data via mining repositories or other means that contain these observed interventions. More complex interventions will likely be more challenging to prepare.

In do_{code}, *counterfactual interventions* produce explanations motivated by both semantic perturbations $T_{[data]}$ to our SE Application Settings (*e.g.,* Program Repair "Buggy/-Fixed Code", (Un)commented Code, Syntactic Differences) and model hyper-parameter variations $T_{[hyp]}$ on NCMs (*e.g.,* layers, units, or heads). Although hyperparameter variations are NOT data perturbations based on SE settings, we include them to extend the analysis of possible causes of models' predictions beyond data interventions.

5.2 Potential Outcomes / Code Predictions

Cross-Entropy (Fig. 7-③), Next Token Prediction (Fig. 7-②), and Distant Metrics (*i.e.,* Jaccard, Levenshtein, and Sorence-Dicen) are relevant values produced at inference time that reflect the effectiveness of a model at *predicting code*. By relating these values to counterfactual interventions T (*i.e.,* program repair), we can gain an understanding of how well a studied NCM is *generating code* under these SE-based interventions.

Cross-Entropy. We refer to Cross-Entropy loss as a measure of a model's *coarse-grained performance* $Y_g : w \rightarrow -\sum_{t \in |w|} P(w_t|d_t) \log Q(w_t|w_{<t})$ as these losses capture the overall performance of a NCM over an entire sequence of tokens w. Due to the discrete nature of the data, the expression $P(w_t|w_{t-1:1})$ can be estimated using a classifier. The classifier, in our particular case, is a NCM [79]. Hence, rather than using n-grams or Markov Models to approximate $P(w_t|w_{t-1:1})$ [44], it is convenient to use a latent model $P(w_t|w_{t-1:1}) \approx P(w_t|d_t)$, where d_t is known as a *hidden state* that embeds the sequence information from past observations up to the time step t. Depending on *how* the sequence is processed, the hidden state d_t can be computed using either an autoregressive network (*i.e.,* such as a Transformer (GPT) [33]) or a Recurrent Neural Network (RNN).

Next Token Prediction (NTP). Conversely, NTP values signal *fine-grained performance* $Y_l : w_{<t} \rightarrow P(w_t|w_{t-1:1})$ within token-level contexts. NTPs capture local predictions for individual tokens that are affected by complex interactions in NCMs and are equivalent to the estimated predicted value (or softmax probability) $\sigma(k)_t$ for each token. Bear in mind that the size of the vector $\sigma(k)_t$ is the vocabulary $|\mathcal{V}|$, in which k represents the non-normalized log probabilities for each output token t. NTPs capture the value of the expected token w_t instead of the maximum value estimated in the vector $\sigma(k)_t$.

Distance Predictions. Similarity distance scores play a crucial role in assessing the model's *distance performance*, defined by the expression $Y_d : s' \rightarrow \nabla(s, s')$. Function ∇ represents the similarity coefficient used for the pairwise comparison of two finite sample sets, s and s'. For instance, if set A denotes Node Types in the AST (Abstract Syntax Tree) of a ground-truth code sample, and set B represents Node Types in the AST of a predicted code sample, we can quantify their similarity using metrics such as the Jaccard Index, Levenshtein distance, and Sorensen-Dice coefficient. The resulting similarity scores, which range from 0 to 1, effectively measure the degree of similarity between the sets, with 0 indicating no similarity and 1 meaning that the sets are identical.

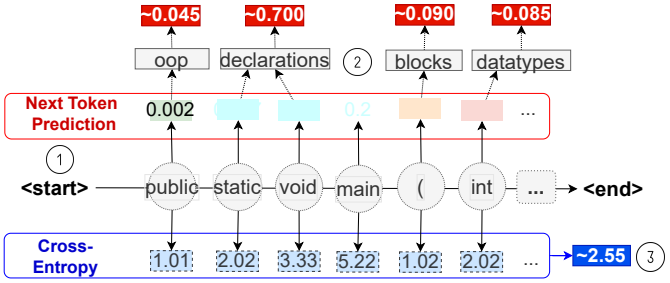


Fig. 7: Potential Outcomes are Code Prediction of NCMs: Cross-Entropy (Y_g), Next Token Predictions (Y_l), or Distance metrics (Y_d).

5.3 Common Causes or SE-based Confounders

In past work, several factors such as code duplication [80] have been illustrated to affect model predictions. As such, we derive a list of potential SE-based confounders that could influence a model's prediction of our clustered syntax categories beyond our interventions defined in the previous section. Our initial set of confounders (also called common causes) include McCabe's complexity, LoC (Lines of code), number of:

- `returns`
- `loops`
- `comparisons`
- `try/catches`
- `parenthesized expressions`
- `string literals`
- `variables`
- `max nested blocks`
- `anonymous classes`
- `inner classes`
- `lambda expressions`
- `unique words`
- `log statements`
- `modifiers`

Practitioners and researchers can extend the search of potential SE-based confounders based on their domain knowledge, empirical analysis on NCMs, or observations of NCMs' behavior in production.

6 Step Two: Identifying Causal Estimand

In the second pipeline step, once the SCM (similar to Fig. 6) is constructed, do_{code} must validate under which conditions the structure of the created graph is sufficient for estimating a causal effect from software data. Graph-based identification criteria are *test* mechanisms or tools employed in Causal Inference to determine whether a causal effect can be estimated for any two variables T and Y. do_{code} supports graph-based criteria such as back-door criterion, front-door

criterion, instrumental variables, and mediation. The application of these criteria depends on the configuration and shape of the SCM. To formulate a correct estimand, do_{code} requires both an *adjustment formula* (see Eq. 2) and a *graph-based criterion*.

While we discuss identification criteria such as front-door and mediation as potential options for graph-based identification criteria, for our case study -in particular- we use the combination of the *back-door criterion*, *instrumental variables*, and the *adjustment formula*, as these cover sufficient interpretability scenarios for a causal graph with the standard shape. The standard shape of proposed SCMs includes data-based $T_{[data]}$ and parameter-based $T_{[hyp]}$ interventions, SE-based confounders $Z \in SE_{metrics}$, potential outcomes Y_l, Y_g, Y_d, and (if applicable) instruments I.

The back-door criterion identifies sets of confounders that should be adjusted/controlled for when we estimate causal effects from software data, while the instrumental criterion comprises variables with a direct effect only on the treatments. The mathematical details of how back-door or any other identification criteria operate are out of the scope of this paper. Detailed information about graph-based criteria can be found in academic manuals [28], [29], [76]. Nonetheless, the *adjustment formula* for computing causal effects by simulating interventions (*i.e.,* do-calculus [37]) is discussed in this section.

Structural Causal Models (SCMs) are stable mechanisms that remain invariant to local changes unlike probabilities computed by Bayesian networks [29]. This characteristic helps us to estimate quantitatively the results of an intervention in the graph without actually performing it in controlled settings (*i.e.,* randomized experiments). The mathematical tool employed to perform these interventions is the $do(\cdot) - operator$ [76]. For instance, if we want to estimate how fixed code affects the performance of a NCM, we need to compute the action $p(Acc|do(Buggy = False))$ (Eq. 1a). These actions are *interventional* distributions since we **set** the value of $Buggy$ to $False$. Note that this interventional distribution is **not** the same as the distribution $p(Acc|Buggy = False)$ (Eq. 1b). This latter distribution is *observational* as we are *conditioning* the performance on the value of the $Buggy$ variable. Intervening on a variable in a SCM means fixing its value and, therefore, changing the value of other variables of the network as a result. Conversely, conditioning on a variable means narrowing the cases that the outcome takes once we assign a value to the intervention.

Example 2. Consider Fig. 6 a generalization of the buggy influence on a deep model's performance. The first graph is a SCM composed of an intervention variable $T = Buggy$ and potential outcome $Y = Acc$. In addition, we identified some common causes $Z = SE_{Metrics}$ for interventions and potential outcomes. These common causes (or confounders) can be mapped to Software Engineering quality metrics (*e.g.,* Lines of Code, McCabe Complexity, Size of Methods). We want to perform the intervention $do(Buggy = False)$, which is the same as $do(T = FixedCode)$. The second graph depicts this program's repair intervention.

Note that fixing the value of T makes the SCM change by eliminating the effect or influence arrow of the confounder Z to the intervention. The disturbance U_t is also eliminated. This elimination process of input arrows to fixed variables is formally known as *graph surgery*. Both $do(\cdot) - operator$ and graph surgery allow us to untangle causal relationships from mere correlations [76]. The law of total probabilities and invariance principle are required to compute the observational and interventional distributions [28]:

$$p(Y|do(t = FixedCode)) = \sum_{z \in_{metrics}} p(Y|t, z)p(z) \quad (1a)$$

$$p(Y|t = FixedCode) = \sum_{z \in_{metrics}} p(Y|t, z)p(z|t) \quad (1b)$$

Note that Eq. 1a differs from Eq. 1b: the prior $p(z)$ in contrast to $p(z|t)$, which is precisely the link that is eliminated in the SCM. Eq. 1a is formally known as the ***adjustment formula***. This formula is one of the building blocks in causal inference since it helps us to adjust common causes or control for confounders to allow the estimation of *causal/treatment effects* [29], [76].

Definition 4. **Treatment Effects.** Given a Structural Causal Model where a set of variables PA denotes the parents of T, the treatment effect of T on the potential outcome Y is given by

$$p(Y = y|do(T = t)) = \quad (2a)$$

$$\Sigma_z p(Y = y|T = t, PA = z)p(PA = z) = \quad (2b)$$

$$\Sigma_z p(T = t, Y = y, PA = z)/p(T = t|PA = z) \quad (2c)$$

In our initial causal statement, we generally accept that buggy code *causes* a NCM to predict poorly. Although the causal statement is true, it is not guaranteed that "every buggy snippet" is certain to make a model predict poorly. Therefore, causal relationships are **uncertain**. This uncertainty is captured by employing conditional probabilities described in Eq. 2b. Note Eq. 2b is a generalization of the adjustment formula Eq. 1a. We connect observational data with our interventional distribution to compute treatment effects. A standard way of connecting data with the interventional distribution is by employing a summation described in Eq. 2c. Note Eq. 2c is obtained with the application of Bayes' rule and algebraic manipulation once we multiply and divide Eq. 2b by the term $p(T = t|PA = z)$. This term is a conditional probability known as the *propensity score*. This propensity score and the joint probability of all the nodes are distributions that can be *obtained directly from data* [29]. We explain this connection for code generation in Sec. 7.

7 Step Three: Estimating Causal Effects

In the third pipeline step, do_{code} estimates the causal effect using statistical and ML methods based on the adjustment formula from the previous step. do_{code} computes *Propensity Score Matching* for binary SE interventions (*i.e.,* Buggy/-Fixed) and *Linear Regressions* for SE discrete interventions (*e.g.,* layers, units, or heads). We refer interested readers to the *doWhy* documentation for full estimation methods details [71]. For completeness, we will solely show how to

estimate a causal effect assuming a binary intervention as an example. We can start by explaining the notion of *Average Treatment Effect (ATE)*. ATE is simply the average score of all treatment effects (see Def. 4) computed for a population. In our case, an individual of the population is just a code snippet x.

Definition 5. **Average Treatment Effect (ATE)** Defining the first intervention as $do(T = 1)$ and the second by $do(T = 0)$, the Average Treatment Effect is the population average of the difference of causal effects of each code snippet x.

$$ATE = \mathbb{E}_{x\sim p(x)}[Y = y|x, do(T = t)] = \tag{3a}$$

$$\mathbb{E}_{x\sim p(x)}[\mathbb{E}[Y|x, do(T = 1)] - \mathbb{E}[Y|x, do(T = 0)]] = \tag{3b}$$

$$\mathbb{E}_{x\sim p(x)}[\mathbb{E}[Y^1|x, T = 1] - \mathbb{E}[Y^0|x, T = 0]] = \tag{3c}$$

The previous Eq. 3 shows the formal definition of an ATE. We can derive the final expression by applying the law of total expectations and the ignorability assumption $Y \perp\!\!\!\perp Z|T$, where the potential outcomes Y are independent of intervention assignments conditioned on covariates Z [76]. That is, the effects of the hidden confounders Z and missing data are ignored. In Eq. 3, the term $\mathbb{E}[Y^1|x, T = 1]$ represents the expected value of a potential outcome under an observable intervention (*i.e.,* FixedCode). Similarly, the term $\mathbb{E}[Y^0|x, T = 0]$ represents an expected value of a potential outcome under an observable intervention (*i.e.,* BuggyCode). Both terms are quantities that can be *estimated from data*. Covariate adjustment (in Eq. 1a), propensity score (in Eq. 2c), and linear regression are some of the estimation methods that we employ to approximate $ATEs$. Their usage depends upon the type of the intervention variable (*i.e.,* binary, discrete, or continuous) and causal graph assumptions.

8 Step Four: Validating Causal Process

Assumptions encoded in causal graphs are justified by observations of a *data generating process*. Therefore, testing for the quality of the causal graph fitting the data would be the main validity issue in the fourth pipeline step. We ask the question *How can we assess the validity of the underlying causal process?* To answer the question, firstly, we must assess whether the estimated causal effect from step three is not significantly altered after assumption violations (*i.e.,* refutation methods). Secondly, we must conduct exploratory causal analyses to scrutinize how strong the correlations are among SCMs' variables. These exploratory analyses are a byproduct of graph validity to provide more evidence that supports causal assumptions on the top of the *testability* with refutations and identification criteria.

8.1 Refuting Effect Estimate

The obtained causal effect can be validated using *refutation methods* that calculate the robustness and sensitivity of the estimate. Refutation methods came from the idea that hypotheses respecting a *data generating process* can be tested with conditions in which the hypotheses would be false. Structural Causal Models (SCMs) encode assumptions or hypotheses regarding a *data generating process*. Consequently, SCMs are *falsifiable* by modifying the model or the data to disprove initial assumptions.

In essence, the refutation methods apply random perturbations to the original Structural Causal Model (SCM) to test for the robustness of the estimated causal effect (ATE). Refutation methods inform researchers and analysts about the validity of the graph by applying *invariant* or *nullifying* transformations. *Invariant* transformations modify the data so that the estimated value should not change; otherwise, the causal model would fail the test. For instance, a refutation method tests a causal graph for confounders when this method affects, adds, or perturbs common causes. As such, if the obtained refutation value for common causes is not robust (*i.e.,* different from the original ATE), it suggests that we should update our initial assumptions about confounders encoded in the graph. On the other hand, *nullifying* transformations modify the data so that the estimated effect should be zero; otherwise, the causal model would fail the test too.

Although researchers can design their tailored methods to test distinct parts of the causal graph (*e.g.,* treatments, confounders, hidden causes, datasets), we chose four baseline methods for our study:

Adding a *random* common cause or covariate $\mathcal{R}_1$. We evaluate if the estimation method changes its estimate after introducing an independent random variable as a confounder from the dataset. This refutation involves computing new causal estimates by simulating a common cause with randomly generated data and reporting the average value. A stable causal effect should remain unaffected by this change, maintaining its original estimate $p(Y|do(T)) = p(Y|do(T), H)$. Fig. 8 depicts the modification.

Adding an *unobserved* common cause or covariate $\mathcal{R}_2$. A new dataset is artificially created for a correlated common cause between treatment T and outcome Y. Next, the effect of the unobserved common cause is recomputed. Considering that the backdoor criterion assumes that all common causes are observed, the new estimate should not be altered drastically after violating the criterion assumption.

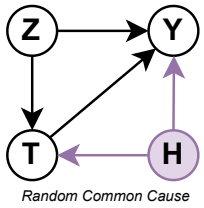


Fig. 8: SCM transformation by adding a random common cause.

Replacing the treatment with a random variable or *placebo* $\mathcal{R}_3$. We evaluate to what extent the estimated causal effect changes if we replace the true treatment variable T with an independent random variable of the same nature (*i.e.,* placebo). Specifically, we simulate datasets where the treatment is the placebo, hence the computed causal effects should have a distribution close to zero. By using this refutation method, we seek to test the reliability of the estimator.

Removing a *random subset* of the data $\mathcal{R}_4$. We evaluate if the estimation method changes its estimate significantly after replacing the given dataset with a randomly selected subset. Through multiple simulations, we calculate the causal estimate using subsets of the original data and report

the mean value. The initial causal estimate should remain unaffected by this modification.

In summary, for high robustness, we anticipate that $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_4$ will closely approximate the estimated ATE (Eq. 3). Conversely, the placebo $\mathcal{R}_3$ is expected to be zero.

8.2 Vetting Causal Graph

Beyond falsifying the structure and data of the causal graph with the refutation methods, we can also vet the validity of the graph at two different times:

Graph Creation Validity. Before or after the causal structure is modeled from *domain knowledge*, we can conduct exploratory data analyses to alleviate the absence of statistical information about the variables under study. While not explicitly required from Causal Inference theory, we measure the correlation between the confounders and the treatments in one analysis $\textcircled{Z} \rightarrow \textcircled{T}$. Then, we measure the correlation between the confounders and the outcomes in a separate analysis $\textcircled{Z} \rightarrow \textcircled{Y}$. The purpose of these correlations is to support domain knowledge observations quantitatively.

Graph Correctness Validity. The correctness of the graph is related to the concept of *testability* [37] in which we test whether a Structural Causal Graph (SCM) could have generated a dataset. Identification criteria such as back-door or front-door are standard ways of evaluating the fitness of a model. In fact, the back-door criterion uses a graphical mechanism known as *d-separation*, which facilitates the demonstration that causal models have testable implications in the data they generate [75].

9 Case Study Design

In this section, we aim to illustrate how do_{code} is applied to *enable causal interpretations* based on different interpretability scenarios. Fig. 9 depicts an overview of seven scenarios (or cases from A to G) and six essential criteria that comprise the case study. Each scenario can be unfolded into the following criteria: (i) the goal of do_{code} (*i.e.,* generating interpretations or validating an interpretability technique), (ii) the setup (*i.e.,* the deep learning architecture under analysis and the evaluation dataset or testbed) (iii) the definition of the Structural Causal Model (SCM) from domain knowledge (*i.e.,* intervention modality, hyperparameter interventions, data interventions, and potential outcomes), (iv) a syntax clustering strategy for grouping potential outcomes based on token predictions, (v) the usage of causal inference measure whether the value is an association (*e.g.,* Pearson, Jensen Shannon) or an intervention (*e.g.,* ATE, CATE), and (vi) the refutation testing method employed to validate the robustness of the interpretations (*e.g.,* placebo, unobserved common cause).

Note that practitioners and researchers should not necessarily have to stick to our seven cases, the configuration criteria can be extended to assess unexplored interpretability scenarios. Various permutations of the criteria outlined in Fig. 9 can be formulated depending on the research goal of the causal analysis. The subsections below detail the criteria that we propose for this study.

9.1 do_{code} Goal

The goal of do_{code} is to enable NCMs' interpretations of code predictions by estimating the causal effects of specific SE-based interventions. On top of that, do_{code} also facilitates causal inference for *validating* interpretability methods. Users of do_{code} should define their goals in alignment with this premise as these goals determine how the SCM is formulated. For example, do_{code} can be employed to interpret the effects of code smells on code predictions produced by given NCM, the SE-based intervention would be a binary treatment of samples with and without smells. On the other hand, we decided not to introduce a scenario in which do_{code} validates an interpretability method as it is out of the scope of this study. However, we offer sufficient guidance on assembling the criteria for this purpose in this section.

9.2 Setup

Applying do_{code} requires selecting both the deep learning architecture (*i.e.,* NCM from Def. 1) and the evaluation dataset or testbed for our interpretability scenario.

About Deep Learning Architectures. Because causal interpretability is agnostic to NCM, do_{code} supports (but is not limited to) architectures such as RNN, GPT, GRU, and BERT. Recurrent Neural Networks (RRNs) update the hidden state h_t using the current input and the previous hidden state, thus $h_t = f(h_{t-1}, w_t)$. RNNs can take the form of a Gated Recurrent Unit (GRU) [81], which uses reset $r_t = \sigma(W_r \cdot [h_{t-1}, w_t])$, and update $u_t = \sigma(W_u \cdot [h_{t-1}, w_t])$ gates to control how it combines the previous hidden state h_{t-1} with a new candidate state $h'_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, w_t])$ to obtain the updated hidden state $h_t = (1 - u_t) \odot h_{t-1} + u_t \odot h'_t$.

Furthermore, GPT and BERT are deep architectures known as Transformers [33]. In transformer models, the hidden state h_t in each time step is updated through a multi-head self-attention mechanism and a feed-forward network. In simpler terms, $h_t = \text{Attention}(Q_t, K_t, V_t) = \text{softmax}(Q_i K_i^T / \sqrt{d_k}) V_i$, where Q_i, K_i, and V_i represent the queries, keys, and values for the time step t, and d_k is the dimension of the key vectors. Transformers adopt the form of Encoder-based (*e.g.,* BERT) or Decoder-based (*e.g.,* GPT).

We employed RNNs and Transformers for all our scenarios as RNNs are widely used in SE [82], while Transformers have gained popularity in the SE/NLP domain for their high performance [4]. In Fig. 9), we opted to train the NCMs for cases $[A-F]$ to gain more control over the training data. This was particularly important given our need to manipulate the tokenizer, allowing us to assign special tokens to specific code categories (*i.e.,* syntax clustering). However, do_{code} is not confined to specialized models trained by the user as demonstrated in the G case, in which we used an in-the-wild BERT model. Therefore, do_{code} operates effectively on already pre-trained NCMs as long as there exists a way to group fine-grained predictions (*i.e.,* BPE tokens) to human-understandable categories (*e.g.,* AST/Grammar-based or keyword-based).

Table 1 provides a detailed overview of the training configurations for each scenario. In $[A-F]$ cases, the NCMs were trained using the Java portion of the *CodeSearchNet* dataset [34], which consists of a diverse array of methods (mts) from GitHub [83]. We partitioned *CodeSearchNet* into

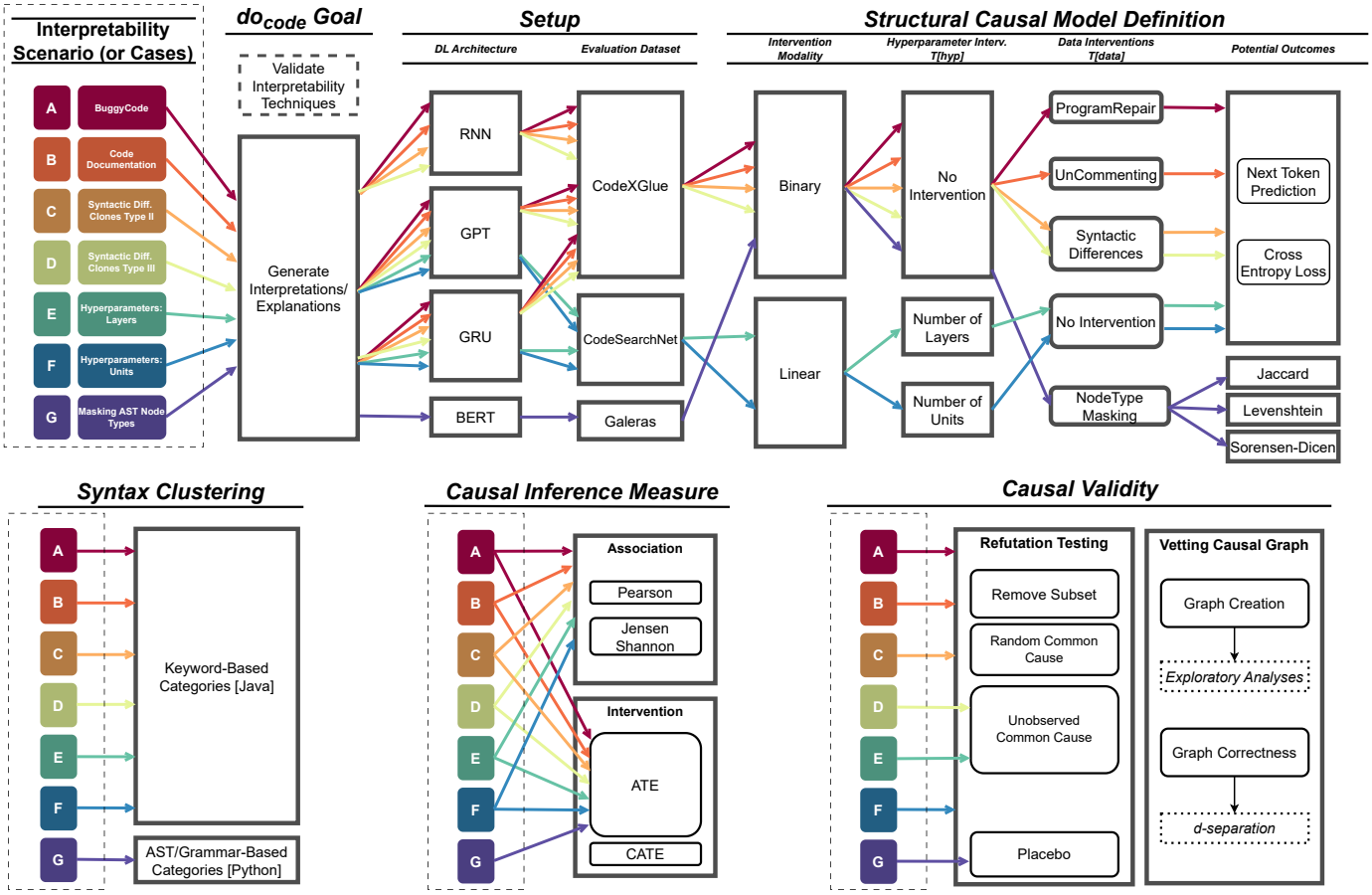


Fig. 9: Configuration criteria for enabling causal interpretability. Seven scenarios are proposed for this case study. However, new interpretability cases can be formulated using a different permutation or extending the criteria.

training, validation, and test sets. For model development, we employed Tensorflow and Pytorch [84], [85], along with Huggingface's Transformers library [86]. To mitigate overfitting, training was terminated if cross-entropy did not improve by at least 1e-2 over 5 epochs. Inputs were standardized with a *start of sentence* token and adjusted to a length of 300. This process was executed on an Ubuntu 20.04 system with an AMD EPYC 7532 32-Core CPU, an A100 NVIDIA GPU with 40GB VRAM, and 1TB RAM. For the G case, we used a pre-trained BERT model: codebert-base-mlm [54], trained specifically for a Masking Language Model (MLM) objective and also using the *CodeSearchNet* dataset.

TABLE 1: NCMs training specifications for Recurrent Neural Networks (*i.e.*, Vanilla RNN, GRU), and Transformers (*i.e.*, GPT-2, BERT)

NCMs Training

RNNs	**Transformers**	**Hyper.**	**Val.**
$NCM_{lyr,unt}$	$NCM_{lyr,hds}$	dropout RNN [87]	0.5
$RNN_{1,1024}$	$GPT\text{-}2_{6,12}$	dropout TF	0.1
$GRU_{1,1024}$	$GPT\text{-}2_{12,12}$	optimizer [88]	adam
$GRU_{2,1024}$	$GPT\text{-}2_{24,12}$	learning rate	1e-3
$GRU_{3,1024}$	$BERT_{12,12}$	beta1,beta2	0.9
$GRU_{1,512}$		epsilon	1e-7
$GRU_{1,2048}$		epochs	64
		batch RNN,TG	512,128

About Evaluation Datasets. To build the data intervention testbeds for our interpretability scenarios, we obtained *Java* samples from *CodeXGLUE* [89] and *Python* samples from *Galeras* [74]. In $[A-F]$ cases, parallel corpora were used to examine the impacts of buggy code (*BuggyTB*: 64,722 methods or mts), code documentation (*CommentsTB*: 6,664 mts), and syntactic alterations in semantically similar snippets, specifically focusing on type II (*BigClone2TB*: 666 mts) and type III (*BigClone3TB*: 8,097 mts) clones from *BigCloneTB*. For the G case, we conducted binary interventions by masking tokens corresponding to each AST element in the *Galeras* samples (8,299 mts), covering all node types in the Python grammar as defined by tree-sitter [90]. In the control group, a comparable number of tokens were randomly masked in each sample.

About Code Tokenization. For all scenarios, Byte Pair Encoding (BPE) tokenization [91] was applied to the testbeds before processing them with the NCMs. Known for its efficacy in training NCMs on code, BPE significantly mitigates the *out-of-vocabulary* problem [87]. For $[A-F]$ cases, we developed a BPE tokenizer trained on 10% of our training data and with a vocabulary size of 10K. Conversely, for the G case, we employed the pretrained BPE tokenizer from the BERT-selected model. Nonetheless, the BPE tokenization process sometimes resulted in sub-tokens that either combined multiple reserved keywords or split keywords across different tokens. This splitting issue presented a significant challenge for our interpretability analysis as our method relies on accurately grouping token predictions into semantic

categories, a criterion that we named *syntax clustering*. To address this, we developed clustering functions to ensure the tokens are correctly aligned with our defined categories, which will be discussed in Sec. 9.4.

9.3 Structural Causal Model Definition

do_{code} users must design the Structural Causal Model (SCM) based on their domain knowledge and available data. This criterion involves choosing the intervention modality, specifying the type of the intervention, and computing the potential outcome. The SE-based interventions, contingent upon the objectives of do_{code}, primarily fall into two categories: Data Interventions $T_{[data]}$ and Hyper-parameter Interventions $T_{[hyp]}$. Data interventions occur within the testbed, whereas Hyper-parameter interventions involve model training parameters (*e.g.,* Learning rate, Batch Size, Number of Epochs, Number of Hidden Layers/Units, dropout rate), which are not embedded in the data per se. Moreover, potential outcomes cover a variety of measures such as Next Token Prediction (NTP), Cross Entropy Loss, BLEU, CODEBLEU, and distance similarity scores including Jaccard, Levenshtein, and Sorensen-Dice. We aimed to estimate the causal effect of $T_{[data]}$ and $T_{[hyp]}$ interventions on different potential outcomes. Fig. 10 describes seven SCMs, the expected potential outcomes with their corresponding treatments T_{A-G}, and examples of interventions per scenario. Additionally, Table 2 depicts the interventions and datasets used for each scenario.

TABLE 2: Overview of SE-based Interventions Experiments.

Counterfactual Interventions

Type	Interv.	Case Id	Intervention	Associated Dataset	PL
$T_{[data]}$	T_A	A	*ProgramRepair*	*BuggyTB* [72]	Java
	T_B	B	*UnCommenting*	*CommentsTB* [34]	Java
	T_C	C	*SyntacticDifferences*	*BigClone2TB* [73]	Java
	T_D	D	*SyntacticDifferences*	*BigClone3TB* [73]	Java
	T_G	G	*ASTNodeTypes*	*Galeras* [74]	Python
$T_{[hyp]}$	T_E	E	*NumberLayers*	*CodeSearchNet* [34]	Java
	T_F	F	*NumberUnits*	*CodeSearchNet* [34]	Java

For A and B cases, we examined the impact of buggy code and inline comments on NCM's predictions. The A case assessed how the presence of bugs influences the cross-entropy and NTP. Similarly, the B case explored the effect of including inline comments on the model's performance.

For C and D cases, we assessed how NCMs respond to minor and major syntactic variations in semantically equivalent code snippets, focusing on the effects of semantic-preserving changes on code generation. Due to the absence of a natural split in clone types in the *BigCloneTB* dataset, a typical control and treatment approach for causal analysis was impractical. The selection between $function_1$ and $function_2$ methods, as categorized by *BigCloneTB*, is arbitrary. Instead, we concentrated on how syntactic differences in methods performing identical functions impact our models. Hence, we used the differences between these function sets as our primary confounders and intervention. Specifically, we employed the Levenshtein Distance – a metric quantifying the necessary edit operations (insert, modify, remove) to transform one sequence into another – to simulate a 'refactoring' treatment. This approach approximated the number of edits needed to convert one method into another and allowed us to bypass the need for a natural split between $function_1$ and $function_2$. It is worth noting that the associated graphs include the instrumental variable *countSubWords*. We found evidence of a subtle correlation between treatments and *countSubWords* in Table 11.

For E and F cases, as indicated in Table 2, our case study involved exploring two distinct deep learning architectures. We particularly examined how variations in hyper-parameters such as layers and units within each architecture influence the prediction performance.

For the G case, our goal was to generate interpretations for the prediction of AST node types using BERT. To achieve this, we essentially constructed the dataset by implementing two treatments: masking tokens corresponding to AST Node types (Treatment 1) and randomly masking an equivalent number of tokens (Treatment 2). We then calculated the normalized similarity distances (*i.e.,* Jaccard, Levenshtein, and Sorensen-Dice) between the AST of the predicted code and the ground truth samples, which served as the potential outcomes in the SCM.

9.4 Syntax Clustering of Code Predictions

Consider the situation where a developer inserts a `'('` character after the `'main'` keyword in a function declaration in Java (①-Fig. 7). Inherently, a developer mentally rationalizes several things such as the concept of a function declaration and expected Java syntax. If a NCM can make a similar prediction, it suggests to us that it has *statistically learned* some understanding of the concept of a function declaration and corresponding syntax. Therefore, we assert that by associating human-interpretable categories (*e.g.,* Programming Languages Keywords or AST nodes) to model predictions and then analyzing the statistical properties of those predictions, we can begin to learn how well a given NCM reflects human knowledge.

A major conjecture in interpretability research is that NCMs are more understandable when they *reflect human knowledge* [92]. One way of determining whether a model reflects human knowledge is testing it to see whether or not it operates (or predicts) *similar to how a human would operate*. do_{code} accomplishes this by grouping code token predictions of NCMs to human interpretable categories.

To help bridge the gap between a given NCMs token-level representation of code and human-understandable categories, do_{code} aggregates individual tokens to well-known syntactic categories from programming languages. Source code tokens can be clustered to any number of syntactic elements, and particularly for do_{code}, we focus on aggregating tokens to different syntactic categories, which ***do not*** require manual labeling. This syntax clustering mitigates the cost involved with large-scale data labeling and still provides explanations rooted in categories with which most programmers and researchers are likely familiar.

The syntax clustering comprises high-level properties of code using a **clustering function** defined by $\phi_{\mathcal{H}} : \vec{w} \rightarrow \vec{h}$, in which the vector $\vec{w}$ corresponds to tokens from a vocabulary $\mathcal{V}$. Thus, each token in a sequence w is assigned to a specific syntax-understandable category h. Note that do_{code} allows the definitions of any potential clustering function, users are not forced to use our clustering category system $\mathcal{H}$.

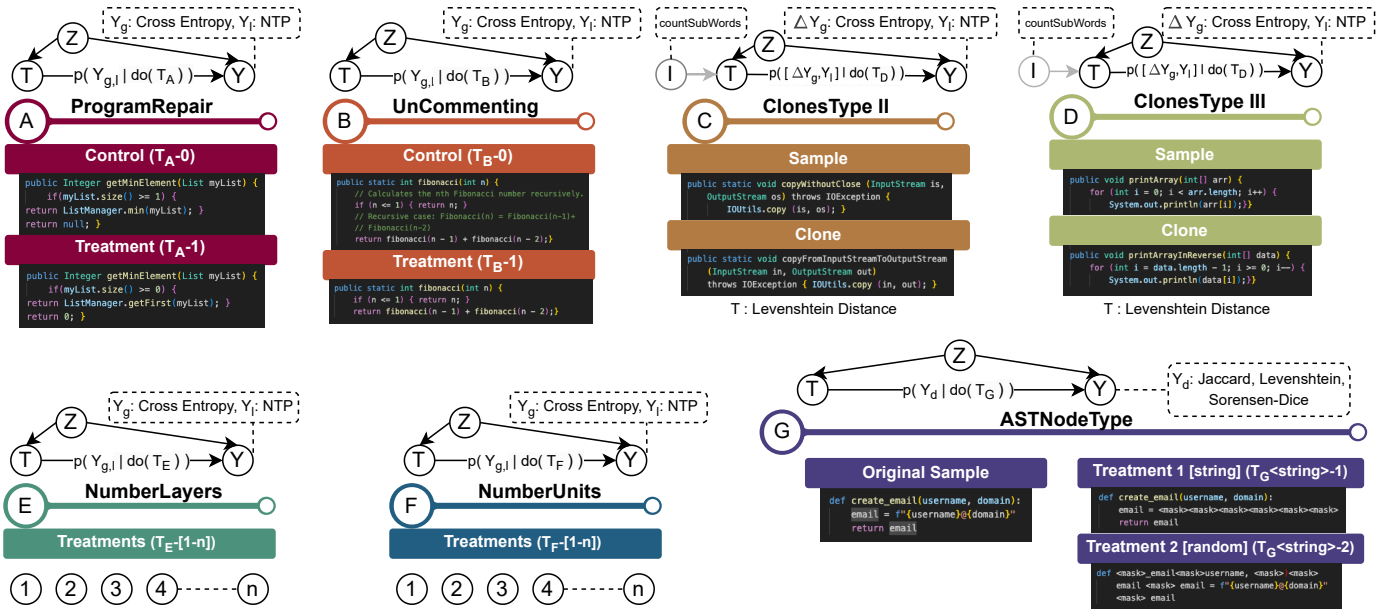


Fig. 10: Structural Causal Models' Definition. Each SCM presents the causal estimand, the potential outcomes, the intervention type, and examples of interventions for each interpretability scenario.

The proposed categories in our system $\mathcal{H}$ are classified into keyword-based or grammar-based. Below, we pose a separate clustering function for Java and another one for Python.

9.4.1 Keyword-Based Aggregation

In programming languages, different types of tokens retain different semantic meanings. For instance `'='` and `'<'` are common `[operators]`. Therefore, tokens can be grouped into semantically meaningful keyword-based *categories* $\mathcal{H}$. We can establish a clustering function $\phi_{\mathcal{H}} : \vec{w} \rightarrow \vec{h}$, in which a token w in a snippet s is clustered to a corresponding keyword category h. We propose an initial set of nine categories for Java. Figure 11 illustrates the categories and associated keyword tokens. For each category, we group keywords by functionality that were collected from the official Oracle Documentation and Manuals [93]. These keywords remain consistent across Java versions from Java 7 onwards. In addition, we present an error analysis in Sec. 11.3 to explore the validity of this clustering function.

9.4.2 AST/Grammar-Based Aggregation

Every expression in a programming language is determined by production rules defined in its Context-Free Grammar defined by the expression $\mathcal{H} = (\alpha, \lambda, \omega, \beta)$, in which α denotes the finite set of non-terminal nodes, λ the finite set of terminal nodes, ω the finite set of production rules, and β the start node. The set of production rules ω for any type of statement (*e.g.*, conditional, assignation, operator) is expressed in terms of the terminal and non-terminal nodes (α, λ). In programming languages, terminal and non-terminal nodes retain different semantic meanings. Following this, we can establish a clustering function $\phi_{\mathcal{H}} : \vec{w} \rightarrow \vec{h}$, where each token w in a snippet s is aggregated and assigned to a corresponding node h. We extracted a total of 196 node types (*i.e.*, terminal and non-terminal), from tree-sitter Python's grammar [94]. This set of categories was then used to group tokens from code snippets. Figure 11 offers a visual example of this clustering.

9.5 Causal Inference Measures

Below, we explain association and intervention metrics that we estimate to validate the reliability of an interpretation. Association metrics serve as baselines that we need to confirm for confounding after computing intervention metrics, which represent actual causal effects.

Association Metrics. For $[A - F]$ cases, we employed two methods to empirically estimate the association distributions $p(Y_g|T)$ and $p(Y_l|T)$. These two methods are the classic Pearson correlation and the Jensen-Shannon distance. For the latter, imagine we wish to understand the correlation between syntactic changes, *i.e.*, variable renaming, alterations in white space, *etc.* and a NCMs performance. One way we can study this is through computing the association of Cross-Entropy values Y under two treatments, the first $T = 0$, would be an unaltered code snippet and the second $T = 1$ would be its Type III clone. Computing this association can be done using the Jensen Shannon distance $p(Y|T) \approx JS(Y^0, Y^1)$ as defined in Def. 6 for four models. Fig. 12 shows the distributions of Y^0 and Y^1 with their distances after applying bootstrapping as an example.

Definition 6. **Jensen-Shannon Distance (JS).** The Jensen-Shannon divergence (JSD) overcomes the asymmetric computation of the KL divergence and provides a measure of the difference between distributions Eq. 4. The JS distance is the square of the JS divergence $p(Y|T) \approx JS(Y^{T=0}, Y^{T=1}) = JSD(Y^0, Y^1)^2$. JS is proportional to the influence of T on Y, which measure the separation of the distributions Y^0, Y^1. The notation $Y^{T=0}$ refers to the potential outcomes observed under the treatment $T = 0$.

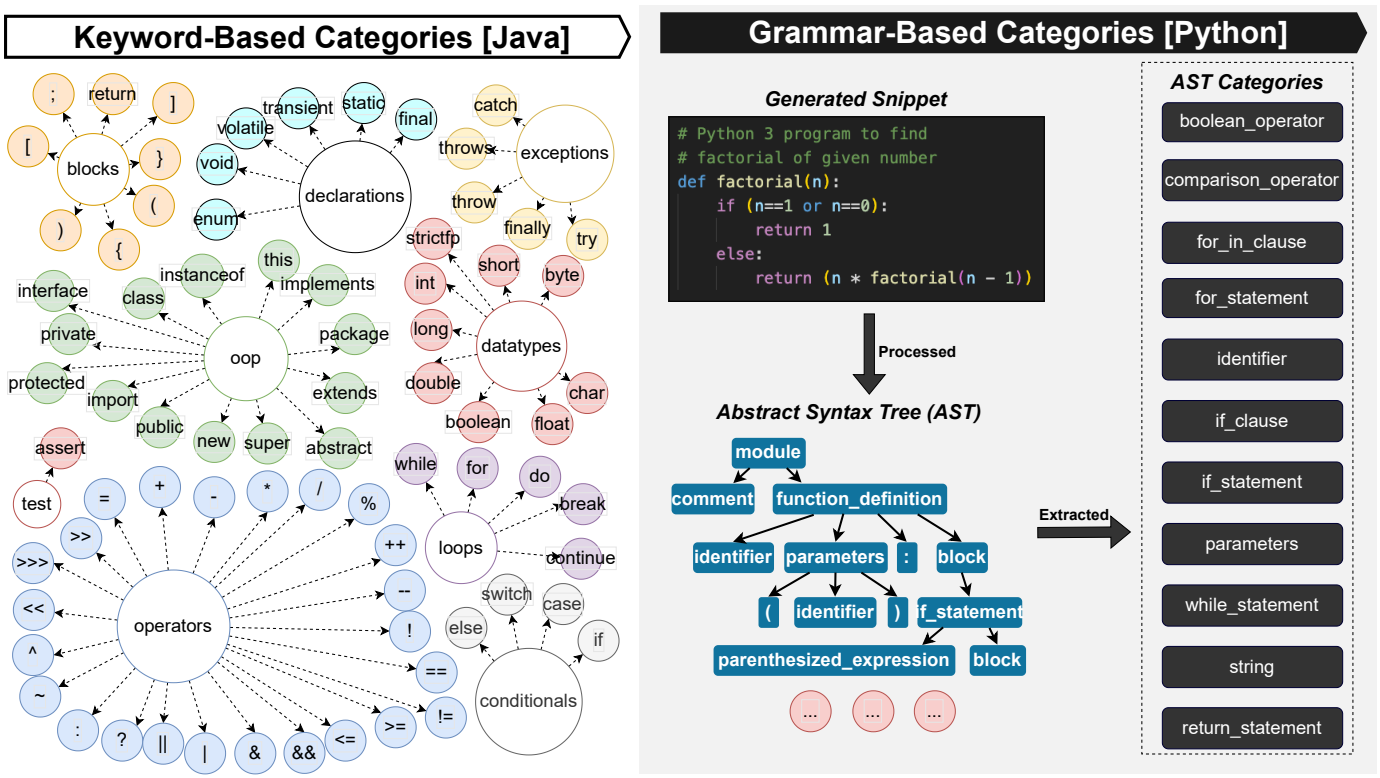


Fig. 11: Code Syntax Clustering. Code predictions are grouped into Keyword-based or AST/Grammar-based categories.

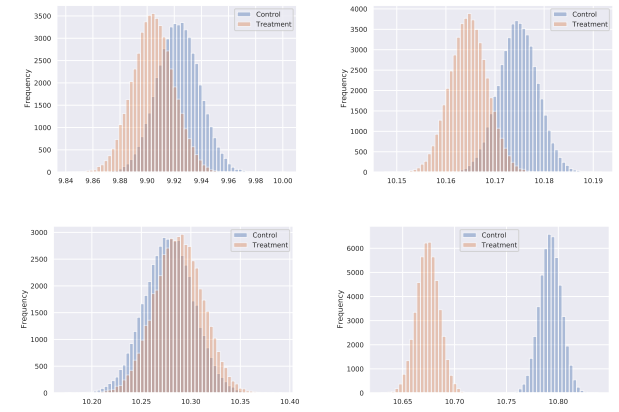


Fig. 12: *SyntacticDifferences* Intervention (*BigClone3TB*) for Global Performance: Bootstrapped Cross-Entropy. Top Left: $\text{RNN}_{1,1024}$ ($JS = 0.3$), Top Right: $\text{GRU}_{1,1024}$ ($JS = 0.8$), Bot Left: $\text{GPT-2}_{6,12}$ ($JS = 0.6$), Bot Right: $\text{GPT-2}_{12,12}$ ($JS = 1$).

$$JSD(Y^0 = y^0, Y^1 = y^1) = \tag{4a}$$

$$\frac{1}{2}\left[D_{KL}\left(y^0||\frac{y^0+y^1}{2}\right) + D_{KL}\left(y^1||\frac{y^0+y^1}{2}\right)\right] = \tag{4b}$$

Intervention Metrics. Conversely, the interventional distributions $p(Y_g|do(T))$, $p(Y_l|do(T))$, and $p(Y_d|do(T))$ are estimated in terms of the Average Treatment Effect (ATE), as previously introduced in Eq .3. do_{code} can potentially estimate other types of causal effect metrics such as *Conditional Average Treatment Effect (CATE)*.

9.6 Causal Validity

About Refutation Testing. After do_{code} estimates causal effects, these effects are validated for robustness. Ergo, do_{code} incorporates various refutation methods to evaluate the sensitivity of the causal estimations as we defined in Sec. 8. We employ the following refutations: introducing a random common cause (*i.e.,* independent common causes added randomly should not impact the causal estimates), applying placebo treatments (*i.e.,* causal effect should go zero if the true treatment is replaced by an independent random variable), considering unobserved common causes (*i.e.,* causal estimates should not be too sensitive when we add an independent common cause correlated with the treatment and outcome), and performing data subset validations (*i.e.,* the causal effect should not change when the dataset is replaced by a random subset).

About Vetting Causal Graph. We conducted exploratory analyses in Sec. 11 that provide statistical evidence of the causal assumptions encoded in the graph. These analyses comprise measuring for correlations among the graph's variables. In addition, the *testability* of each proposed causal graph occurs in the identification step (Sec. 6). The identification step consists of finding an estimand for each proposed graph using the *d-separation* criteria. These criteria help us to verify the fitness of the causal models in the datasets they generate. We can reject or accept the causal graph depending on the conditional independence tested from the data. All proposed SCMs in Fig. 10 were accepted.

9.7 Research Questions

Inspired by the notion of Pearl's Ladder of Causation [29], [95], we framed two research questions to explore do_{code}'s potential in enabling causal interpretability for NCMs. $\mathbf{RQ}_1$ computes the feasibility of some SE-based Interventions and $\mathbf{RQ}_2$ performs a sensitivity analysis to validate the robustness of previously computed causal effects.

$\mathbf{RQ}_1$: Causal Inference: *To what extent do SE interventions affect code prediction?*

$\mathbf{RQ}_2$: Causal Validity: *How robust is the underlying causal graph under assumed SE interventions and code confounders?*

10 Results

This section presents the results for both research questions per interpretability scenario introduced in Sec. 9.

To answer $\mathbf{RQ}_1$, we begin by examining the correlations between potential outcomes and the interventions defined for each scenario. These correlations provide an early indication of how much the interventions might impact the outcomes. Table 3 offers an overview of the correlation coefficients observed between the Cross-Entropy (*i.e.,* Y_g 5.2) and our proposed interventions. Additionally, Table 4 details the Pearson correlations results for NTP values (*i.e.,* Y_l 5.2) for our *SyntacticDifferences* interventions, calculated with all the models we used in our experiments excluding $\text{BERT}_{12,12}$ (used in scenario G).

Next, do_{code} generates causal explanations of code predictions for each scenario. Tables 5 and 6 display the computed causal effects for cross-entropy (Y_g). Similarly, Tables 7 and 8 report both the causal inference metrics and correlation coefficients for each token category within Y_l. Table 9 presents both the correlation and causal effects computed for explaining the distance predictions (*i.e.,* Y_d 5.2) in scenario G.

We answer $\mathbf{RQ}_2$ by validating our causal effect estimates employing the four standard refutation methods outlined in Section Sec. 8.

10.1 Interpretability Scenario A: BuggyCode

To what extent does *ProgramRepair* affect the predictions of NCMs? For the A case, focusing on the *ProgramRepair* intervention, both $\text{RNN}_{1,1024}$ and $\text{GPT-2}_{6,12}$ models exhibit considerably high JS distances of 0.73 and 0.67, indicating a strong correlation between this intervention and the cross-entropy: $P(Y_g|T_A)$. Additionally, we observed a similar pattern in NTP outcomes for $Y_{l\texttt{[blocks]}}$ with the $\text{GRU}_{1,1024}$ model, which had the highest correlation at 0.626 for *ProgramRepair*.

However, when we controlled for SE covariates, the correlations initially observed turned out to be misleading for causation. Across the three models ($\text{RNN}_{1,1024}$, $\text{GRU}_{1,1024}$, and $\text{GPT-2}_{6,12}$), the ATEs were very small, both in cross-entropy $p(Y_g|do(T_A))$ with values of $-3E-4$, $-2.33E-05$, and $-2E-4$, and in NTP outcomes $p(Y_{l\texttt{[blocks]}}|do(T_A))$ with a value of $-5.25E-4$. Having null effects after observing high correlations confirms the presence of confounding bias for this case. Therefore, the prediction performance is being affected by other confounders different from the actual buggy code intervention.

How robust are the causal effect estimations? To verify the sensitivity of the ATEs for Y_g, we computed four robustness tests: $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$. The results for $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_4$ aligned closely with the ATEs we found for the three models. Furthermore, the results for $\mathcal{R}_3$ were nearly zero. This indicates that the estimated causal effects were robust. Similarly, $\mathcal{R}_1$ values for Y_l results were close to the obtained ATEs.

> **BuggyCode Finding F_A:** The presence or absence of buggy code does not appear to causally influence (or explain) the prediction performance of our NCMs even under measured high correlation.

10.2 Interpretability Scenario B: Code Documentation

To what extent does *UnCommenting* (or inline comments) intervention affect the predictions of NCMs? In the *UnCommenting* interventions, there were no strong correlations in cross-entropy $P(Y_g|T_B)$ for the $\text{RNN}_{1,1024}$, $\text{GRU}_{1,1024}$, and $\text{GPT-2}_{6,12}$ models, with values of 0.18, 0.22, and 0.25 respectively. However, the NTP outcomes showed significant associations in specific categories: for $\text{GRU}_{1,1024}$, a strong correlation in `[datatype]` ($P(Y_{l\texttt{[datatype]}}|T_B) = 0.749$), and for $\text{GPT-2}_{6,12}$, notable correlations in `[operators]` ($P(Y_{l\texttt{[operators]}}|T_B) = 0.998$), `[conditionals]` ($P(Y_{l\texttt{[conditionals]}}|T_B) = 0.821$), and `[tests]` ($P(Y_{l\texttt{[tests]}}|T_B) = 0.724$).

Once we adjusted for covariates, the ATEs in both $p(Y_g|do(T_B))$ and $p(Y_l|do(T_B))$ were close to zero, showing a trend towards no causal effects. This suggests that actions like removing comments from code have little to no causal impact on cross-entropy or NTP values.

How robust are the causal effect estimations?The obtained results across refutation tests indicated unstable causal effects. Specifically, the outcomes for $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_4$ did not align closely with the ATEs. This discrepancy could stem from several factors: 1) a lack of sufficient data samples, 2) inaccuracies in our causal diagram assumptions (*e.g.,* confounders, instrumental variables, and effect modifiers), and/or 3) the treatment is inadequate.

> **Code Documentation Finding F_B:** Despite observing strong correlations between the removal of comments and NTP, we cannot causally interpret code predictions from inline comments since measured ATEs are not robust after refutations. This suggests that other hidden confounders are influencing the estimation of this causal effect.

10.3 Interpretability Scenario C: Clones Type II

To what extent does *SyntacticDifferences* intervention (Clones Type II) affect the predictions of NCMs? In the *SyntacticDifferences* intervention for Type II clones, we observed a positive correlation between Levenshtein "edit" distance and the difference of cross-entropy values (*i.e.,* $p(\Delta Y_g|T_C)$) across $\text{RNN}_{1,1024}$, $\text{GRU}_{1,1024}$, and $\text{GPT-2}_{6,12}$. The correlation coefficients were 0.45, 0.6, and 0.452 for each model respectively, as detailed in Table 6.

TABLE 3: *Jensen-Shannon Dist.*, and *Pearson Corr.* values obtained between Cross-Entropy Y_g and Treatments in scenarios A,B,C,D and F (**bold**:strong corr.)

			SyntacticDifferences		
	ProgramRepair - T_A	***UnCommenting*** - T_B	**BigClone2TB** - T_C	**BigClone3TB** - T_D	***NumberLayers*** - T_F
NCM	JS Dist.	JS Dist.	Pearson	Pearson	Pearson
$\text{RNN}_{1,1024}$	$\mathbf{0.730}_{JS}$	0.180_{JS}	$\mathbf{0.45}_{PR}$	-0.056_{PR}	-
$\text{GRU}_{1,1024}$	0.230_{JS}	0.220_{JS}	0.598_{PR}	0.14_{PR}	-0.093_{PR}
GPT-$2_{6,12}$	$\mathbf{0.670}_{JS}$	0.250_{JS}	0.452_{PR}	-0.14_{PR}	-0.485_{PR}

TABLE 4: Average *Pearson Corr.* of *SyntacticDifferences* interventions using all RNNs, GRUs, and GPTs.

	SyntacticDifferences	
Category	*Clones Type II* - T_C	*Clones Type III* - T_D
$[blocks]$	0.12 ± 0.039	0.214 ± 0.107
$[exceptions]$	0.097 ± 0.068	0.036 ± 0.102
$[top]$	0.388 ± 0.241	0.037 ± 0.023
$[tests]$	1.0 ± 0.0	nan $\pm$ nan
$[declarations]$	0.192 ± 0.128	0.161 ± 0.135
$[conditionals]$	0.078 ± 0.103	0.077 ± 0.144
$[loops]$	0.183 ± 0.163	0.019 ± 0.094
$[operators]$	0.316 ± 0.202	0.072 ± 0.05
$[datatypes]$	0.016 ± 0.083	-0.009 ± 0.064
$[extra]$	0.163 ± 0.065	0.159 ± 0.14

Conversely, we found weak correlations in NTP scores across all NCMs (excluding $\text{BERT}_{12,12}$) in the nine categories analyzed. For instance, Y_l was correlated with `[oop]` tokens and `[operators]` with $p(Y_{l\texttt{[oop]}}|T_C) = 0.388 \pm 0.241$ and $p(Y_{l\texttt{[operators]}}|T_C) = 0.316 \pm 0.202$, as detailed in Table 4. The high standard deviation observed across the NCMs in these categories indicates a wide variance in what the models statistically learned.

The computed causal effects on ΔY_g, using backdoor criterion, showed an exceptionally high influence of clones Type II on the cross-entropy, with values of 0.63, 0.87, and 0.56 for $\text{RNN}_{1,1024}$, $\text{GRU}_{1,1024}$, and GPT-$2_{6,12}$. On the other hand, when we used $I_{[countSubWords]}$ as an instrumental variable for estimand identification, the observed causal effects were substantially lower (see 6).

Since most of the ATEs for Y_l could not be computed, using backdoor criterion or the instrumental variable, we omitted showcasing these results in Table 8. Formulating continuous interventions was hindered by the non-linear distribution of the data points. This can be explained due to the nature of Clone Types II, which oftentimes, maintains the same functionality while introducing minimal alterations in the structure of the source code. This led to a lower variability in the computed Levenshtein distances.

How robust are the causal effect estimations? For testing the robustness of the ATEs, we could not compute the refutation methods *Placebo* and *Remove Subset* due to limited data. However, the other two methods, *Random Comm. Cause* and *Unobserved Comm. Cause*, showed stability, reinforcing our confidence in the ATEs for global results.

Clones Type II Finding F_C: The presence of Clones Type II impacts (or causally explains) the cross-entropy on $\text{RNN}_{1,1024}$, $\text{GRU}_{1,1024}$, and GPT-$2_{6,12}$, as obtained correlations and causal effects suggest an influence of some interventions in identifiers, literals, types, white spaces, layout, and comments.

10.4 Interpretability Scenario D: Clones Type III

To what extent does *SyntacticDifferences* intervention (Clones Type III) affect the predictions of NCMs? By contrast, our analysis showed a different pattern for Type III clones. We detected no strong correlations between the Levenshtein "edit" distance and the cross-entropy (*i.e.*, $p(\Delta Y_g|T_D)$) in any of our NCMs. This is evident from the marginal correlation values for the three models—$\text{RNN}_{1,1024}$, $\text{GRU}_{1,1024}$, and GPT-$2_{6,12}$: -0.056, 0.14, and -0.14, respectively. Additionally, we found even lower correlation values in NTP scores (*i.e.*, $P(Y_l|T_D)$) across all NCMs (excluding $\text{BERT}_{12,12}$), as detailed in Table 4. The highest coefficient is observed for `[extraTokens]` tokens, with a value of $p(Y_{l\texttt{[extraTokens]}}|T_C) = 0.159 \pm 0.14$.

In our models, the causal effect on cross-entropy using the back-door criterion was generally small but showed interesting variations. For the GPT-$2_{6,12}$ model, this effect was negative, with $p(Y_g|do(T_D)) \approx -0.274$, whereas, for the $\text{GRU}_{1,1024}$ model, it was positive at $p(Y_g|do(T_D)) \approx 0.11$. We observe a tendency to null causal effects when using $I_{[countsubWords]}$ as instrumental variable to identity the causal estimand.

Additionally, the causal effect on NTP outcomes for `[blocks]` was negligible $p(Y_{l\texttt{[blocks]}}|do(T_D)) = 2.80E-05$. Unfortunately, most of the ATEs could not be computed for each keyword category since it was unfeasible to fit a linear model due to the shape of the data (*i.e.*, grouped predictions by syntax categories have similar values).

How robust are the causal effect estimations? Similar to the findings with clone type II, testing the robustness of the ATEs presented challenges. Due to limited data, we were unable to apply $\mathcal{R}_3$ and $\mathcal{R}_4$. However, the use of the other two methods, $\mathcal{R}_1$ and $\mathcal{R}_2$, demonstrated stability in our results.

Clones Type III Finding F_D: The presence of Clones Type III does not consistently impact (or causally explain) the cross-entropy and NTP, as revealed by negative and positive ATEs obtained across the NCMs under analysis.

10.5 Interpretability Scenario E: Layers

To what extent does *NumberLayers* intervention affect the predictions of NCMs? We observed that *NumberLayers* interventions are negatively correlated with cross-entropy. This trend is evident in the values of $p(Y_g|T_E)$ for $\text{GRU}_{1,1024}$ and GPT-$2_{6,12}$ models: -0.093 and -0.485 respectively. This suggests that as the number of layers increases, the cross-entropy tends to decrease. Conversely, we found strong correlations for NTP outcomes

TABLE 5: Causal Interventions $p(Y_g|do(T))$ of Cross-Entropy across models and datasets. (background:best effect)

	***ProgramRepair* - T_A**			***UnCommenting* - T_B**			***NumberLayers* - T_E**	
NCM	$RNN_{1,1024}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$	$RNN_{1,1024}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$
Causal Eff. ATE	-0.0003	-2.33E-05	-0.0002	0.0023	2.90E-05	0.0026	-0.0058	-0.0124
Random Comm. Cause	-0.0003	-2.45E-05	-0.0002	0.0011	-0.0004	0.0015	-0.0058	-0.0124
Unobserved Comm. Cause	-0.0003	1.54E-05	-0.0001	0.0002	-0.0001	0.0007	-0.0050	-0.0108
Placebo	0.0001	1.44E-05	0.0001	0.0006	-1.33E-05	0.0006	-0.0001	-2.77E-05
Remove Subset	-0.0003	-3.68E-05	-0.0002	0.0012	-0.0003	0.0016	-0.0058	-0.0124

TABLE 6: Causal Interventions $p(Y_g|do(T))$ of Cross-Entropy across Clone datasets. (background:best effect)

	SyntacticDifferences											
	Backdoor Criterion						Instrumental Variable $I_{[countSubWords]}$					
	***BigClone2TB* - T_C**			***BigClone3TB* - T_D**			***BigClone2TB* - T_C**			***BigClone3TB* - T_D**		
NCM	$RNN_{1,1024}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$	$RNN_{1,1024}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$	$RNN_{1,1024}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$	$RNN_{1,1024}$	$GRU_{1,1024}$	$GPT\text{-}2_{6,12}$
Causal Eff. ATE	0.6288	0.8713	0.5635	-0.1042	0.1085	-0.2739	0.00076	0.0018	0.0017	-0.18	0.72	-0.25
Random Comm. Cause	0.6297	0.8720	0.5651	-0.1043	0.1084	-0.2741	0.00076	0.0018	0.0017	-0.18	0.72	-0.25
Unobserved Comm. Cause	0.2950	0.4257	0.2737	-0.0800	0.0830	-0.2168	0.00077	0.0018	0.0018	-0.18	0.72	-0.26

TABLE 7: NTP Association Results $p(Y_l|T)$ are *Jensen-Shannon Dist*. Causal Effects are ATEs $p(Y_l|do(T))$ (**bold**:strong corr., background:best effect)

	***ProgramRepair* - T_A**						***UnCommenting* - T_B**					
NCM		$GRU_{1,1024}$			$GPT\text{-}2_{6,12}$			$GRU_{1,1024}$			$GPT\text{-}2_{6,12}$	
Categories	*Association JS Dist.*	*Causal Eff. ATE*	*R1*	*Association JS Dist.*	*Causal Eff. ATE*	*R1*	*Association JS Dist.*	*Causal Eff. ATE*	*R1*	*Association JS Dist.*	*Causal Eff. ATE*	*R1*
[*blocks*]	0.374	-0.0005	-0.00052	0.794	-0.0001	-0.000115	**0.867**	0.0003	0.000488	**0.948**	-0.0006	-0.000352
[*exceptions*]	**0.893**	-1.00E-06	1.00E-06	0.349	-1.20E-05	-1.10E-05	**0.835**	-4.80E-05	-4.80E-05	**0.909**	-1.20E-05	-4.20E-05
[*oop*]	**0.952**	2.00E-05	1.20E-05	**0.942**	-1.30E-05	-9.00E-06	**0.930**	-7.00E-06	-4.70E-05	**0.910**	-4.90E-05	-2.60E-05
[*tests*]	0.419	9.00E-06	8.00E-06	0.003	1.00E-05	9.00E-06	0.405	1.90E-05	4.00E-05	0.276	-1.70E-05	2.00E-06
[*declarations*]	**0.929**	4.00E-06	4.00E-06	**0.946**	1.00E-06	1.00E-06	**0.939**	-2.10E-05	-0.000145	**0.852**	4.60E-05	6.00E-05
[*conditionals*]	0.655	-3.90E-05	-3.90E-05	**0.824**	-8.00E-06	-1.00E-05	**0.889**	8.00E-06	7.40E-05	0.179	-0.000596	-0.000603
[*loops*]	**0.870**	-2.00E-06	-3.00E-06	0.677	2.10E-05	1.90E-05	**0.973**	-1.70E-05	1.50E-05	**0.915**	1.80E-05	3.00E-06
[*operators*]	**0.877**	-6.00E-06	-5.00E-06	**0.961**	1.50E-05	1.10E-05	0.722	0.0002	0.000345	0.002	0.0079	0.008991
[*datatypes*]	0.747	-1.00E-05	-1.00E-05	**0.832**	9.00E-06	1.00E-05	0.251	0.0003	0.000328	0.568	0.0002	0.000111
[*extra*]	**0.831**	2.60E-05	2.00E-05	**0.879**	5.70E-05	6.00E-05	0.632	0.0014	0.001016	0.474	0.0024	0.002004

TABLE 8: NTP Association Results $p(Y_l|T)$ are *Pearson Corr*. Causal Effects are ATEs $p(Y_l|do(T))$. We use $-$ to indicate undetermined causal effects due to an unfeasible linear model.(**bold**:strong corr., background:best effect)

	***SyntacticDifferences* - T_D**						***NumberLayers* - T_E**					
NCM		$GRU_{1,1024}$			$GPT\text{-}2_{6,12}$			$GRU_{1,1024}$			$GPT\text{-}2_{6,12}$	
Categories	*Association Pearson*	*Causal Eff. ATE*	*R1*	*Association Pearson*	*Causal Eff. ATE*	*R1*	*Association Pearson*	*Causal Eff. ATE*	*R1*	*Association Pearson*	*Causal Eff. ATE*	*R1*
[*blocks*]	0.026	-1.50E-05	-1.50E-05	0.186	-2.80E-05	-2.80E-05	-0.102	-0.010559	-0.010559	**0.725**	0.018004	0.018004
[*exceptions*]	0.017	-	-	0.002	-	-	-0.070	-	-	0.349	-	-
[*oop*]	0.049	-	-	0.012	-	-	0.019	-	-	0.255	-	-
[*tests*]	-	-	-	-	-	-	-0.130	-	-	0.174	-	-
[*declarations*]	0.375	-	-	0.034	-	-	-0.257	-	-	0.405	-	-
[*conditionals*]	0.274	-	-	-0.087	-	-	-0.009	-	-	**0.682**	-	-
[*loops*]	0.024	-	-	0.111	-	-	0.042	-	-	0.275	-	-
[*operators*]	0.099	-	-	0.062	-	-	-0.032	-	-	0.389	-	-
[*datatypes*]	0.037	-	-	-0.069	-	-	0.002	-	-	0.275	-	-
[*extra*]	0.192	-3.00E-05	-3.00E-05	-0.017	5.40E-05	5.40E-05	0.092	0.014588	0.014588	**0.606**	0.014525	0.014525

across keyword-based categories ($p(Y_l|T_E)$): the categories `[blocks]`, `[conditionals]`, and `[extraTokens]` showed notably high correlations, with values of 0.725, 0.682, and 0.606 respectively.

Nonetheless, we detected confounding bias in almost all categories of our intervention analysis for NTP outcomes. This was indicated by the extremely low ATEs, such as $p(Y_{\texttt{[blocks]}}|do(T_E)) = 0.018$ and $p(Y_{l\,\texttt{[extraTokens]}}|do(T_E)) = 0.0145$. A plausible explanation for this phenomenon can be traced to SE metric confounders. For instance, the size of the methods appears to have a significant influence on both cross entropy and NTP results across the NCMs.

How robust are the causal effect estimations? To verify the sensitivity of the ATEs for Y_g, we estimated $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$. The results for $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_4$ aligned closely with the ATEs we found for $GRU_{1,1024}$ and $GPT\text{-}2_{6,12}$. Furthermore, $\mathcal{R}_3$ results were nearly zero. Thus, the estimated causal effects were robust. Similarly, $\mathcal{R}_1$ values for Y_l results were close to the obtained ATEs, reinforcing the robustness of our findings.

Layers Finding F_E: Although it is observed strong correlations between the number of layers and NTP, which might suggest a causal interpretation of code predictions, the reported ATEs close to zero demonstrate the presence of confounding bias.

10.6 Interpretability Scenario F: Units

To what extent does *NumberUnits* intervention affect the predictions of NCMs?

In a similar vein, *NumberUnits* interventions tend to be negatively correlated with the cross-entropy. The number of units showed a negative effect with the $GRU_{1,1024}$ model, with $p(Y_g|T_F) \approx -0.084$.

Likewise, we found negative values for $P(Y_l|T_F)$, suggesting that increasing the number of units negatively impacts the NTP outcomes across all keyword-based categories. However, these values were relatively low, making it challenging to draw definitive conclusions. For example, the highest negative correlation observed was $P(Y_{l\,\texttt{[exceptions]}}|T_F) \approx -0.253$ for NTP.

TABLE 9: Masking AST Node Types. Distance Association Results $p(Y_d|T_G)$ are *Pearson Corr* and Causal Effects are ATEs $p(Y_d|do(T_G))$. (**bold**:strong corr., background:best effect)

Distance metric Y_d	***ASTNodeTypes* - T_G**								
	<jaccard>			<levenshtein>			<sorensen-dice>		
Categories	*Association assoPR*	*Causal Eff. ATE*	*R4*	*Association assoPR*	*Causal Eff. ATE*	*R4*	*Association assoPR*	*Causal Eff. ATE*	*R4*
[boolean_operator]	**-0.3294**	-0.1508	0.0052	-0.2845	-0.1363	0.0147	-0.2809	-0.0947	0.0009
[comparison_operator]	-0.2648	-0.0272	-0.0066	-0.2071	-0.0183	-0.0031	-0.2228	-0.0154	-0.0019
[for_in_clause]	**-0.4232**	-0.0532	-0.0004	**-0.3724**	-0.0449	0.0013	**-0.3568**	-0.0291	0.0002
[for_statement]	**-0.4006**	-0.1011	0.0341	-0.2525	-0.0411	-0.0124	**-0.3949**	-0.0832	-0.0037
[identifier]	0.0024	-0.0752	0.0230	-0.0375	-0.0734	-0.0017	0.0243	-0.0387	0.0068
[if_clause]	**-0.3904**	-0.0407	0.0002	**-0.3666**	-0.0365	5.9143	**-0.3395**	-0.0220	0.0011
[if_statement]	**-0.3667**	-0.1211	-0.0328	-0.2409	-0.0933	-0.0130	**-0.3624**	-0.0954	-0.0131
[parameters]	**-0.3287**	-0.0481	0.0071	**-0.3230**	-0.0457	-0.0104	-0.2951	-0.0294	0.0032
[return_statement]	-0.2450	-0.1211	0.0062	-0.2250	-0.1127	-0.0158	-0.2261	-0.0756	-0.0083
[string]	**-0.3651**	-0.1683	-0.0041	-0.2961	-0.1450	0.0274	**-0.3199**	-0.1161	0.0036
[while_statement]	-0.2677	-0.0964	-0.0197	-0.1383	-0.0091	-0.0027	-0.2979	0.0113	0.9199

Negative correlations observed for the cross-entropy were consistent with the corresponding negative ATEs estimated. However, the negative correlation observed for `[blocks]` was not in fact causal: $P(Y_{l\texttt{[blocks]}}|do(T_F)) \approx$ -5E-06. Similarly, for the other categories, the ATEs were so minimal that they can be considered as causal null effects.

We omit the results of *NumberUnits* intervention from Tables 5, 6, 7 and 8, since the causal effect estimates for Y_g using $\text{RNN}_{1,1024}$ and $\text{GPT-2}_{6,12}$, as well as most Y_l categories, were either null or could not be determined due to the shape of the data (see online Appendix in [35]).

How robust are the causal effect estimations? Just as with *NumberLayers*, we conducted robustness tests $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$ for Y_g. The outcomes of $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_4$ closely matched the ATEs we observed in the $\text{GRU}_{1,1024}$ model. Furthermore, the $\mathcal{R}_3$ values were nearly zero, reinforcing the robustness of the estimated causal effects. Similarly, the $\mathcal{R}_1$ values for Y_l are also closely aligned with the corresponding ATEs, further confirming the robustness of the results.

Units Finding F_F: Intervening the number of units tends to be negatively correlated with the cross-entropy; in fact, measured causal effects on NTP are null suggesting that this intervention does not explain code predictions.

10.7 Interpretability Scenario G: Masking AST Nodes

To what extent does *ASTNodeTypes* intervention affect the predictions of NCMs? We computed the distance outcomes Y_d for some AST node types, as detailed in Sec. 9.4.2. However, in Table 9, we specifically focus on a subset of both terminal and non-terminal nodes. This subset was chosen for its familiarity with developers and includes elements such as conditional statements, identifiers, and repetition statements. The intervention demonstrated strong negative correlations between AST node types and the distance outcomes Y_d, consistent across all distance metrics. For example, $P(Y_{d\texttt{<jaccard>}}|T_{G\texttt{[for\_in\_clause]}}) = -0.4232$, $P(Y_{d\texttt{<levenshtein>}}|T_{G\texttt{[for\_in\_clause]}}) = -0.3724$, and $P(Y_{d\texttt{<sorensen-dice>}}|T_{G\texttt{[for\_in\_clause]}}) = -0.3568$.

The correlation results are further supported by the estimated ATEs $P(Y_d|do(T_{G\texttt{[node]}}))$, demonstrating no confounding bias. Although the causal effects are lower than the estimated correlations, they reveal the presence of negative causation.

How robust are the causal effect estimations? We calculated $\mathcal{R}_1$, and we obtained the same values for ATEs indicating that the causal effects are robust. Furthermore, we also calculated $\mathcal{R}_4$, and the values were close to zero.

Masking AST Nodes Finding F_G: The intervention of masking random tokens has more impact on code predictions than masking grammar-based categories. This suggests that $\text{BERT}_{12,12}$ does not entirely capture the nodes' information of Abstract Syntax Trees (ASTs).

11 Exploratory Causal Analyses

This section introduces supplementary analyses to validate the assumptions made during the formulation of the SCMs in our interpretability scenarios, thereby facilitating the *causal discovery* of our graphs. We exhaustively explored the datasets of interventions to support the encoding process from domain knowledge. We assess the validity of the causal graph encoding by exploring correlations among SE confounders (see Sec. 5.3), potential outcomes, and interventions. In other words, by finding statistical dependencies among models' variables, we provide sufficient evidence for creating SCMs.

It is worth clarifying that we employ the term *covariates* to refer to a general set of Software Engineering metrics that describe code. Conversely, the term *confounders* refers to a proven subset of covariates that influence treatments and outcomes. Therefore, some covariates might or might not be confounders. Because this exploration is usually conducted before confounder identification, preliminary correlational analyses are performed with a rudimentary list of SE covariates.

Moreover, we introduce a descriptive *error analysis* of the syntax clustering categories in Sec. 9.4. Specifically, we measured the average prediction by keyword-based aggregations for each Neural Code Model baseline. The error analysis aims to explore the feasibility of the proposed clustering function, which is fundamental for defining interpretable outputs across the do_{code} pipeline.

11.1 Statistical dependencies between confounders and potential outcomes $\textcircled{Z} \rightarrow \textcircled{Y}$

The accuracy of NCMs is significantly influenced by the context window size as these architectures rely heavily on contextual information to predict the next token [96]. Motivated by the previous premise, we found a strong correlation between the number of words $Z_{[countSubwords]}$

TABLE 10: Most correlated covariates influencing cross-entropy results (*i.e.*, Y_g) and NTP scores (*i.e.*, Y_l) across NCMs and datasets (**bold**:strong corr.)

		Cross Entropy Loss		*Next Token Prediction*			
Dataset	**Model**	**Correlation with [z_count_subwords]**	**Max correlation excluding [z_count_subwords]**	**Max Correlation with [z_count_subwords]**		**Max correlation excluding [z_count_subwords]**	
		ρ	Covariate $[Z]$	ρ	Potential Outcome $[Y_l]$	**Potential Outcome $[Y_l]$, Covariate** $[Z]$	ρ
BuggyTB	$\text{RNN}_{1,1024}$	**0.87**	$Z_{[loc]}$	0.25	$Y_{[loops]}$	0.1	$Y_{[blocks]}$, $Z_{[loc]}$	-0.417
BuggyTB	$\text{GRU}_{1,1024}$	0.36	$Z_{[parenthesizedExpsQty]}$	-0.47	$Y_{[blocks]}$	-0.072	$Y_{[blocks]}$, $Z_{[parenthesizedExpsQty]}$	**0.682**
BuggyTB	$\text{GPT-2}_{6,12}$	**0.88**	$Z_{[loc]}$	0.27	$Y_{[tests]}$	0.065	$Y_{[blocks]}$, $Z_{[parenthesizedExpsQty]}$	0.476
CommentsTB	$\text{RNN}_{1,1024}$	0.41	$Z_{[uniqueWordsQty]}$	**0.54**	$Y_{[tests]}$	0.197	$Y_{[tests]}$, $Z_{[uniqueWordsQty]}$	0.3
CommentsTB	$\text{GRU}_{1,1024}$	0.15	$Z_{[maxNestedBlocksQty]}$	0.35	$Y_{[operators]}$	0.082	$Y_{[blocks]}$, $Z_{[parenthesizedExpsQty]}$	0.152
CommentsTB	$\text{GPT-2}_{6,12}$	0.39	$Z_{[uniqueWordsQty]}$	0.52	$Y_{[extraTokens]}$	0.059	$Y_{[declarations]}$, $Z_{staticMethodsQty}$	-0.245

and the Cross-Entropy ($\rho = 0.87$) for the $\text{RNN}_{1,1024}$ using the *BuggyTB* dataset. A similar trend was observed using $\text{GPT-2}_{6,12}$ with a correlation value of $\rho = 0.88$. For other architectures with different configurations (see Table 10), the correlation still exists below 0.5, indicating that while there is some evidence of dependencies, the impact of other potential covariates could be stronger. For instance, in the case of the *CommentsTB* dataset, when using both $\text{RNN}_{1,1024}$ and $\text{GPT-2}_{6,12}$, the most significant covariate impacting performance was the number of unique words $Z_{\text{[uniqueWordsQty]}}$, with correlation coefficients of $\rho = 0.54$ and $\rho = 0.52$, respectively.

More broadly, Table 10 showcases the Pearson correlation values (ρ) after exploring the dependencies among proposed covariates (see Sec. 5.3), the *BuggyTB* and *CommentsTB*. These testbeds were used for the *ProgramRepair* intervention in scenario A and the *SyntacticDifferences* intervention in scenario B.

Similarly, we computed correlation coefficients between our keyword-based categories and potential covariates. Table 10 reveals that the NTP scores grouped by categories are not significantly correlated with $Z_{[countSubwords]}$, as evidenced by the low maximum coefficient values. Conversely, $Z_{[paranthesizedExpsQty]}$ seems to be appreciably correlated with the prediction of $[blocks]$ in the *BuggyTB* dataset using $\text{GRU}_{1,1024}$ ($\rho = 0.682$), and $\text{GPT-2}_{6,12}$ ($\rho = 0.476$). These findings seem to correspond with the grammar rules of Java since the parenthesized expressions are often used within blocks of code.

Finding Ⓩ $\rightarrow$ Ⓨ : The statistical dependency between covariates and outcomes varies across different NCMs. The Cross-Entropy is highly correlated to the number of subwords for the *BuggyTB* dataset. In contrast, the *CommentsTB* dataset exhibits lower correlation values. Additionally, NTP scores for *blocks* tokens show a strong dependency on the number of parenthesized expressions for both binary treatments.

11.2 Statistical dependencies between confounders and interventions Ⓩ $\rightarrow$ Ⓣ

In this analysis, we delve into statistical dependencies between covariates Z and interventions. Specifically, *ProgramRepair*, *UnCommenting*, and *SyntacticDifferences* (Clones Type II and III) interventions. We assess and compare the distributions of covariates Z across the treatment and control groups within each dataset using the Jensen-Shannon distance. Measured divergence and distance values do not exhibit substantial evidence of a significant separation between the number of tokens ($Z_{[countSubwords]}$) in the control and treatment distributions. Furthermore, all covariate distributions follow a similar tendency of lower distances between treatment and control groups (see Table 11).

TABLE 11: Differences between $Z_{\text{[countSubwords]}}$ Treatment and Control groups across multiple datasets using Jensen-Shannon divergences and distances.

	Control vs Treatment	
Dataset	***ΔJS Divergence***	***ΔJS distance***
CommentsTB	0.005	0.07
BuggyTB	0.004	0.06
BigClone2TB	0.004	0.06
BigClone3TB	0.002	0.04

Findings Ⓩ $\rightarrow$ Ⓣ: A subtle separation between control and treatment distributions exists when segregating by any SE covariate Z.

11.3 Code Syntax Clustering Error Analysis

We conducted an exploratory analysis to evaluate and describe the behavior of NCMs predictions grouped by keyword-based categories. These categories are detailed in Fig. 11. Specifically, we computed the normalized NTP values of tokens for each sample in the dataset *BuggyTB*, used in the *ProgramRepair* intervention for scenario A. Subsequently, we clustered token-predicted probabilities for each of the categories. The exploration aims to unveil the average prediction *error* for a set of outcomes Y_l for three NCMs: $\text{GPT-2}_{6,12}$, $\text{RNN}_{1,1024}$, and $\text{GRU}_{1,1024}$. An *error* is any keyword-based category whose average predicted probability is lower than 0.5.

This exploration reveals that each keyword-based category follows a similar error tendency regardless of the NCM type as illustrated in Fig. 13. For example, the $[blocks]$ category in $\text{GPT-2}_{6,12}$ exhibited the highest median value at 0.09 followed by $[extraTokens]$ and $[tests]$ with a value of 0.06 and 0.01 respectively. Conversely, $[operators]$ has the lowest median value, and the interquartile range (*iqr*) for $[extraTokens]$ is narrower than $[blocks]$. Similar distributions are observed in $\text{RNN}_{1,1024}$ and $\text{GRU}_{1,1024}$, where $[blocks]$, $[tests]$ and $[extraTokens]$ are among the categories with better prediction values. Moreover, a peculiar behavior is visible in the $[OOP]$ category for $\text{GRU}_{1,1024}$, where the NTP scores are slightly higher, as indicated by the third quartile approaching 0.05. However, for half of the data points, the values remain close to the median, which is nearly zero, as highlighted by the second quartile.

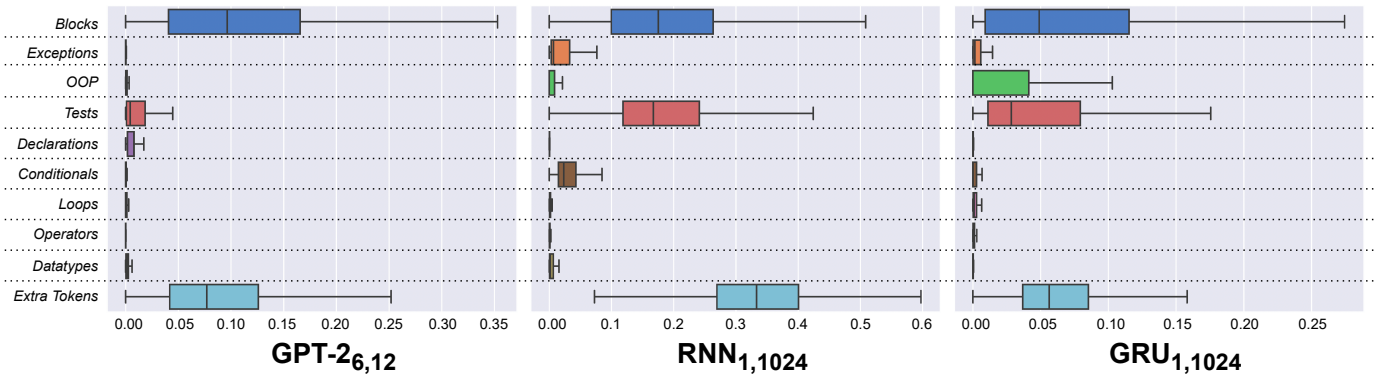


Fig. 13: Descriptive Error Analysis. Keyword-Based Categories probabilities (Normalized Y_l - NTP) for GPT-2$_{6,12}$ RNN$_{1,1024}$ and GRU$_{1,1024}$ on the *BuggyTB* dataset. Higher values indicate less erroneous predictions.

More broadly, the highest normalized NTP value observed among the categories falls below 0.5, which suggests an absence of empirical evidence that supports a significant *statistical understanding* related to these categories. Nonetheless, our error analysis entails a statistical technique demonstrating the feasibility of measuring code predictions by syntax categories.

We also investigated the distribution of tokens by category within each Java dataset associated with $T_{[data]}$ interventions: *BuggyTB*, *CommentsTB*, *BigClone2TB*, and *BigClone3TB*. Our findings, depicted in Fig. 14, reveal that tokens categorized under $[blocks]$ are the most frequent across four datasets. On the other hand, tokens associated with $[tests]$ are consistently the least frequent.

Error Analysis Finding: GPT-2$_{6,12}$, RNN$_{1,1024}$ and GRU$_{1,1024}$ produce erroneous Y_l predictions (*i.e.,* normalized NTP < 0.5) for all the syntax categories in the *BuggyTB* dataset.

12 Discussion

The ultimate goal of interpretability in Deep Learning for Software Engineering is to generate explanations of NCMs' predictions. We achieved this goal by formulating causal interpretations of such predictions, quantified through performance values (*e.g.,* Cross-Entropy, Next Token Prediction, or Distance Metrics). We demonstrate that regardless of conventional evaluations of NCMs (*e.g.,* BLEU, Accuracy, Perplexity), producing causal interpretations is vital for model understanding. For example, even if a model demonstrates a deficient BLEU score when its outputs are evaluated against the ground truth for a downstream task, we can still generate causal explanations of those outputs by defining an appropriate SCM.

Our interpretability method, do_{code}, generates causal explanations (or interpretations) based on the idea of estimating *the effect of* ***interventions***. These interventions take place on the data inputs or parameters that configure NCMs. Furthermore, estimating their causal effects is grounded in Pearl's theory of causation [29], [37], [76].

Below, we pose three aspects of the discussion: 1) some general insights from the case study, 2) a brief explanation of the guidelines to use do_{code} in practice, and 3) a list of challenges and opportunities practitioners might face when adapting do_{code} for their analyses.

12.1 Insights From The Case Study

In do_{code}, we examine model semantics by categorizing code tokens into different conceptual categories and examining both raw model performance according to these groups, and the causal relationships of these categories across treatments. Syntax can be measured by how often the model predicts a given token that is syntactically correct. While we did not directly report syntactic correctness, in our observations, our studied models rarely made syntactic errors. However, the local prediction performance of tokens across different semantic categories tends to vary quite a bit. This is partially explained by the prevalence of different token categories in the training set, but further work on model architectures that improve the performance of underperforming token groups could help to advance research on NCMs.

For each NCM, a strong correlation across testbeds for a given code token category means that we observe the prediction performance to have *changed* significantly across the SE-based treatments (*e.g.,* commented/uncommented code). This change could correspond to prediction performance for these token categories increasing or decreasing. However, because our approach uses a causal graph (*i.e.,* Structural Causal Model), we can determine the amount of causal effect between the correlated variables based on covariates, and determine whether or not they are **spurious correlations**, due to *confounding bias* Fig. 4, or true causal relationships caused by the change in treatment. Based on our experiments, NCMs seem to have a stronger *statistical understanding* of syntax and a more limited understanding of semantics, according to our definition of semantics. However, once we started exploring Masking Language Models such as BERT-like code models, we found the opposite: AST node tokens are not better predicted than a random and, therefore, unstructured set of tokens.

As the results suggest, our NCMs under study learn to predict tokens related to code blocks (*e.g.,* brackets, parentheses, semicolons) more effectively than most other code token types (*e.g.,* loops, conditionals, datatypes), we found that our NCMs are *sensitive to seemingly subtle changes in code syntax*, reinforcing previous studies concluding the same [65], and our models are only *marginally impacted* by

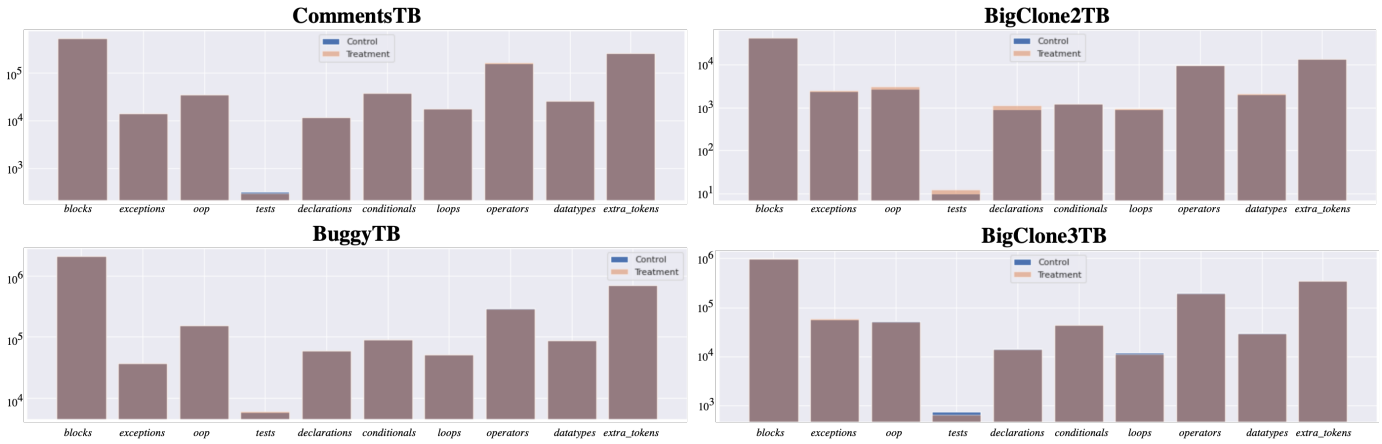


Fig. 14: Frequency analysis of keyword-based categories by NCM.

the presence of comments and bugs, which challenges findings from previous work [97]. Consequently, practitioners can identify which *categories* of code tokens were important in the model's decision-making process. In other words, do_{code} supports the identification of which 1) tokens, 2) layers, or 3) hyperparameters are impacting code predictions, which we describe below:

Tokens. In Sec. 9.4, we propose a *syntax clustering* function for grouping the NCMs' code predictions into more understandable syntax categories (*i.e.,* keyword-based and AST/Grammar-based). Although the syntax clustering formalism was originally omitted from the do_{code} pipeline, we realized during the experimentation phase that a clustering strategy is vital as practitioners must make sense of generated subwords or fine-grained levels of code.

Our syntax clustering formalism stems from the need to make fine-grained code predictions more understandable to practitioners. As such, our approach enables determining which token categories a model can learn (or not) across different application settings (*e.g.,* SE-based interventions). For example, given the normalized NTP values computed for scenario A, as depicted in Fig. 13, one could conclude that block-category tokens are more effectively predicted by GPT-$2_{6,12}$, $\text{RNN}_{1,1024}$ and $\text{GRU}_{1,1024}$. Yet such NCMs typically struggled to predict operator-category tokens. More importantly, we can observe causal relationships related to the importance of code tokens in decision-making *changes* across treatments (*i.e.,* data interventions or model parameters alterations).

Layers. Our method supports interventions on the *number of layers* for a given architecture. Therefore, do_{code} can identify if the layers, as a hyperparameter, influence the performance. Nonetheless, identifying specific layers to influence the performance would require going beyond the causal interpretability scope and exploring the inner mechanisms of the neural network [30].

Hyperparemeters. do_{code} supports the identification of hyperparameters that influence neural network performance by extending SE-based intervention definitions at the model level (see Sec. 5.1). Our case study supports two types of parameters: *NumberOfLayers* and *NumberOfUnits*.

About causal graph validity. In our exploratory analyses, we conducted experiments accounting for other types of variables in the SCM beyond treatments, outcomes, and confounders. For example, we collected information on software-based effect modifiers (*i.e.,* variables affecting only outputs) and instrumental variables (*i.e.,* variables affecting only treatments). Particularly, for semantic preserving treatments, we made *subword counting* an instrumental variable instead of a confounder. We found that these graph configurations were not robust under refutation methods. However, further research is required to construct proper tools to identify instrumental variables and modifiers.

About Causal Transportability. Although we concentrated on datasets and keyword-based categories for Java (*i.e.,* scenarios $A - F$) in our exploratory analyses, the assumptions we made to build the SCMs still hold for scenario G. This *transportability* of assumptions across similar domains have been researched in Causal Inference [98]. As such, the causal information learned from our experiments can be reused in analogous settings if there is homogeneity in the effect modifiers. Transportability allows us to maintain our assumptions across scenarios, as long as the underlying structure of the causal graph remains consistent, keeping the same type of treatments, potential outcomes, and confounding variables.

12.2 do_{code} in practice

While it may appear that NCMs have begun to achieve promising performance, it is insufficient to *only* generate code predictions (*i.e.,* the ***what*** of NCMs' decision). This current status quo, at best, provides an incomplete picture of the limitations and caveats of NCMs for code. Given the potential impact and consequence of these models and their resulting applications, there is a clear need to strive for a more complete understanding of how they function in practice. As such, we must push to understand how NCMs arrive at their predictions (*i.e.,* the ***why*** of NCMs' decision) as shown in our proposed Structural Causal Models for each scenario. With the design and results of interpretability scenarios, we demonstrate that do_{code} comprises a causal explanation method that aims to make Deep Learning for Software Engineering NCMs, and their decision-making process, understandable to practitioners.

To that end, we have created a checklist that summarizes the general process researchers can use to apply causal interpretability to Neural Code Models 15. For instance, one of the practical applications of do_{code} is facilitating the

debugging process of NCMs. By debugging a NCM we refer to the tasks of reducing the amount of *confounding bias* between SE-interventions and code predictions. Our proposed guidelines below help to design a pipeline in which do_{code} facilitates the detection of this bias for a given setting.

As shown in Fig. 15, the proposed checklist comprises five guidelines, which correspond to both the approach pipeline (see Sec. 4) and the definition of a syntax clustering function (see Sec. 9.4). In the first guideline, a researcher determines what would be understandable to the target audience and constructs a syntax clustering function. This clustering function translates the model's fine-grained code predictions to the target audience. Once this clustering is built, the researcher can move on to the second guideline that resolves around defining a Structural Causal Model (SCM). It is important to have this step after defining the clustering function so the interventions, outcomes, and confounders can be properly modeled. With the SCM at hand, in the third guideline, the researcher can now collect the data holding the observable SE-based interventions (*e.g.,* buggy vs. fixed code), used to estimate the causal effect of the treatment. Then, in the fourth guideline, existing popular libraries (*e.g.,* doWhy) can be used to estimate the causal effect. Lastly, in the fifth guideline, the researcher must check the original assumptions (*i.e.,* the confounders and SCM) using refutation methods. With this checklist, we hope to ease the complexity around causal analysis for researchers.

12.3 Challenges & Future Work

In this section, we list some challenges CHs that practitioners might face when adapting our method to their interpretability analyses.

CH_1**: Proposing a new syntax clustering criterion.** Our proposed categories, designed to group fine-grained code predictions into more understandable categories, are based entirely on definitions and guidelines extracted from Programming Languages (*i.e.,* Java and Python). This might represent a limitation if a practitioner requires a more in-depth analysis of the syntax interactions. do_{code} introduces a clustering formalism that can be extended and reformulated to specific interpretability needs. Different clustering functions give rise to different challenges. For instance, one BPE token may simultaneously belong to multiple categories and span across different AST nodes. Future research needs to address the following questions: How do practitioners deal with this overlapping? Do they enforce a singular classification for the BPE token, in a similar way to our keyword-based clustering? or do they allow for some statistically controlled overlaps as we did in AST/Grammar-based clustering?

CH_2**: Collecting data for formulating SE-based interventions.** do_{code} was not implemented to perform *causal discovery*, unveiling the causes of code predictions directly from observations. In fact, do_{code} is restricted to estimate causal inference measures (*e.g.,* correlations, ATEs, CATEs) based on hypotheses and assumptions, from domain knowledge, embedded in Structural Causal Models. Therefore, users of do_{code} must define and formulate specific SE-based interventions depending on the available data, and also provide the possible confounding factors that can alter the effect on code predictions. Future work can be oriented to facilitate the definition of the Structural Causal Model into a more automatic approach. Such an automatic approach should include recent techniques on causal discovery.

CH_3**: Integrating do_{code} in Deep Learning for Software Engineering life-cycle.** To use do_{code} in practical settings, we must propose strategies for integrating interpretability approaches into Software Engineering pipelines that incorporate Deep Learning models. This will not only improve the reliability of the system but also facilitate the monitoring of critical information.

CH_4**: Creating the Structural Causal Model.** The quality of the causal explanations generated by do_{code} heavily relies on the structure of the causal graph. Consequently, employing do_{code} in practical scenarios would require statistical evidence supporting each SCM's component. Although we provide evidence supporting these components for the proposed scenarios, it may not hold for every new setting. For instance, the confounders and code syntax clustering introduced in our study might differ for functional programming languages. Building a sound SCM would entail automatically identifying potential confounders and formulating assumptions from a rigorous causal discovery process not covered in our pipeline.

In summary, our interpretability method do_{code} is based on the idea of outlining causal queries, given a SCM defined from the domain knowledge. These causal queries are obtained by estimating an interventional distribution, where the potential outcome is generally a prediction performance value of the Neural Code Model under study. Furthermore, the interventions are a set of software-based properties that help us construct explanations about the generative model. do_{code} has been implemented in an open-source library, which is available in our repository [35].

13 Conclusion

We presented do_{code}, an interpretability method for understanding NCMs for code. do_{code} combines rigorous statistical instruments and causal inference theory to give rise to contextualized SE explanations of NCMs using Structural Causal Models (SCMs). SCMs provide a more robust and interpretable framework for modeling complex systems such as Neural Code Models in software engineering. SCMs can help to better understand the underlying mechanisms and causal relationships between different variables, which can improve the accuracy and generalizability of deep learning models. In addition, SCMs can provide a more transparent and explainable approach to Deep Learning for Software Engineering, allowing for a better understanding of the decision-making process of the model and facilitating more effective detection of *confounding bias* and error analysis. Additionally, we also carried out a case study evaluation using do_{code} on popular NCMs, namely, RNNs, GRUs, and Transformers, to interpret code predictions. do_{code} exposes erroneous syntax-based categories (*e.g.,* conditionals, or loops) by examining Average Treatment Effects and refutation methods. We hope our research opens the field for posterior empirical analysis in interpretability to empower researchers with statistical and causal inference methods.

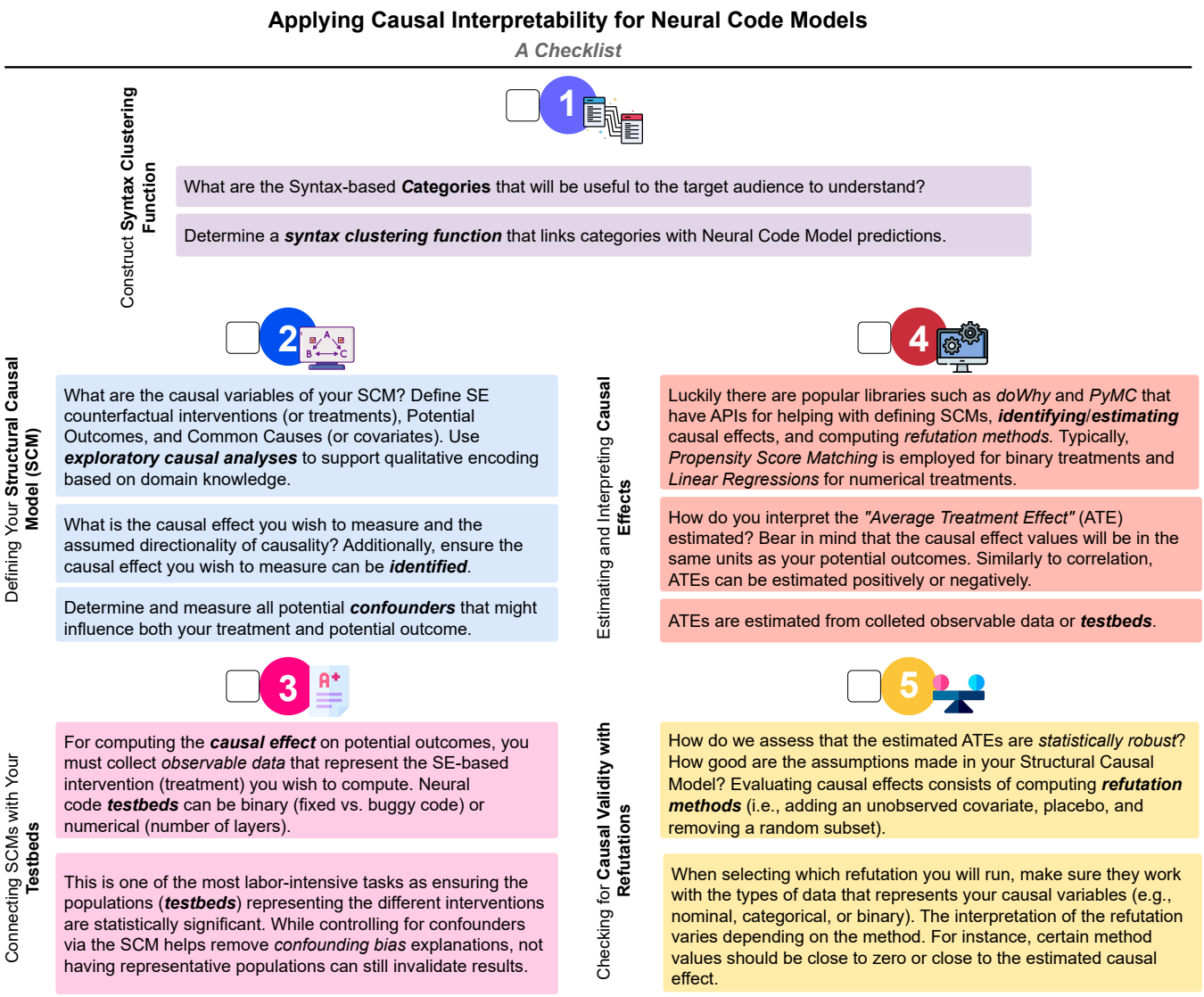


Fig. 15: Five Guidelines for Applying Causal Interpretability to Neural Code Models.

Acknowledgments

This research has been supported in part by the NSF CCF-2311469, CNS-2132281, CCF-2007246, CCF-1955853, CCF-2311468, and CCF-2132285. We also acknowledge support from Cisco Systems. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] C. Watson, M. Tufano *et al.*, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1398–1409. [Online]. Available: https://doi.org/10.1145/3377811.3380429

[2] M. White, C. Vendome *et al.*, "Toward Deep Learning Software Repositories," in *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories (MSR'15)*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[3] M. Ciniselli, N. Cooper *et al.*, "An empirical study on the usage of transformer models for code completion," 2021.

[4] A. Mastropaolo, S. Scalabrino *et al.*, "Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks," pp. 336–347, 2021.

[5] M. Tufano, C. Watson *et al.*, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 542–553.

[6] M. White, M. Tufano *et al.*, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 479–490.

[7] R. Tufano, L. Pascarella *et al.*, "Towards automating code review activities," in *43rd International Conference on Software Engineering, ICSE'21*, 2021. [Online]. Available: https://arxiv.org/abs/2101.02518

[8] M. Ciniselli, N. Cooper *et al.*, "An empirical study on the usage of BERT models for code completion," *CoRR*, vol. abs/2103.07115, 2021. [Online]. Available: https://arxiv.org/abs/2103.07115

[9] Z. Chen, S. J. Kommrusch *et al.*, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[10] Microsoft, "Visual studio intellicode — visual studio - visual studio." [Online]. Available: https://visualstudio.microsoft.com/services/intellicode/

[11] "Code faster with ai code completions." [Online]. Available: https://www.tabnine.com/

[12] W. Zaremba, G. Brockman, and OpenAI, "Openai codex," Aug 2021. [Online]. Available: https://openai.com/blog/openai-codex/

[13] GitHub, "Github copilot · your ai pair programmer." [Online]. Available: https://copilot.github.com/

[14] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Softw.*, vol. 23, no. 4, p. 76–83, Jul. 2006. [Online]. Available: https://doi.org/10.1109/MS.2006.105

[15] M. Chen, J. Tworek *et al.*, "Evaluating large language models trained on code," 2021.

[16] T. Wu, M. T. Ribeiro *et al.*, "Errudite: Scalable, reproducible, and testable error analysis," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 747–763. [Online]. Available: https://aclanthology.org/P19-1073

[17] M. T. Ribeiro, T. Wu *et al.*, "Beyond accuracy: Behavioral testing of NLP models with CheckList," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4902–4912. [Online]. Available: https://aclanthology.org/2020.acl-main.442

[18] R. Rei, C. Stewart *et al.*, "COMET: A neural framework for MT evaluation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 2685–2702. [Online]. Available: https://aclanthology.org/2020.emnlp-main.213

[19] T. Kocmi, C. Federmann *et al.*, "To ship or not to ship: An extensive evaluation of automatic metrics for machine translation," *CoRR*, vol. abs/2107.10821, 2021. [Online]. Available: https://arxiv.org/abs/2107.10821

[20] M. Dehghani, Y. Tay *et al.*, "The benchmark lottery," *CoRR*, vol. abs/2107.07002, 2021. [Online]. Available: https://arxiv.org/abs/2107.07002

[21] P. Liang, R. Bommasani *et al.*, "Holistic evaluation of language models." [Online]. Available: http://arxiv.org/abs/2211.09110

[22] E. M. Bender, T. Gebru *et al.*, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 610–623. [Online]. Available: https://doi-org.proxy.wm.edu/10.1145/3442188.3445922

[23] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," pp. 1–13. [Online]. Available: http://arxiv.org/abs/1702.08608

[24] ——, "Considerations for Evaluation and Generalization in Interpretable Machine Learning," pp. 3–17, 2018.

[25] C. Molnar, G. Casalicchio, and B. Bischl, "Interpretable machine learning – a brief history, state-of-the-art and challenges," no. 1, pp. 1–15. [Online]. Available: http://arxiv.org/abs/2010.09337

[26] Z. C. Lipton, "The mythos of model interpretability." [Online]. Available: http://arxiv.org/abs/1606.03490

[27] F. Doshi-Velez and B. Kim, "Towards A Rigorous Science of Interpretable Machine Learning," *arXiv: Machine Learning*, no. Ml, pp. 1–13, 2017. [Online]. Available: http://arxiv.org/abs/1702.08608

[28] B. Schölkopf and J. von Kügelgen, "From statistical to causal learning," 4 2022. [Online]. Available: http://arxiv.org/abs/2204.00607

[29] J. Pearl, M. Glymour, and N. P.Jewell, *Causal Inference in Statistics, A Primer*, 2016.

[30] C. Molnar, *Interpretable Machine Learning*, 2019, https://christophm.github.io/interpretable-ml-book/.

[31] D. Lo, "Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps." [Online]. Available: http://arxiv.org/abs/2309.04142

[32] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[33] A. Vaswani, N. Shazeer *et al.*, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NeurIPs'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[34] H. Husain, H.-H. Wu *et al.*, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[35] "WM-SEMERU/CausalSE," Mar. 2024, original-date: 2024-03-06T23:51:39Z. [Online]. Available: https://github.com/WM-SEMERU/CausalSE

[36] T. Han, S. Srinivas, and H. Lakkaraju, "Which explanation should i choose? a function approximation perspective to characterizing post hoc explanations." [Online]. Available: http://arxiv.org/abs/2206.01254

[37] J. Pearl and D. Mackenzie, *The book of why: The New Science of Cause and Effect*, 2018.

[38] A. Hindle, E. T. Barr *et al.*, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 837–847.

[39] T. Nguyen, A. Nguyen, and H. Nguyen, "A statistical semantic language model for source code," in *ESEC/FSE 2013*, 2013.

[40] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[41] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–280. [Online]. Available: https://doi-org.proxy.wm.edu/10.1145/2635868.2635875

[42] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" *ESEC/FSE 2017: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

[43] R. M. Karampatsis and C. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *arXiv*, no. Lm, 2019.

[44] R. M. Karampatsis, H. Babii *et al.*, "Open-Vocabulary Models for Source Code (Extended Abstract)," *Proceedings - 2020 ACM/IEEE 42nd International Conference on Software Engineering: Companion, ICSE-Companion 2020*, pp. 294–295, 2020.

[45] B. Roziere, M.-A. Lachaux *et al.*, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 20 601–20 611.

[46] X. Hu, G. Li *et al.*, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 200–210. [Online]. Available: https://doi-org.proxy.wm.edu/10.1145/3196321.3196334

[47] M. Tufano, C. Watson *et al.*, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 832–837. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240732

[48] ——, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[49] M. Tufano, J. Pantiuchina *et al.*, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 25–36.

[50] M. White, M. Tufano *et al.*, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.

[51] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[52] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 858–868.

[53] Y. Hussain, Z. Huang *et al.*, "Deep transfer learning for source code modeling," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 30, pp. 649–668, 2020.

[54] Z. Feng, D. Guo *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[55] D. Guo, S. Ren *et al.*, "Graphcode{bert}: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[56] I. Tenney, D. Das, and E. Pavlick, "BERT rediscovers the classical NLP pipeline," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy:

Association for Computational Linguistics, Jul. 2019, pp. 4593–4601. [Online]. Available: https://aclanthology.org/P19-1452

[57] D. Dai, L. Dong *et al.*, "Knowledge neurons in pretrained transformers," *CoRR*, vol. abs/2104.08696, 2021. [Online]. Available: https://arxiv.org/abs/2104.08696

[58] A. Wang, A. Singh *et al.*, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=rJ4km2R5t7

[59] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for SQuAD," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 784–789. [Online]. Available: https://aclanthology.org/P18-2124

[60] S. Lu, D. Guo *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[61] Y. Liu, C. Tantithamthavorn *et al.*, "On the reliability and explainability of language models for program generation." [Online]. Available: http://arxiv.org/abs/2302.09587

[62] A. H. Mohammadkhani, C. Tantithamthavorn, and H. Hemmati, "Explainable AI for pre-trained code models: What do they learn? when they do not work?" [Online]. Available: http://arxiv.org/abs/2211.12821

[63] U. Khandelwal, H. He *et al.*, "Sharp nearby, fuzzy far away: How neural language models use context," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 284–294. [Online]. Available: https://aclanthology.org/P18-1027

[64] V. Prabhakaran, B. Hutchinson, and M. Mitchell, "Perturbation sensitivity analysis to detect unintended model biases," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5740–5745. [Online]. Available: https://aclanthology.org/D19-1578

[65] M. R. I. Rabin, N. D. Bui *et al.*, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106552, 2021.

[66] A. Karpathy, J. Johnson, and F. Li, "Visualizing and understanding recurrent networks," *CoRR*, vol. abs/1506.02078, 2015. [Online]. Available: http://arxiv.org/abs/1506.02078

[67] T. Wu, M. T. Ribeiro *et al.*, "Polyjuice: Generating counterfactuals for explaining, evaluating, and improving models," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Online: Association for Computational Linguistics, Aug. 2021, pp. 6707–6723. [Online]. Available: https://aclanthology.org/2021.acl-long.523

[68] J. Cito, I. Dillig *et al.*, "Counterfactual explanations for models of code," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 125–134. [Online]. Available: https://doi.org/10.1145/3510457.3513081

[69] Y. Hu and J. Tian, "Neuron dependency graphs: A causal abstraction of neural networks," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka *et al.*, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 9020–9040. [Online]. Available: https://proceedings.mlr.press/v162/hu22b.html

[70] A.-H. Karimi, K. Muandet *et al.*, "On the relationship between explanation and prediction: A causal view." [Online]. Available: http://arxiv.org/abs/2212.06925

[71] A. Sharma, E. Kiciman *et al.*, "DoWhy: A Python package for causal inference," https://github.com/microsoft/dowhy, 2019.

[72] M. Tufano, C. Watson *et al.*, "Learning How to Mutate Source Code from Bug-Fixes," *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pp. 301–312, 2019.

[73] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pp. 131–140, 2015.

[74] D. Rodriguez-Cardenas, D. N. Palacio *et al.*, "Benchmarking causal study to interpret large language models for source code," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 329–334.

[75] J. Pearl, "The seven tools of causal inference, with reflections on machine learning," vol. 62, no. 3, pp. 54–60.

[76] ——, "Causality: models, reasoning, and inference, by judea pearl, cambridge university press, 2000," *Econometric Theory*, vol. 19, no. 4, p. 675–685, 2003.

[77] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[78] Y. S. Abu-Mastafa, "Learning from data." [Online]. Available: http://work.caltech.edu/slides/slides01.pdf

[79] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," *Advances in Neural Information Processing Systems*, vol. 3, pp. 1137–1155, 2003.

[80] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Onward! OOPLSA 2019*, 2019, pp. 143–153. [Online]. Available: https://doi.org/10.1145/3359591.3359735

[81] K. Cho, B. van Merrienboer *et al.*, "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches," Oct. 2014, arXiv:1409.1259 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1409.1259

[82] C. Watson, N. Cooper *et al.*, "A systematic literature review on the use of deep learning in software engineering research," *CoRR*, vol. abs/2009.06520, 2020. [Online]. Available: https://arxiv.org/abs/2009.06520

[83] github, "Github," 2020. [Online]. Available: https://github.com/

[84] M. Abadi, A. Agarwal *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[85] A. Paszke, S. Gross *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle *et al.*, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[86] T. Wolf, L. Debut *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[87] R. M. Karampatsis, H. Babii *et al.*, "Big code != big vocabulary: Open-vocabulary models for source code," *Proceedings - International Conference on Software Engineering*, pp. 1073–1085, 2020.

[88] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2015.

[89] S. Lu, D. Guo *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[90] "Tree-sitter Introduction." [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[91] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[92] B. Kim, M. Wattenberg *et al.*, "Interpretability beyond feature attribution: Quantitative Testing with Concept Activation Vectors (TCAV)," *35th International Conference on Machine Learning, ICML 2018*, vol. 6, pp. 4186–4195, 2018.

[93] "Java Language Keywords (The Java™ Tutorials > Learning the Java Language > Language Basics)." [Online]. Available: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

[94] "tree-sitter/tree-sitter-python," Dec. 2023, original-date: 2016-12-15T06:22:25Z. [Online]. Available: https://github.com/tree-sitter/tree-sitter-python

[95] A. Sharma, V. Syrgkanis *et al.*, "DoWhy : Addressing Challenges in Expressing and Validating Causal Assumptions," 2021.

[96] Q. Dong, L. Li *et al.*, "A Survey on In-context Learning," Jun. 2023, arXiv:2301.00234 [cs]. [Online]. Available: http://arxiv.org/abs/2301.00234

[97] B. Ray, V. Hellendoorn *et al.*, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 428–439. [Online]. Available: https://doi.org/10.1145/2884781.2884848

[98] J. Pearl and E. Bareinboim, "Transportability of Causal and Statistical Relations: A Formal Approach."

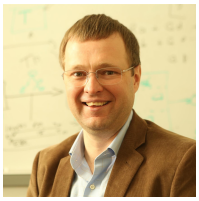
David N. Palacio is a Ph.D. Candidate in Computer Science at William & Mary, where he is a member of the SEMERU Research Group supervised by Dr. Denys Poshyvanyk. He received his MSc. in Computer Engineering at Universidad Nacional de Colombia (UNAL), Colombia, 2017. His research is concentrated on interpretable methods for deep learning code generators, specifically, on using causal inference to explain deep software models. His fields of interest lie in complexity science, neuroevolution, causal inference, and interpretable machine learning for the study and automation of software engineer processes. More information is available at https://danaderp.github.io/danaderp/.

Alejandro Velasco is a Ph.D. student in Computer Science at William & Mary, currently a member of SEMERU Research Group under the mentorship of Dr. Denys Poshyvanyk. He obtained his B.Sc and MSc. in Computer Engineering at Universidad Nacional de Colombia (UNAL). His research interests include software engineering, deep learning, interpretable machine learning, and software maintenance and evolution. More information is available at https://svelascodimate.github.io

Nathan Cooper is a nerd and received a B.S. degree in Software Engineering from the University of West Florida in 2018. He is currently a Ph.D. candidate in Computer Science at William & Mary under the mentorship of Dr. Denys Poshyvanyk and is a member of the Semeru Research group. He has research interests in Software Engineering, Machine / Deep Learning applications for Software Engineering, information retrieval, and question & answering applications for Software Engineering. He has published in the top peer-reviewed Software Engineering venues ICSE and MSR. He has also received the ACM SIGSOFT Distinguished paper award at ICSE'20. More information is available at https://nathancooper.io/#/.

Alvaro Rodriguez is a MSc. Candidate in Computer Engineering at Universidad Nacional de Colombia (UNAL). He received his B.S degree in Computer Engineering in 2018. His research interests include Artificial Intelligence for Software Engineering and Evolutionary Computation.

Kevin Moran Kevin Moran is an Assistant Professor of Computer Science and a member of the Cybersecurity & Privacy (CyberSP) Cluster at UCF. He directs the SAGE research group. He graduated with a B.A. in Physics from the College of the Holy Cross in 2013, an M.S. and Ph.D. from William & Mary in 2015 and 2018 respectively. His main research interest involves facilitating the processes of software engineering, security, and maintenance by building developer tools enhanced by machine learning. He has published over 30 papers at various software engineering and security conferences, and his research has been recognized with ACM SIGSOFT distinguished paper awards at ESEC/FSE 2019 and ICSE 2020, and a Best Paper Award at CODASPY'19. He was also recently recognized with the 2023 MOBILESoft Rising Star Award. More information is available at https://www.kpmoran.com/.



Denys Poshyvanyk is a Chancellor Professor of Computer Science at William & Mary. He received the MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in Computer Science from Wayne State University in 2008. He has served as a program co-chair for FSE'25, ASE'21, MobileSoft'19, ICSME'16, ICPC'13, WCRE'12 and WCRE'11. He currently serves as a Guest Editor-in-Chief of the AI-SE Continuous Special Section at the ACM Transactions on Software Engineering and Methodology. He served on the editorial board of IEEE Transactions on Software Engineering and ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering Journal (Springer), Journal of Software: Evolution and Process (Wiley) and Science of Computer Programming. His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering and software repository mining. His research papers received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13, CODAPSY'19 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16, ASE'17, ESEC/FSE'19 and ICSE'20. He also received the Most Influential Paper Awards at ICSME'16, ICPC'17, ICPC'20 and ICSME'21. He is a recipient of the NSF CAREER award (2013). He is an IEEE Fellow and an ACM distinguished member. More information is available at: http://www.cs.wm.edu/~denys/.