Counterfactual Explanations for Models of Code

Jürgen Cito [1], Işıl Dillig [2], Vijayaraghavan Murali[3], and Satish Chandra[4]

Abstract: Machine learning (ML) models play an increasingly prevalent role in many software engineering tasks. However, it can be difficult for developers to understand why the model came to a certain conclusion and how to act upon the model's prediction. Motivated by this problem, this talk explores counterfactual explanations for models of source code. Such counterfactual explanations constitute minimal changes to the source code under which the model "changes its mind". We integrate counterfactual explanation generation to models of source code in a real-world setting at Meta. We describe considerations that impact both the ability to find realistic and plausible counterfactual explanations, as well as the usefulness of such explanations to the developers that use the model. In a series of experiments, we investigate the efficacy of our approach on three different Large Language Models (LLMs) operating over source code.

Our original publication of the study has been presented at the International Conference on Software Engineering in the Software Engineering in Practice Track (ICSE-SEIP 2022).

Keywords: machine learning, explanations, source code models

1 Study Overview and Results Summary

Machine learning models, especially large language models, have become integral in various code-related tasks such as bug detection, auto-completion, and type inference. However, the adoption of these learning-based code analysis methods introduces challenges concerning unknown soundness and completeness characteristics. These models often exhibit high false positive rates, impeding trust from end users, notably software engineers who leverage these models in their development workflow. We introduce counterfactual explanations as a black-box explainability method within Meta for source code models. Our technique facilitates debugging for ML engineers and also offers individual prediction explanations crucial during the software development workflow (e.g., during code review) for software engineers who use these models.

Our counterfactual explanation approach introduces heuristic-driven methods to identify minimal perturbations in the source code input—specifically, within our performance prediction models. In contrast to existing work on robustness and adversarial examples for machine learning models, we introduce semantics-altering instead of semantics-preserving perturbations. By pinpointing mutations that flip model predictions, our technique highlights areas where the model derives its signal, empowering users to reason about prediction

1 TU Wien, Vienna, Austria, juergen.cito@tuwien.ac.at, https://orcid.org/0000-0001-8619-1271
2 UT Austin, Austin, TX, USA, isil@cs.utexas.edu, https://orcid.org/0000-0001-8006-1230
3 Meta, Menlo Park, CA, USA, vijaymurali@meta.com
4 Google, Mountain View, CA, USA, schandra@acm.org

doi:10.18420/sw2024_27

correctness. The comparison between the original source code (i.e., the input to the model) and the perturbed version serves as a contrastive explanation, facilitating deeper insights into model behavior and aiding in the validation of predictions. We experimentally evaluate our approach guided by the following research questions:

RQ1: How often do users find counterfactual explanations for models of code helpful? We conduct a formative study with 3 software engineers and research scientists from different stakeholder teams within Meta. We randomly sample 30 instances from the validation dataset used during training of these models. For each instance, we produce counterfactual explanations using our approach. We then ask the participants whether they found the explanation to be useful to understand the prediction. Our study participants found the explanations useful or very useful in 83.3% (25/30) of cases. Very useful explanations made up 30% (9/30) of cases. They only found 16.6% (5/30) of the explanations not useful (or were indifferent about them). When analyzing these cases together with rationales given by the participants, we found that this mostly had to do with explanations that introduced irrational perturbations.

RQ2: How do users utilize counterfactual explanations to discern between true-positive and false-positive predictions in models of code? We also ask the study participants to assess whether they think prediction is a true-positive or false-positive. They follow a think-aloud protocol in which they are encouraged to verbalize their thought process such that we can capture the rationale of their decision. We qualitatively analyze their responses and report on their experience with utilizing explanations. Our study participants were able to accurately determine the underlying ground truth in 86.6% (26/30) of cases. We noticed a distinction in how participants came to the conclusion on how to interpret their explanation. In true-positive instances, participants noted that the explanation guided them to parts of the code that were aligned with their mental model. More specifically, the explanation reinforced their hypothesis that had been induced through the prediction. (One participant did note that while this line of reasoning seems sound, it could be prone to confirmation bias.) In false-positive instances, participants noted that the strongest signal not to trust the prediction was the level of unreasonableness of the explanation. If the explanation pointed to irrelevant parts of the input or introduced an irrational perturbation, it was a sign that the prediction could probably not be trusted.

RQ3: Are counterfactual explanations for models of code aligned with human rationales provided by domain experts? We perform a case study based on an internal dataset where a team of domain experts for a taint propagation detection task had provided human rationales for code changes that we compare to our generated counterfactual explanations. We randomly sample a set of 30 inputs and filter out instances where we cannot find counterfactual explanations perturbing at most 5 tokens. We are eventually left with 11 instances for our analysis. Since our research questions involve human assessment of the counterfactual explanations, we use up to 5 explanations per input regardless of the number of explanations generated. Around 90% (10/11) of the instance explanations (at least one of the top 5 offered) aligned with the human rationale.