Counterfactual Explanations for Models of Code

Jürgen Cito
TU Wien and Facebook
Austria

Isil Dillig
UT Austin†
U.S.A.

Vijayaraghavan Murali
Facebook
U.S.A.

Satish Chandra
Facebook
U.S.A.

ABSTRACT

Machine learning (ML) models play an increasingly prevalent role in many software engineering tasks. However, because most models are now powered by opaque deep neural networks, 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 paper 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. We describe considerations that impact both the ability to find *realistic* and *plausible* counterfactual explanations, as well as the *usefulness* of such explanation to the user of the model. In a series of experiments we investigate the efficacy of our approach on three different models, each based on a BERT-like architecture operating over source code.

1 INTRODUCTION

With the rapid advances in deep learning over the last decade, complex machine learning models have increasingly made their way into software engineering. In particular, models based on deep neural networks are now routinely used in code-related tasks such as bug detection [29], auto-completion [3], type inference [28], code summarization [1, 48], and more.

While deep learning models are remarkably accurate and generally quite effective, one growing concern about their adoption is *interpretability*. In particular, because deep neural networks are extremely complex, it is often very difficult to understand why they make the predictions that they do. This issue of interpretability is a particularly relevant concern in software engineering since the outputs of machine learning models are routinely used to make predictions about code quality and non-functional properties of code. Even worse, because the predictions of these models are often consumed by software developers to change or improve an existing code base, it is particularly important that developers are able to understand *why* machine learning models make certain predictions.

To provide a more concrete illustration, consider a machine learning model that has been trained to detect whether some piece of code contains a security vulnerability [37]. If the developer does not understand *why* the model thinks a piece of code is vulnerable, they may fail to take a true vulnerability seriously. Furthermore, even when they do take the model prediction seriously, they might not be able to take remedial action if they cannot understand why something is considered to be a vulnerability.

This paper takes a step towards improving the interpretability of machine learning models used in code for *practical downstream applications*. Motivated by the shortcomings of existing methods like LIME [34], SHAP [17], and attention-based methods [44] in the context of source code, we develop a new technique for generating *counterfactual explanations* for models of code. Such explanations demonstrate how the model's prediction *would* have changed had the program been modified—or, *perturbed*—in a certain way. We believe that counterfactual explanations are particularly useful in the context of software engineering, as they reveal *not only which* parts of the code the model is paying attention to, but they also how to *change* the code so that the model makes a different prediction. This mode of so-called contrastive reasoning through counterfactuals is also aligned with how humans explain complex decisions [24].

In order to generate useful counterfactual explanations for code, we have conducted a formative study involving software engineers using different machine learning models at Facebook. Informed by the findings from this formative study, we then formalize what a counterfactual explanation entails in the context of code and present an algorithm for generating such explanations. At the heart of our technique lies a mechanism for modifying the program such that the resulting code is "natural". This ability is extremely important because perturbations that result in "unnatural" code can confuse the model by generating programs that come from a different distribution that the model has *not* been trained on. Thus, such modifications often reveal robustness problems in the model as opposed to yielding useful counterfactual explanations. Furthermore, counterfactual explanations that result in "unnatural code" are also not useful to developers because they do not provide meaningful insights on what a proper contrastive example looks like that flips the model prediction. We show an example of this in Section 2.

In this paper, we propose using *masked language models* (MLMs) as a way of generating meaningful counterfactual explanations. Our proposed technique "masks" a small set S of tokens in the original program and uses an MLM to predict a new set of tokens S' that can be used to replace S. This results in a "natural"-looking perturbation of the original program for which the model can make meaningful predictions, as the perturbed program comes from the same distribution that the model has been trained on. Our method then systematically builds a search space of perturbations and tests whether they result in a change of prediction for the relevant downstream task, and if so, returns it as a counterfactual explanation.

We conduct a series of experiments in the context of three machine learning models at Facebook that are applied to source code

† Work done at Facebook as visiting scientist.

diffs: performance regression prediction, testplan screenshot prediction, and taint propagation detection. We conduct a user study with 3 software engineers and research scientists at Facebook to see whether counterfactual explanations help them discern true positive from false positive predictions (they can in 86% of cases), and how useful they think these explanations are (83% of cases). We also assess how our explanations aligns with human rationales for prediction provided by domain experts. For our study sample, our generated counterfactual explanations correspond to human rationale in over 90%.

Contributions. This paper makes the following contributions:

- We present a method of generating counterfactual explanations for ML models that predict certain properties of code or code changes. These properties are typically checked by humans during code review, so if ML models were to be used in place of humans, an explanation of the model predictions is necessary.
- We present desiderata for counterfactual explanations in the context of source code that are *useful* to the end user who has to consume the predictions of the ML models, as well as the counterfactual explanation offered along with it.
- We have applied counterfactual explanations for predictions generated by models for three distinct real-world code quality machine learning models. We show empirical results on the usefulness of these explanations to the users of these models. We also show, quantitatively, that counterfactual explanations have a better correspondence with human-generated explanation, compared to a previously presented perturbation-based explanation technique [23].

The paper is organized as follows. Section 2 gives the context of this work in terms of the model family we use, as well as the tasks for which these models are trained. We walk the reader through examples of counterfactual explanations. Section 3 describes informally the requirements from useful counterfactual explanations; these are based on interactions with prospective users. Section 4.1 tackles the problem of explanation generation formally, and presents our core algorithm. Section 5 describes our experimental setting and the results.

2 CONTEXT OF THIS WORK

BERT-based models of code. Deep-learning architectures drawn from NLP have become commonplace in software engineering [1, 2, 48]. Among these, recently large-scale language models such as BERT [10] have shown state-of-the-art performance in NLP tasks, and correspondingly, the software engineering community has adopted these architectures for code-related purposes, e.g. in CodeBERT [14]. Since our models fall in this category, we give a very brief background here.

The idea behind BERT (and CodeBERT) is to *pre*-train a sequence model using self-supervision. This is done by giving a neural network the training task of predicting one or more tokens in the sequence that are purposely masked out. The network's training objective is to predict the masked-out tokens correctly. This is known popularly as a *masked language model*, or MLM. The purpose of the MLM pre-training is to learn a generic embedding of a

```
1    private async function storeAndDisplayDialog(
2       SomeContext $vc,
3       SomeContent $content,
4  -  ): Awaitable<SomethingStoreHandle> {
5  +  ): Awaitable<SomeUIElement> {
6  -   $store_handle = await SomethingStore::genStoreHandle($vc);
7  +   $store_handle = await SomethingStore::genHandle($vc);
8  +   ... other code ...
9  +   $store_success = await $store_handle->store(
10 +     $store_handle,
11 +     $content,
12 +   );
13 -   return $store_handle;
14 +   return await $store_success->genUIElementToRender();
15    }
```

Listing 1: A code change that we use as running example. The added lines are marked with a "+" and the deleted lines with a "-"

token sequence; as in LSTMs, the hidden state after the last token in a sequence is taken to be the embedding of the entire sequence.

While pre-training is generic, a second step, called *fine tuning*, is applied to customize the network to a specific task at hand. Here one uses a supervised dataset (i.e., a language fragment as well as a label that a human associates with the fragment), to become a classifier for a specific task. For instance, in a sentiment analysis application, a (binary) label could be whether the token sequence reflects a positive sentiment.

Automated code review. At companies small and large (including Facebook), there is considerable interest in more thorough and more automated code review with the hope of giving feedback to developers early. Whenever a developer submits a code commit (aka a "diff"), a series of automated analyses are carried out in addition to a human code review. These automated analyses try to find both stylistic inconsistencies, and to the extent possible, functional and non-functional issues in code.

Complementing traditional code analyses are ML-based analyses for a variety of purposes, and are increasingly being used in software engineering tasks in industrial practice [4]. Here we illustrate three of these analyses (models), using a common small example. Our models are based on BERT, described above.

Listing 1 represents a code change that could be made in a diff (for simplicity, we omit other features such as the commit message). It implements a function that stores a piece of content in a data store and returns a UI dialog element to signify that the operation has been successful. By convention, the added lines are shown with a "+" and the deleted lines are shown with a "-". The unchanged lines are shown for context, which is also important for the model to train well.

We now discuss the three downstream tasks while referring to Listing 1. We illustrate several aspects of counterfactual explanations in the first of these tasks, and keep the other two brief.

Performance Prediction. In the diff shown above, replacing line 6 (an optimized operation for store handles) with line 7 (an

```
-   $store_handle = await SomethingStore::genStoreHandle($vc);
+   $store_handle = await SomethingStore::genHandle($vc);
                          SomethingStore::genSimple($vc)
+   ... other code ...
```

Listing 2: Counterfactual explanation of the model's prediction that the code change in Listing 1 will cause a performance regression. The explanation says that if we replace the red part with green, the model will no longer make that prediction. We refer to such replacements as *perturbations*.

unoptimized version for generic handles) could cause a performance problem. Lines 8-11 are also added, which could independently cause performance problems. It is not known at code review time whether or not this diff will cause a performance regression for sure, because the code has not been deployed in a real environment yet.

In support of an intelligent, automated code review, a BERT-based predictive model (trained on a curated set of past performance regressions) provides an educated guess at whether this diff is a likely performance regression. In this example, suppose that the model's prediction is "yes". In the current status quo, the developer has to start a complex investigation process that involves hypothesis building, statically inspecting potential culprits, and expensive testing with delta-debugging that involves benchmarks [5]. If it then turns out that the machine learning prediction was a false positive, it could lead to frustration, as is well-known from existing experiences with deploying any kind of uncertain program analysis [7, 38].

To help the developer gain better insight about the model's prediction, our system automatically generates a counterfactual explanation, as shown in Listing 2. This counterfactual states that had `genStoreHandle` been replaced with `genSimple` instead of `genHandle`, then the model would *not* have predicted a performance regression. Such feedback is useful to the developer, as it highlights that `genHandle` is the likely culprit and allows them to ignore changes to the other parts of the code. Thus, the part elided as "... other code ..." can be ignored by the developer when trying to figure out how to address this automated feedback.

Multiple Blame Points. In many cases, the model's prediction does not depend on a single line of the diff, of even a consecutive fragment of code. Rather, there may be multiple parts of the diff that cause the model to make a certain prediction. For instance, Listing 3 shows a scenario where the counterfactual involves multiple parts of the diff. For example, if the procedure `store` relies on an optimization performed by `genStoreHandle`, then it would make sense that the model's prediction is informed by *both* of these function calls. Observe that techniques based on delta debugging [31] might not be able to pinpoint blame points in non-contiguous pieces of code, for example, due to the "...other code..." in this example.

Preserving in-distribution. As illustrated by the previous two examples, our method produces counterfactuals that look "natural" — that is, the replacements proposed by our technique *could* have

```
-   $store_handle = await SomethingStore::genStoreHandle($vc);
+   $store_handle = await SomethingStore::genHandle($vc);
                          SomethingStore::genSimple($vc)
+   ... other code ...
+   $store_success = await $store_handle->store(
                           $store_handle->probe
+     $store_handle,
+     $content,
+   );
```

Listing 3: A counterfactual with two changes that must be made together for the model to change its prediction.

plausibly occurred in the diff. This is an important aspect of this work, as candidate explanations that are implausible often cause the model to "change its mind" due to robustness issues in the model. For example, consider the candidate explanation shown in Listing 4. In this case, replacing the return value by `await 5` yields a non-sensical code snippet that comes from a different distribution than the one our model has been trained on. Thus, the model is *not* expected to make reliable predictions for such code and can *erroneously* predict that the diff (after perturbation) does not have a performance regression. In this case, the out-of-distribution nature of the perturbation results in an "explanation" that does not blame the true culprit. We want to avoid such out-of-distribution perturbations.

Testplan Screenshot Prediction. So far, we highlighted salient aspects of our approach in the context of a model for performance prediction. However, such counterfactuals are more broadly useful across different downstream tasks. Next, we illustrate how explanations can be useful in the context of *testplan screenshot prediction.*

When submitting a diff for review, the developer also has to provide a testplan that details how exactly that particular diff should be tested. The testplan usually includes instructions on how to run automated tests that exercise the code change as well as the results of those tests. A testplan that has to do with UI functionality should also include a screenshot; otherwise it is not a very good quality test plan. Calling out poor quality test plans is another check that code review is expected to do.

At Facebook, we use a predictive model for testplan quality that indicates whether a testplan should contain a screenshot of the introduced functionality. Such a model is trained over a curated dataset of diffs with manually-specified binary labels. For instance,

```
+   ... other code ...
-   return $store_handle;
+   return await $store_success->genUIElementToRender();
                 5
```

Listing 4: A counterfactual based on an out-of-distribution code change.

```
+   $store_handle = await SomethingStore::genHandle($vc);
+   ... other code ...
+   $store_success = await $store_handle->store(
+     $store_handle,
+     $content,
+   );
-   return $store_handle;
+   return await $store_success-> genUIElementToRender ();
                                  getValue
    }
```

Listing 5: A counterfactual for testplan screenshot prediction model

```
   private async function storeAndDisplayDialog(
      SomeContext $vc,
      SomeContent $content ,
       $count
      ...
+   ... other code ...
+   $store_success = await $store_handle->store(
+     $store_handle,
+      $content ,
        $count
+   );
```

Listing 6: A counterfactual for taint propagation detection model

in our running example, the model might predict that the testplan should require a screenshot, possibly because the functionality that is being changed involves displaying and rendering UI elements (line 14 in the original example). Our proposed method also helps justify such testplan-related predictions by producing the counterfactual explanation shown in Listing 5. With such an explanation, the developer gets a clear indication that the prediction is related to the call to `genUIElementToRender`, absolving her from the burden of manually inspecting irrelevant parts of a (potentially very large) diff.

Taint Propagation Detection. Next, we discuss the usefulness of counterfactuals in the context of a model that predicts whether a diff introduces data flow that may propagate tainted information. Going back to our running example, a model trained for this task might flag the diff as suspect because the added code stores some content in a data store (line 9-12 in the original code example). However, given that the diff makes many other changes, it may not be apparent to the developer why the model makes this prediction. Our technique could also be useful in this context by producing the counterfactual shown in Listing 6. This explanation informs the developer that the model based its decision on the use of the variable name `content`.

3 DESIDERATA FOR COUNTERFACTUAL EXPLANATIONS FOR CODE

The series of examples presented in Section 2 hopefully convey the idea that the results of ML models can be made more useful, persuasive, and actionable if accompanied by a counterfactual explanation. However, when designing explanatory systems, there are several criteria and pitfalls to consider, some of which have been discussed in the literature [25]. Additionally, since our method targets models of code, we need to take into account specific considerations of software engineers submitting their changes for code review. For this purpose, we conduct a formative study with three software engineers who are domain experts in the downstream tasks that we investigate in this paper. Our goal is to understand the desiderata of explanation systems as part of the software engineering process.

In this study, we present users with the predictions of the model for specific tasks and ask them to provide human rationale for each prediction. We also show them different counterfactual explanations and ask them for feedback. Overall, we discuss around 10 predictions with each participant. At this point of our research, we were mostly interested in open-ended, qualitative feedback to understand the frustrations and confusions of software engineers and assess overall room for improvement. This study informed our design decisions that in turn influenced our problem formulation, implementation, and experiments. In what follows, we briefly summarize our findings, many of which are consistent with similar findings in different (non-code related) contexts.

Plausibility and actionability. Plausibility and actionability are regularly mentioned terms in the literature on counterfactual explanations [27, 39] and algorithmic recourse [18, 33]. Plausibility means that we should not generate counterfactuals that are neither realistic nor believably part of the original input. Similarly to plausibility constraints in other domains, a code counterfactual is plausible if the code retains its naturalness as a result of the modification. We showed Listing 2 as a plausible counterfactual, and Listing 4 as one that is *im*plausible.

Another concern frequently brought up in our study is that of *actionability*, i.e., does the explanation show actionable recourse options? Pragmatically, we could take into account actionability constraints in our problem formulation and only allow counterfactuals that modify actionable features of the input. For example, in the case of code diffs, we could restrict modifications to be only applied to the added lines in the diff. However, after more discussions with our formative study participants, we realized that deleted lines and context matter just as much to the model — otherwise we could have trained the model only on added lines in the first place, making it a lot more myopic and consequently imprecise. Therefore we decided to *not* restrict which parts of the input to perturb, with the rationale that the model designer can always later apply a filter to the generated counterfactuals in a post-processing step. Listing 6 showed a counterfactual in which one of the perturbations is outside the lines changed in the diff.

Consistency. Another important finding from our formative study is that counterfactuals that modify *different occurrences* of the *same* identifier are neither particularly useful nor plausible. For example, consider a diff involving a class called “SomethingStore".

If the modification renames this class in the import statement but not elsewhere in the code, the modified code neither makes sense nor leads to useful counterfactuals. Based on this finding, we decided to restrict code modifications we consider when generating counterfactuals to those that are *consistent*. That is, our approach only allows input modifications that change the same identifier consistently throughout the code diff.

Proximity. Multiple formative study participants brought up that the distance between multiple perturbations matters to the comprehensibility of explanations. Perturbations that were part of the explanation of one instance, but that were "too spread out" were confusing and not really helpful (e.g., an explanation that added perturbations in the beginning of the diff and at the far end of the diff). We use this finding to design a heuristic to select between multiple candidate counterfactuals. In particular, when our method finds multiple explanations, we prefer those where the modifications are not too far apart.

4 COUNTERFACTUAL EXPLANATIONS FOR MODELS OF CODE

In this section, we formalize the problem of generating counterfactual explanations for code and describe our algorithm.

4.1 Problem Formulation

Our problem formulation utilizes the concept of *perturbation* which is a transformation π that can be applied to a program. Given a program P and perturbation π, we write $\pi(P)$ to denote the resulting program obtained by applying π to P.

Given a machine learning model $\mathcal{M}$ trained on set $S \sim \mathcal{D}$ for some task t and a program $P \sim \mathcal{D}$, a *counterfactual explanation* is a perturbation π of P satisfying the following criteria:

(1) The model makes different predictions for the original program and the perturbed version, i.e., $\mathcal{M}(P) \neq \mathcal{M}(\pi(P))$
(2) The ground truth for P and $\pi(P)$ are different for task t, i.e., $\mathcal{G}(P) \neq \mathcal{G}(\pi(P))$
(3) The perturbed program $\pi(P)$ is "natural", meaning that it comes from the same distribution as the original program P (i.e., $\pi(P) \sim \mathcal{D}$)

Note that our definition of counterfactual explanation is quite similar to *adversarial examples* [32, 47] in that both are defined as small perturbations π that cause the model to "change its mind" (i.e., condition (1) from above). However, there are two key differences between counterfactual explanations and adversarial examples that are outlined in conditions (2) and (3) in the above definition. First, adversarial examples are inputs that are designed to intentionally fool the model – that is, from a human's perspective, the prediction result for P and $\pi(P)$ should be exactly the same. In contrast, a perturbation only makes sense as a counterfactual explanation if P and $\pi(P)$ are semantically different for task t from the user's perspective. That is, the ground truth prediction for P and $\pi(P)$, denoted by $\mathcal{G}(P)$ and $\mathcal{G}(\pi(P))$ resp., should be different, as stipulated in condition (2) of our definition.

In addition, a second key difference between counterfactual explanations and adversarial examples is whether they are "natural". In particular, since adversarial examples are specifically crafted to fool the model, there is no requirement that they are drawn from the same distribution that the model has been trained on. On the other hand, for a perturbation π to make sense as an explanation of the model's behavior as opposed to a *robustness* issue, the perturbed program needs to be drawn from the same distribution as the training data, as stipulated in condition (3).

We illustrate the differences between counterfactuals and adversarial examples in Figure 1 in the context of a classification problem with four classes, depicted by different colors. As we go out from the center, the data gets increasingly out of distribution. If we want to show an adversarial example, we try a perturbation that pulls the input out of distribution to make the model mispredict, but we want to retain the same ground truth. If we want to find a counterfactual, we try to perturb such that the ground truth label does change (as well as the model prediction) while staying within distribution.

Discussion. While our problem formulation stipulates that a counterfactual should flip the *ground truth* prediction, our algorithm for generating counterfactuals can never enforce condition (2), as we do not have access to the ground truth. In fact, an explanation generated by *any* technique (including ours) could violate condition (2). However, *assuming* that the model is quite accurate for inputs drawn from the target distribution (i.e., $P \sim \mathcal{D}$), we have $\mathcal{M}(P) \approx \mathcal{G}(P)$, so conditions (1) and (2) become interchangeable. Thus, for models with high accuracy, an algorithm that ensures conditions (1) and (3) is unlikely to generate counterfactual explanations that violate condition (2).

Additionally, the contrapositive of the above observation suggests a way of using counterfactuals to diagnose a model's mispredictions. In particular, suppose that an algorithm produces a counterfactual π where the user thinks $\mathcal{G}(P)$ should be the same as $\mathcal{G}(\pi(P))$. In this case, assuming that the counterfactual generation algorithm enforces conditions (1) and (3), the above assumption that $\mathcal{M}(P) \approx \mathcal{G}(P)$ is violated. Thus, in practice, we have observed a strong correlation between "non-sensical" counterfactuals and model mispredictions. Said differently, if the algorithm produces a counterfactual that does not make sense to users, this can serve as a strong hint that the model's prediction is wrong.

As an example of this way of diagnosing model mispredictions, revisit Listing 1. Suppose, hypothetically, that Listing 5 were offered as the (only) counterfactual for the performance model's prediction of a potential regression. Suppose further that a human judge immediately sees that the unperturbed and the perturbed code should have the same ground truth, because the UI-related call has nothing to do with performance. In that case, we can conclude the model's prediction of a performance regression was a misprediction.

4.2 MLM-Based Perturbations for Code

Our problem formulation does not specify the exact nature of perturbations; however, a counterfactual generation algorithm needs to consider a universe of possible modifications to a given code snippet. To ensure that our counterfactual explanations satisfy the naturalness criterion (i.e., condition (3)), we propose generating perturbations using *masked language modeling (MLM)* [10]. At a high level, the idea is to replace each token with a blank ("mask") and use MLM to come up with a plausible replacement for the

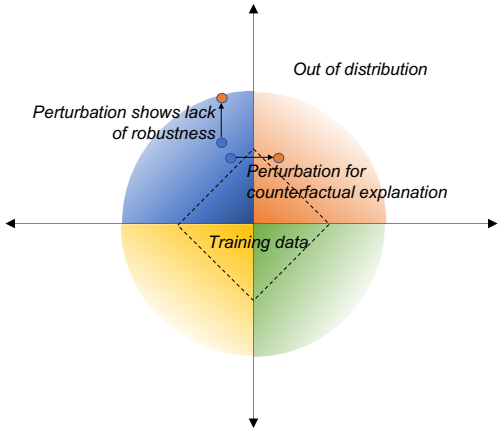

Figure 1: The four colors show four different classes. As we go out from the center, the data gets more out of distribution. Two perturbations are shown: one that exposes a robustness issue, and the other that is useful for counterfactual explanation. The colors of the small circles show the class that the model predicts.

original token. In more detail, our approach first tokenizes the input program P and produces an indexed sequential representation $\mathcal{T}(P) = \{p_i\}_{i=1}^{N}$. Then, given a specific token p_j that we want to perturb, we produce a new sequence $\mathcal{T}(P') = \langle p_1, \ldots, p_{j-1}, \texttt{[MASK]}, p_{j+1}, \ldots p_N \rangle$ and use MLM to produce a probability distribution for possible instantiations of [MASK]. Since MLM leverages all other tokens in the sequence to make predictions, it can produce more natural perturbations than other (more traditional) language models that only leverage preceding tokens in the sequence.

4.3 Algorithm for Generating Counterfactuals

We now describe our algorithm for generating counterfactual explanations. Our algorithm takes as input a program P and a machine learning model $\mathcal{M}$ and returns a set of counterfactual explanations E. Each explanation $e \in E$ is a mapping from a token $t_i \in P$ to a new token t_i' such that replacing every t_i with t_i' causes the model $\mathcal{M}$ to change its prediction.

At a high level, our algorithm explores explanations of increasing size, where the size of an explanation is the number of replaced tokens. Specifically, Algorithm 1 maintains a set (called *explore*) of *candidate replacements*, where each candidate is a set of tokens to be replaced. Then, in each iteration, it grows the most promising candidate replacement by one additional token (Line 6) to generate a new set of candidate replacements called *new_candidates* (Line 7). Then, for each candidate c in this set, Algorithm 2 invokes a procedure called FindCounterfactual to test whether there exists a counterfactual explanation whose domain is exactly c. If so, all explanations returned by FindCounterfactual are added to set *explanations* (line 11). On the other hand, if there is no counterfactual whose domain is c, we add c to set *explore* so that the domain of the explanation is grown in subsequent iterations (line 13). Note

Algorithm 1: GenerateCounterfactualExplanations

Input : Input program P, Model $\mathcal{M}$: $\mathcal{D} \mapsto \{T, F\} \times \mathcal{R}$ returning a classification and a score, iteration count ITER
Output : List of counterfactuals that lead to prediction changes

1 $explore = \emptyset$
2 $tokens = \{t_i \mid t_i \in \mathcal{T}(P)\}$
3 explanations = $\emptyset$
4 **for** _ *in* ITER+*1* **do**
5 $\quad c_{best}$ = Choose($explore$)
6 $\quad$ new_candidates = $\{(c_{best}, t_i) \mid t_i \in (tokens \setminus explanations)\}$
7 $\quad$ **for** c *in new_candidates* **do**
8 $\quad\quad E$ = FindCounterfactual($P, \mathcal{M}, c$)
9 $\quad\quad$ **if** $E \neq \emptyset$ **then**
10 $\quad\quad\quad$ explanations.add(E)
11 $\quad\quad$ **else**
12 $\quad\quad\quad$ explore.add(c)
13 $\quad$ return explanations

Algorithm 2: FindCounterfactual

Input : Input program P, Model $\mathcal{M}$, Token or token combination to perturb $\langle p_1, \ldots, p_k \rangle$ where $p_i \in \mathcal{T}(P)$
Output : Whether counterfactual exists (boolean), Score after perturbation, Counterfactual P' (input after perturbation)

1 $E = \emptyset$
2 $S = \mathsf{MLM}(P, \langle p_1, \ldots, p_k \rangle)$
3 **for** $\langle p_1', \ldots, p_k' \rangle \in S$ **do**
4 $\quad P' = P[p_1'/p_1, \ldots, p_k'/p_k]$
5 $\quad$ **if** $\mathcal{M}(P) \neq \mathcal{M}(P')$ **then**
6 $\quad\quad E = E \cup [p_1 \mapsto p_1', \ldots, p_k \mapsto p_k']$
7 return E

that, since smaller counterfactual explanations are always preferable, there is no point in adding c to *explore* if FindCounterfactual returns a non-empty set.

Algorithm 2 shows the implementation of the FindCounterfactual procedure. At a high level, this method uses the masked language modeling technique described in Section 4.2 to find plausible replacements for *all* tokens in c (line 4). That is, given a set of tokens $p_1, \ldots, p_n$, we replace each p_i in this set with a mask and ask the language model to generate a set S of plausible replacements for the tokens $p_1, \ldots, p_n$. Then, for each replacament $\langle p_1', \ldots, p_n' \rangle$ in set S, we check whether substituting each p_i with p_i' *actually* causes the model $\mathcal{M}$ to change its prediction (line 7). If so, we then add the mapping $[p_1 \mapsto p_1', \ldots, p_k \mapsto p_k']$ to the set of generated counterfactual explanations (line 8).

5 EXPERIMENTS

To explore the effectiveness of our proposed approach, we perform a series of experiments based on models and tasks in the context of a large deployment scenario at Facebook. In what follows, we provide a brief description of the underlying setting, including machine learning models and downstream tasks, and then describe the experiment methodology to answer our research questions.

5.1 Setting

The models and tasks we employ in our experiments take as input a diff (e.g., akin to pull-requests in open source projects) and assign a task-dependent probability about the code changes (context, additions, deletions) of that diff. Each model has been pre-trained on millions of diffs at Facebook (with no particular downstream task in mind) and then fine-tuned for specific tasks.

To evaluate our approach, we focus on the downstream tasks described in Section 2, namely performance regression, test plan quality (whether a screenshot is warranted), and detecting whether there is a taint propagation issue.

5.2 Research Questions and Methodology

The goal of our experimental evaluation is to address the following research questions.

RQ1: How often do users find counterfactual explanations for models of code helpful? We conduct a study with 3 software engineers and research scientists (we will collectively call them "users" going forward) from different stakeholder teams within Facebook. 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.

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.

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 the *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 diffs and filter out instances where we cannot find counterfactual explanations perturbing at most 5 tokens[1]. 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 diff regardless of the number of explanations generated. The ones shown to users are chosen based on the model's prediction likelihood for the counterfactual. That is, the lower the probability of the positive label from the model, the bigger was the influence of the perturbation, and the better is the explanation.

[1]Anecdotally, we can say that perturbing any more than 5 tokens will make an explanation practically useless

5.3 Results

RQ1: Usefulness. Our users 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. There are several kinds of perturbations that were considered irrational in the study. For instance, it was noted that a perturbation did not make sense because a method invocation on a class was perturbed into method that does not exist as part of that class. A particular explanation was considered not very useful because the counterfactual introduced *too many* perturbations. Even though we aim for sparse explanations, this can happen due to our goal of *consistency* (see Section 3.) If we perturb one particular token (e.g., a type) and it occurs multiple times in the input, we perturb all instances of it. Perturbations that occurred in non-actionable parts of the input were also initially dismissed and deemed not useful. However, upon reflection, users noted that it is still useful for models where contextual cues (i.e., where a change occurs) is sometimes more important than the change itself. This reinforces the perspective on feasibility and actionability we observed in our formative study (Section 3).

RQ2: Discerning TP vs FP. In addition to eliciting usefulness signals, we wanted to observe how our explanations can help discerning whether a prediction is a true-positive or a false-positive. Our users 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.

Overall, our qualitative analysis showed that the counterfactual explanations provided our study participants with intuition and confidence in understanding exactly where the model is picking up its signal. Whether that signal was reasonable or not helped them decide to trust or discard the prediction.

RQ3: Correspondence with Human Rationale. We investigate to what extent generated explanations align with rationale provided by human experts. To determine this alignment, two of the authors compare the generated explanations to the rationales. Specifically, we map the natural language description to tokens of the source

Table 1: Experiment overview summarizing the quantitative results of RQ1 and 2

User Study	TP/FP Guess	Usefulness
Overall	86.6% Accuracy	83.3% useful / 16.6% not
Performance	85% Accuracy	85% useful / 15% not
Testplan Screenshot	90% Accuracy	80% useful / 20% not

code. We consider an explanation to match the rationale if their token attributions overlap at least 50%. In our sample, ~90% (10/11) of the instance explanations (at least one of the top 5 offered) aligned with the human rationale. Even in the one case we deemed to not align entirely, while the top 5 explanations by our scoring did not align with the human rationale, two among the top 10 did.

In addition to analyzing alignment, we wanted to see how our approach compares to a representative baseline from the literature. Thus, we also generate explanations that are based on occlusion (i.e., token removal instead of replacement) using the SEDC algorithm [23], a greedy approach that successively removes token combinations from the input until the prediction flips. Occlusion has been used as a simple, yet effective, baseline in related work in NLP [12, 13, 20].

Table 2 provides a quantitative overview on the results of our approach (CFEX) with the baseline (SEDC) including: number of explanations generated (**#Exp**), size of the diff measured in number of tokens (**Size**), and summary statistics on the size of explanations in number of tokens (**Avg, Min, Max**). We compare both approaches by determining a winner (**Wins**) based on the following criteria: First, the approach needs to produce an explanation that aligns with the human rationale. If both approaches generate aligning explanations, the shorter explanation wins (less number of attributive tokens). If the explanation size is the same, it is considered a tie. In our results, we observe wins or ties in ~90% (10/11) for CFEX and ~19% (2/11) for SEDC.

This indicates that CEFX produces explanations that are more aligned with human explanation while also producing shorter explanations than the baseline technique does. Overall, we were able to see that counterfactual explanations generated through perturbations proposed by MLM are *useful*, can help users discern false- and true-positives, and seem likely to align human rationale.

Table 2: Overview of generated counterfactual explanations (CFEX) and explanations generated by token removal (SEDC)

		# Exp	Size	Avg	Min	Max	Wins
1	CFEX	1	234	4	4	4	**x**
	SEDC	-					
2	CFEX	7	143	5	5	5	**x**
	SEDC	2	143	4.5	4	5	**x**
3	CFEX	23	274	3.83	3	4	
	SEDC	13	274	4	4	4	**x**
4	CFEX	11	307	2.91	2	3	**x**
	SEDC	11	307	3.64	1	4	
5	CFEX	8	48	2.75	2	3	**x**
	SEDC	2	48	3	3	3	
6	CFEX	75	315	4	4	4	**x**
	SEDC	-					
7	CFEX	13	96	3.62	3	4	**x**
	SEDC	-					
8	CFEX	27	219	2.96	2	3	**x**
	SEDC	22	219	3.95	3	4	
9	CFEX	88	292	2.77	2	3	**x**
	SEDC	30	292	3	3	3	
10	CFEX	124	301	2.95	2	3	**x**
	SEDC	-					
11	CFEX	15	117	2.27	1	3	**x**
	SEDC	22	117	2.23	1	3	

6 DISCUSSION

Deployment Considerations. While most research on explanatory systems discusses how to provide faithful explanations to predictions, they implicitly require that prediction to be a true-positive. Most research papers discussing explanations do not even consider false-positives in their evaluation scenarios. However, in the context of deploying explanations as part of large-scale machine learning deployments, considering false-positives is vital to the success of the model. This is especially true in the context of learned models for code quality, where developers are rightly disillusioned with the lack of certainty and soundness of developer tools, leading them to chase ghosts in the light of wrong predictions. That is why we put an emphasis on evaluating our approach not only on a subjective consideration of "usefulness", but also whether (and how) an explanation instills trust in both correct and wrong predictions. Another aspect we want to consider for deployment is enabling interactive use with our generated counterfactuals to allow for exploration of decision boundaries (Figure 1).

Interactive Counterfactual Exploration. Using generative models to produce counterfactuals incurs non-negligeble runtime costs that ranges in the order of 30 seconds for smaller diffs to up to 10 minutes on the very large side of the diff size spectrum. In our current setting (Section 2), we can allow ourselves to process counterfactual generation offline and attach it as part of the code review. However, we envision counterfactual explanations to become part of an interactive exploration process. While part of making such an interactive experience possible is certainly performance engineering, we may have to think of other creative ways to make counterfactual generation with language models more instant. A possibility we want to explore is leveraging the traces of token replacements produced in the offline search to learn a neural model that mimics the MLM filling with much faster inference times.

Limitations and Scope. We developed and applied the proposed approach in the context of workflows and tools within Facebook. While nuances of the internal workflow may have specific peculiarities, we generally think that the mode of submitting commits for code review is widely established both in wider industry and open-source projects. Also, while we run our experiments on large transformer, i.e. BERT-like, models, our approach and algorithm are model agnostic and only require repeated access to label and score

information that most statistical learning models provide. Nevertheless, other kinds of models could exhibit other characteristics when it comes to the kinds of counterfactuals that are produced.

7 RELATED WORK

There has been significant recent interest in improving the interpretability of machine learning models. Efforts in this space can be broadly classified into two categories, namely *local* and *global* interpretability. Local explanability techniques aim to provide justification for predictions on a specific input [17, 34, 35]. On the other hand, global explanability techniques try to explain the behavior of the *whole* model, for instance by constructing a simpler surrogate model that emulates the original model [11, 19]. Since our goal in this work is to provide justifications for individual predictions, this work falls under *local explanation* techniques. In what follows, we give a more detailed overview of relevant work in this space.

White-box techniques for local explanations. Techniques for generating local explanations can be further classified as being either *white-box* or *black-box*. As their name indicates, white-box techniques are customized to specific ML models and exploit the internals of model when generating explanations. A common approach to white-box interpretability of deep learning is through so-called *attention mechanisms* where the idea is to use weights of attention layers inside the network to determine the importance of each input token [9, 15, 21, 43]. However, recent work has shown that different weights can lead to the same prediction and has called into question whether attention-based mechanisms are actually meaningful as explanations [16]. Similarly, other work has shown that it is possible to systematically manipulate attention while still retaining the same prediction [30].

Another popular white-box technique for generating local explanations is *integrated gradients* [40]. The high-level idea behind this method is to create interpolations of the input and evaluate the model on those interpolated inputs. Unfortunately, unlike image data, code does not lend itself easily to such interpolation. For instance, there is no meaningful token that can be obtained by combining the embedding of a pad token and that of a keyword token. In fact, we initially investigated using integrated gradients for generating local explanations for models of code, but we were not successful in generating useful explanations. This is, presumably, because unlike images, there is no natural interpolation between a zero token and the token in the input text.

Perturbation-based explanation mechanisms. Another common approach for generating local explanations is through perturbation-based mechanisms such as LIME [34] and SHAP [17]. These techniques remove or replace a subset of the features in the input space and track the score differential of the model's prediction. The aggregated score difference over many samples is then attributed to features involved in the perturbation. While these techniques can be applied to multiple types of ML models, they do not generate counterfactual explanations. Instead, they highlight input features that are most important for the model's prediction.

In the realm of understanding for source code models, recent work uses delta-debugging techniques to reduce a program to a set of statements that is minimal and still preserves the initial model prediction [31, 42]. The intuition here is that essentially the remaining statements are the important signal being picked up by the model. Another effort empirically show that attention scores in neural networks are highly correlated with code perturbations (statement removals in the source code input) [6]. However, these works have not investigated (or considered) the effects of token removal that may lead to out-of-distribution inputs. These in turn can lead to unreliable prediction outcomes [26] and misleading attention scores [16].

Counterfactual explanations. Another common approach to local interpretability is through the generation of counterfactuals. These techniques are very related to the previously discussed perturbation-based mechanisms, but they come with a stronger guarantee, namely that the perturbations are *guaranteed* to change the model's prediction. The generation of counterfactual explanations has received significant attention in the NLP community [22, 23, 36, 45, 46]. In the simplest case, these counterfactuals can be generated by deleting words from the input text [23] or via rewrite-rules such as adding negations or shuffling words [45]. Similar to our goal of generating natural-looking programs, these techniques also aim to generate *fluent* text that is grammatically correct. Among the counterfactual generation techniques, perhaps the most relevant to ours is the *minimal contrastive editing (MiCE)* technique of [36]. Specifically, they train an *editor* to predict words to edit and use a generative model (called *predictor*) to predict replacements for the chosen words.

Interpretability of SE models. There has also been recent interest in improving interpretability of models used in software engineering [6, 8, 31, 41]. Two of these efforts [31, 41] propose to simplify the code while retaining the model prediction. Another effort called AutoFocus [6] aims to rate and visualize the relative importance of different code elements by using a combination of attention layers in the neural network and deleting statements in the program. Another recent effort [8] aims for global interpretability and helps model developers by identifying which types of inputs the model performs poorly on. We believe that the counterfactual explanation generation technique proposed in this paper complements all of these efforts on improving SE model interpretability.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019).

[3] G. Aye, S. Kim, and H. Li. 2021. Learning Autocompletion from Real-World Datasets. , 131-139 pages. https://doi.org/10.1109/icse-seip52600.2021.00022

[4] Johannes Bader, Seohyun Kim, Sifei Luan, Erik Meijer, and Satish Chandra. 2021. AI in Software Engineering at Facebook. *IEEE Software* (2021).

[5] Sebastian Baltes, Oliver Moseler, Fabian Beck, and Stephan Diehl. 2015. Navigate, understand, communicate: How developers locate performance bugs. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[6] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: interpreting attention-based neural networks by code perturbation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 38–41.

[7] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343.

[8] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. 2021. Explaining Mispredictions of Machine Learning Models Using Rule Induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 716–727. https://doi.org/10.1145/3468264.3468614

[9] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What Does BERT Look at? An Analysis of BERT's Attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. 276–286.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*.

[11] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint* (2017).

[12] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-Box Adversarial Examples for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 31–36.

[13] Shi Feng, Eric Wallace, Alvin Grissom II, Mohit Iyyer, Pedro Rodriguez, and Jordan Boyd-Graber. 2018. Pathologies of Neural Models Make Interpretations Difficult. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 3719–3728.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*.

[15] Andrea Galassi, Marco Lippi, and Paolo Torroni. 2020. Attention in natural language processing. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[16] Sarthak Jain and Byron C Wallace. 2019. Attention is not Explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 3543–3556.

[17] Ehud Kalai and Dov Samet. 1987. On weighted Shapley values. *International journal of game theory* 16, 3 (1987), 205–222.

[18] Amir-Hossein Karimi, Bernhard Schölkopf, and Isabel Valera. 2021. Algorithmic recourse: from counterfactual explanations to interventions. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. 353–362.

[19] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. 2017. Interpretable & explorable approximations of black box models. *arXiv preprint arXiv:1707.01154* (2017).

[20] Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. 2016. Visualizing and Understanding Neural Models in NLP. In *Proceedings of NAACL-HLT*. 681–691.

[21] Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. Understanding neural networks through representation erasure. *arXiv preprint arXiv:1612.08220* (2016).

[22] Nishtha Madaan, Inkit Padhi, Naveen Panwar, and Diptikalyan Saha. 2021. Generate Your Counterfactuals: Towards Controlled Counterfactual Generation for Text. arXiv:2012.04698 [cs.CL]

[23] David Martens and Foster Provost. 2014. Explaining data-driven document classifications. *MIS quarterly* 38, 1 (2014), 73–100.

[24] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence* 267 (2019), 1–38.

[25] Christoph Molnar, Gunnar König, Julia Herbinger, Timo Freiesleben, Susanne Dandl, Christian A Scholbeck, Giuseppe Casalicchio, Moritz Grosse-Wentrup, and Bernd Bischl. 2020. General pitfalls of model-agnostic interpretation methods for machine learning models. *arXiv preprint arXiv:2007.04131* (2020).

[26] Jose G Moreno-Torres, Troy Raeder, Rocío Alaiz-Rodríguez, Nitesh V Chawla, and Francisco Herrera. 2012. A unifying view on dataset shift in classification. *Pattern recognition* 45, 1 (2012), 521–530.

[27] Rafael Poyiadzi, Kacper Sokol, Raul Santos-Rodriguez, Tijl De Bie, and Peter Flach. 2020. FACE: Feasible and actionable counterfactual explanations. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. 344–350.

[28] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.

[29] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[30] Danish Pruthi, Mansi Gupta, Bhuwan Dhingra, Graham Neubig, and Zachary C Lipton. 2020. Learning to Deceive with Attention-Based Explanations. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

[31] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding Neural Code Intelligence through Program Simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3468264.3468539

[32] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043* (2020).

[33] Kaivalya Rawal and Himabindu Lakkaraju. 2020. Beyond Individualized Recourse: Interpretable and Interactive Summaries of Actionable Recourses. *Advances in Neural Information Processing Systems* 33 (2020).

[34] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144. https://doi.org/10.1145/2939672.2939778

[35] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*.

[36] Alexis Ross, Ana Marasović, and Matthew E. Peters. 2021. Explaining NLP Models via Minimal Contrastive Editing (MiCE). arXiv:2012.13985 [cs.CL]

[37] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE.

[38] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.

[39] Ilia Stepin, Jose M Alonso, Alejandro Catala, and Martín Pereira-Fariña. 2021. A survey of contrastive and counterfactual explanation generation methods for explainable artificial intelligence. *IEEE Access* 9 (2021), 11974–12001.

[40] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*. PMLR, 3319–3328.

[41] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. 2021. Probing Model Signal-Awareness via Prediction-Preserving Input Minimization. arXiv:2011.14934 [cs.SE]

[42] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[43] Shikhar Vashishth, Shyam Upadhyay, Gaurav Singh Tomar, and Manaal Faruqui. 2019. Attention interpretability across nlp tasks. *arXiv preprint arXiv:1909.11218* (2019).

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[45] Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel S. Weld. 2021. Polyjuice: Generating Counterfactuals for Explaining, Evaluating, and Improving Models. arXiv:2101.00288 [cs.CL]

[46] Fan Yang, Ninghao Liu, Mengnan Du, and Xia Hu. 2021. Generative Counterfactuals for Neural Networks via Attribute-Informed Perturbation. arXiv:2101.06930 [cs.LG]

[47] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[48] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.