Semantic Robustness of Models of Source Code

Goutham Ramakrishnan [*1] **Jordan Henkel** [*1] **Zi Wang** [1] **Aws Albarghouthi** [1] **Somesh Jha** [1] **Thomas Reps** [1]

Abstract

Deep neural networks are vulnerable to adversarial examples—small input perturbations that result in incorrect predictions. We study this problem in the context of models of source code, where we want the network to be robust to source-code modifications that preserve code functionality. We define a natural notion of robustness, k-*transformation robustness*, in which an adversary performs up to k semantics-preserving transformations to an input program. We show how to train robust models using an adversarial training objective inspired by that of Madry et al. (2018) for continuous domains.

We implement an extensible framework for adversarial training over source code, and conduct a thorough evaluation on a number of datasets and two different architectures. Our results show (1) the increase in robustness following adversarial training, (2) the ability of training on weak adversaries to provide robustness to attacks by stronger adversaries, and (3) the shift in attribution focus of adversarially trained models towards semantic vs. syntactic features.

1. Introduction

While deep neural networks (DNNs) have been widely adopted in many areas of computing, it has been repeatedly shown that they are vulnerable to *adversarial examples* (Szegedy et al., 2013; Biggio et al., 2013; Goodfellow et al., 2014; Papernot et al., 2017): small, seemingly innocuous perturbations to the input that lead to incorrect predictions. Adversarial examples raise safety and security concerns, for example, in computer-vision models used in autonomous vehicles (Eykholt et al., 2018; Bhagoji et al., 2018) or for user authentication (Sharif et al., 2016). Significant progress has recently been made in identifying adversarial examples (attacks) and training models that are robust to such examples (defenses). However, The majority of the research has targeted computer-vision tasks (Carlini & Wagner, 2017; Madry et al., 2018; Szegedy et al., 2013), a continuous domain (we refer the reader to a recent tutorial by Kolter & Madry (2020) for a comprehensive overview.)

In this paper, we study the problem of robustness to adversarial examples in the discrete domain of deep neural networks for source code. With the growing adoption of neural models for programming tasks, robustness is becoming an important property. Why do we want robust models of code? There are many answers, ranging from usability to security. Consider, for instance, a model that explains in English what a piece of code is doing—the *code-captioning* task. A developer using such a model to navigate a new code base should not be exposed to completely different explanations for similar pieces of code. For a concrete illustration, consider the behavior of the state-of-the-art code2seq model (Alon et al., 2018) on the Java code in Fig. 1, where the prediction changes after adding logging print statements. Alternatively, imagine the security-critical setting of using a model to classify malware. We do not want a small modification to the malware's binary to cause the model to deem it safe.

We are not the first to study this problem. Wang & Christodorescu (2019) demonstrated the drop in accuracy of deep models over source code when applying standard transformations and refactorings; however, they did not propose defenses in their work. Recently, Yefet et al. (2019) developed a gradient-based attack that is specific to variable-name substitution or dead-code insertion, similar in spirit to attacks on natural-language models, like HotFlip (Ebrahimi et al., 2017), which can efficiently estimate gradients for token substitution and insertion. Yefet et al. (2019) propose a defense based on outlier detection, but do not consider arbitrary program transformations or adversarial training.

Our aim in this paper is to consider the general setting of an adversary that can apply a sequence of predefined source-code transformations. Specifically, we goal is to answer the following question:

Can we train models of source code that are robust to

[*]Equal contribution [1]University of Wisconsin–Madison. Correspondence to: Goutham Ramakrishnan <gouthamr@cs.wisc.edu>, Jordan Henkel <jjhenkel@cs.wisc.edu>, Zi Wang <zw@cs.wisc.edu>, Aws Albarghouthi <aws@cs.wisc.edu>, Somesh Jha <jha@cs.wisc.edu>, Thomas Reps <reps@cs.wisc.edu>.

```
int □(Object target)
  System.out.println("Begin search");
  int i = 0;
  for (Object elem: this.elements)
    if (elem.equals(target)) {
      System.out.println("Found");
      return i;
    i++;
  return -1;
```

Figure 1. code2seq (Alon et al., 2018) correctly predicts the name of this function as `indexOfTarget`. When we insert the logging statements, highlighted in gray, it predicts `search`. (Parentheses are elided for brevity.)

sequences of semantics-preserving transformations?

k-robustness. We begin by defining the notion of k-*transformation robustness* (or k-robustness, for short). The idea is that we have an adversary that can select up to k transformations from a prespecified set of semantics-preserving transformations T, and apply them to an input program. The adversary succeeds if it manages to change the prediction of the neural network. For example, a transformation may add dead code to a program, replace *for* loops with *while* loops, change variable names, replace one API call with another equivalent one, replace an expression like $\mathsf{x} \neq \mathsf{y}$ with the equivalent $\mathsf{y} \neq \mathsf{x}$, and so on.

As with other machine-learning tasks, the space of possible adversarial perturbations is infinite, and we cannot hope to model every possible adversary. We assume that we have a predefined set of transformations for the task at hand. For example, if we are training a code-understanding tool for C++, the set of transformations we use may be standard refactorings that developers apply in C++.

Challenges of training k-robust models. To train a k-robust model, we adapt the robust-optimization objective of Madry et al. (2018) to our discrete setting. Specifically, Madry et al. propose modeling the adversary within the training loop, such that the loss for a training point x is replaced with the loss for the *worst-case* perturbation of x. For example, in the case of image recognition, we can deploy a projected-gradient descent (PGD) attack (Carlini & Wagner, 2017) within the training loop to approximate the worst-case loss.

We can do the same in our setting: Instead of computing the loss for an input x (a program), we can compute the worst-case loss resulting from applying up to k transformations to x. While this approach is theoretically sound, it is hopelessly inefficient for two reasons.

First, the space of semantics-preserving transformations T is generally non-differentiable, so we cannot employ gradient-based approaches to search for an attack. As such, we have to explicitly search for a sequence of transformations. The number of sequences of transformations of length up to k explodes combinatorially with increasing values of k. Thus, even for small values of k, the search space can make training impractically slow.

Second, there is typically a large representation gap between programs, as objects of a data structure, and how they are presented to the neural network. Specifically, program transformations are typically defined as tree transformations over *abstract syntax trees* (AST). But the neural network usually receives as input some other program format, some of which lose program structure or entire portions of the program—for example, tokens or subtokens of program text, or randomly sampled paths through the AST (Alon et al., 2019; 2018). Therefore, modeling an adversary during training becomes challenging: every time we wish to transform a training point, we have to (1) go back to its original AST, (2) transform the AST, and, finally, (3) produce the neural-network representation of the transformed AST. This process makes the in-training adversary even more time consuming, because it requires invoking external, non-differentiable program-transformation tools.

To work around the complexity of training in this setting, we propose to generate sequences of transformations offline. This approach allows us to avoid applying AST transformations during training. Second, we demonstrate experimentally that training on small values of k results in models that are robust to larger values of k. This allows us to efficiently train adversarially without incurring a combinatorial explosion in the number of transformation sequences.

Framework and evaluation. In the process of conducting this research, we have developed an extensible framework for writing transformations and generating adversarial datasets. Our framework currently applies to both Java and Python, and can generate datasets in the form of subtokenized programs, ASTs, and sets of AST paths. The framework is available anonymously (FrameworkURL, 2020). We hope that it will aid in accelerating research in robust deep learning for source code.

We evaluate our approach on the task of *code summarization* (prediction of a method name, given its body). We consider a range of datasets for Java, a statically and explicitly typed language, and Python, a dynamically typed scripting language. We consider two different model architectures: a sequence-to-sequence BiLSTM model (Graves, 2013; Schuster & Paliwal, 1997) and an AST-path-based model, code2seq (Alon et al., 2018).

Our results demonstrate that (1) adversarial training radically improves the robustness of the model, e.g., for one of

```
( block_info [ ] blocks ) {
this . blocks = blocks ;
}
```

```
( block_info [ ] blocks )
this . blocks = blocks ;
}
```

a. Attribution scores under a normally trained model

```
( block_info [ ] blocks ) {
this . blocks = blocks ;
}
```

```
( block_info [ ] blocks )
this . blocks = blocks ;
}
```

b. Attribution scores under an adversarially trained model

Figure 2. For the above, tokenized code snippet, both the normally and adversarially trained (seq2seq) models predict the method name: `set_blocks`. However, the attributions assigned by the two models to the input tokens are very different. For each model, the token-wise attributions are shown for the prediction of `set` and `blocks` respectively, from left to right (with the coloration proportional to the attribution value for each token). Note how the robustly trained model assigns higher attribution values to input tokens with semantic meaning (such as `this`) rather than syntactic tokens (such as parentheses).

our Java datasets, the adversary can drop the F1 score of a normally trained model by 32 points, while only by 4 points for the adversarially trained one; and (2) we can train with a simple adversary of $k = 1$ and obtain a model that is robust on larger values of k, thus enabling efficient training.

To understand the qualitative effects of adversarial training, we investigate *feature attribution* using the *integrated gradients* method (Sundararajan et al., 2017). Our key finding is that adversarially trained models, in comparison with normally trained models, focus more on *semantic* features instead of *syntactic* features. Specifically, with adversarial training, we see greater attribution to language keywords and operators, and a drop in attribution to brackets, punctuation, numbers, etc. For a simple example, see Fig. 2, where the adversarially trained model assigns higher attribution values to input tokens with semantic meaning (such as `this` and `.`) vs. syntactic tokens (such as parentheses).

Contributions. We summarize our contributions below:

- We define *k-transformation robustness* for source-code tasks—i.e., robustness to an adversary that is allowed up to k semantics-preserving transformations to an input program.
- We propose the first adversarial training method for training neural models of code. Our method adapts the adversarial training objective of Madry et al. (2018) to our setting. To efficiently model the adversary during training, we pre-generate sequences of semantics-preserving transformations.
- We implement an extensible framework for adversarially training models of source code. Our framework allows defining transformations for Java and Python and generating adversarial datasets in different formats. It is available anonymously (FrameworkURL, 2020).
- We conduct a thorough evaluation of our approach on a variety of datasets and architectures. Our results demonstrate improvements in robustness, the power of weak adversaries for training, and the shift towards semantic features induced by adversarial training.

2. Related Work

Adversarial examples. Several researchers have explored various types of attacks on machine-learning algorithms, such as test-time, training-time, membership-inference, and backdoor attacks. Test-time attacks are most relevant to this paper. In test-time attacks an adversary perturbs an example so that it is misclassified by a model (untargeted attack) or the perturbed example is classified to an attacker-specified label (targeted attack) (Athalye et al., 2018; Biggio et al., 2013; Ilyas et al., 2018; Chen et al., 2017; Carlini & Wagner, 2017). Initially, test-time attacks were explored in the context of images. However, the domain considered in this paper is discrete (i.e., adding a vector to a program does not make semantic sense). Our domain is closer to test-time attacks in natural language processing (NLP). There are several test-time attacks in NLP that consider discrete transformations, such substituting words or introducing typos (Lei et al., 2019; Mudrakarta et al., 2018; Ebrahimi et al., 2017; Zhang et al., 2019b). A key difference between our domain and NLP is that in the case of programs one has to worry about semantics (i.e., the program has to work even after transformations).

Deep learning for source code. Recent years have seen huge progress in deep learning for source-code tasks; see Allamanis et al. (2018) for a survey. In our evaluation, we have focused on two key encodings of programs for neural networks. First, we considered (sub-)tokenizing the program, in a way analogous to natural-language-processing tasks, and using a variant of recurrent neural networks (RNNs). This technique has appeared in numerous papers, e.g., the pioneering work of Raychev et al. (2014) for code completion and the recent work of Hellendoorn et al. (2018) on type inference. We have also considered the AST (abstract syntax tree) paths encoding pioneered by Alon et al. (2019; 2018). Researchers have considered more structured networks, like graph neural networks (GNNs) (Allamanis et al., 2016) and tree-LSTMs (Zhao et al., 2018), where typically the AST is encoded as a graph or a tree. These would be interesting to consider for future experimentation in the context of adversarial training. The task we evaluated on, code summarization, was, to the best of our knowledge, first introduced by Allamanis et al. (2016).

In Section 1, we additionally discussed two recent works (Yefet et al., 2019; Wang & Christodorescu, 2019) on adversarial examples of source code, which, to our knowledge, are the only two that have studied the problem.

3. Transformation Robustness

In this section, we formally define k-transformation robustness and our robust-optimization objective. We begin with preliminaries and background on adversarial training.

3.1. Adversarial training via robust optimization

Learning problem. We assume a data distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ is the space of samples and $\mathcal{Y}$ is the space of labels. In the *empirical risk minimization (ERM)* framework, we wish to solve the following optimization problem:

$$\operatorname*{argmin}_{w \in \mathcal{H}} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} L(w, x, y)$$

where $\mathcal{H}$ is the hypothesis space and L is the loss function. Once we have solved the optimization problem given above, we obtain a w^*, which yields a function $F_{w^\star} : \mathcal{X} \to \mathcal{Y}$.

Adversarial training objective. To construct classifiers resilient to adversarial attacks, we use a *robust-optimization* framework (Ben-Tal et al., 2009), which solves the following optimization problem:

$$\operatorname*{argmin}_{w \in \mathcal{H}} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \max_{z \in U(x)} L(w, z, y)$$

In the equation given above, $U(x) \subseteq \mathcal{X}$ is called the *uncertainty set*, and represents a set of possible *perturbations* of the sample x. The robust-optimization problem is usually solved using *adversarial training*. Adversarial training works as follows: let $\mathcal{A}$ be an attack algorithm that, given an x, attempts to find z according to the objective

$$\max_{z \in U(x)} L(w, z, y)$$

In other words, an attacker tries to find an example in the uncertainty set that maximizes the loss. Note that $\mathcal{A}$ might not achieve the objective exactly because the inner maximization problem can be hard to solve. We think of $\mathcal{A}$ as producing a subset of the uncertainty set, i.e., $\mathcal{A}(x) \subseteq U(x)$, for any $x \in \mathcal{X}$. Therefore, with adversarial training, a learner solves the following problem:

$$\operatorname*{argmin}_{w \in \mathcal{H}} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \max_{z \in \mathcal{A}(x)} L(w, z, y)$$

3.2. Adversarial training for source-code tasks

We are interested in tasks where the sample space $\mathcal{X}$ is that of programs in some programming language. We do not constrain the space of outputs $\mathcal{Y}$—it could be a finite set of labels for classification, a natural-language description, etc.

Abstract syntax trees. We view a program x as an *abstract syntax tree* (AST)—a standard data structure for representing programs. The internal nodes of an AST typically represent program constructs, such as while loops and conditionals, and the leaves of the tree denote variable names, constants, fields, etc. Fig. 3 shows a conditional statement and its AST representation.

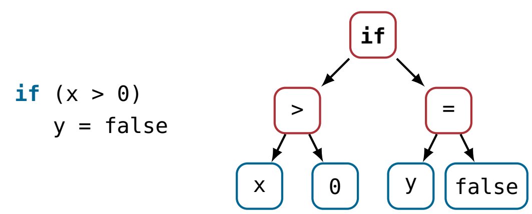


Figure 3. Program snippet and its AST

Transformations. A transformation T of a program x transforms it into another program x'. Typically, T is defined over ASTs, and therefore it is a tree-to-tree transformation. For our goal of semantic robustness, we will focus on *semantics-preserving* transformations, i.e., ones that do not change the behavior of the program. In other words, viewed functionally, for any input i, both x and the transformed x' return the same output. However, our approach is not tied to semantics-preserving transformations, and one could define transformations that, for example, introduce common typos and bugs that programmers make.

As an example, consider the semantics-preserving transformation T shown in Fig. 4, where we replace `x > 0` with `0 < x` (the transformed subtree is highlighted).

k-adversary. Given a set of transformations $\mathcal{T}$, a k-*adversary* as a function that tries to find a sequence of transformations of size up to k that maximizes the loss function.

We use q to denote a sequence of transformations $T_1, \ldots, T_n$, where each $T_i \in \mathcal{T}$. We use $q(x)$ to denote

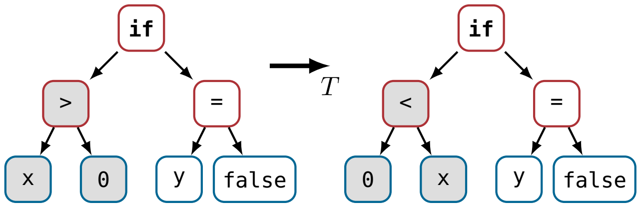


Figure 4. Example of an AST transformation

the program $T_n(\ldots T_2(T_1(x)))$, i.e., the result of applying all transformations in q to a program x. Let Q^k and $Q^{\leqslant k}$ denote the set of all sequences of transformations from $\mathcal{T}$ that are of length equal to k and $\leqslant k$, respectively. Finally, the goal of a k-adversary is to solve the following objective:

$$\max_{q \in Q^{\leqslant k}} L(w, q(x), y)$$

Robust-optimization objective. Now that we have formally defined a k-adversary, we can solve an optimization problem to achieve robustness to such attacks—we call this k-robustness. Specifically, we solve the following problem:

$$\underset{w \in \mathcal{H}}{\operatorname{argmin}} \underset{(x,y)\sim\mathcal{D}}{\mathbb{E}} \underbrace{\max_{q \in Q^{\leqslant k}} L(w, q(x), y)}_{\text{objective of } k\text{-adversary}}$$

3.3. Representation mismatch and efficient training

There is usually a mismatch between the program as an AST and the format received by the neural network. For example, in a recent popular work (Alon et al., 2019; 2018), the neural network is given a sampled set of paths from one leaf of the AST to another, passing through the least-common ancestor. We assume that we have a function R that maps a program x into the appropriate representation for the neural network.[1] Therefore, technically, we are training with the following optimization objective:

$$\underset{w \in \mathcal{H}}{\operatorname{argmin}} \underset{(x,y)\sim\mathcal{D}}{\mathbb{E}} \max_{q \in Q^{\leqslant k}} L(w, R(q(x)), y)$$

As we discussed in Section 1, the difficulty of training with respect to this objective is two-fold:

1. Solving the inner objective (max) is complex: Because the space of transformations is non-differentiable, we have to explicitly search for a sequence of transformations. The number of such sequences explodes combinatorially with k.
2. A program x is an AST and the adversary's transformations are defined over ASTs. But the neural network expects as input a different representation, $R(x)$. Therefore, to solve the inner maximization problem, we have to translate back and forth between ASTs and their representation. This approach is expensive to employ during training, as one has to apply transformations using an external program-analysis tool and convert the transformed AST back to its training representation.

We address these challenges as follows: Empirically, we observe that training with a k-adversary is sufficient to produce models that are robust for much larger values of k. This observation implies that we can train with small values of k for which we can exhaustively generate all possible transformation sequences. Second, to avoid calling program-analysis tools to perform transformations within the training loop, we pre-generate all possible transformed programs that are considered by the adversary; that is, for every x in the training set S, we generate the following set offline, before training: $S^x = \{R(q(x)) \mid q \in Q^{\leqslant k}\}$. Therefore, we end up solving the following robust-optimization objective:

$$\underset{w \in \mathcal{H}}{\operatorname{argmin}} \underset{(x,y)\sim\mathcal{D}}{\mathbb{E}} \max_{z \in S^x} L(w, z, y)$$

Next, we describe our data-generation framework and discuss the results of our experimental evaluation.

4. Data-Generation Framework & Evaluation

In this section, we first describe how our data-generation framework (FrameworkURL, 2020) produces data in representations amenable to our model architectures, and the program-transformation frameworks we employ. We then describe the primary research questions we investigate and our experimental setup. Finally, we present our findings in a two-fold evaluation of our adversarially trained models: (1) analysis of their robustness to attack by a k-adversary, and (2) an interpretability study that suggests that adversarial training helps models focus more on semantic features.

4.1. Data-generation framework

We conduct experiments on four datasets in two languages, Java and Python. Java is statically typed with types explicitly stated in the code, while Python is a dynamically typed scripting language and mostly involves no type annotations. The datasets originate from three different sources: (i) code2seq's java-small dataset (c2s/java-small) (Alon et al., 2018), (ii) GitHub's CodeSearchNet Java and Python datasets (csn/java, csn/python) (Husain et al., 2019), and SRI Lab's Py150k dataset (sri/py150) (Raychev et al., 2016).

[1]Note this is unrelated to *representation learning*. R is simply a translation of an AST into a different format, e.g., sequence of tokens or AST paths.

Table 1. Normalized (function-level) datasets

Dataset	Train	Validation	Test
c2s/java-small	500.3k	26.3k	38.9k
csn/java	454.3k	15.3k	26.9k
sri/py150	693.7k	81.7k	86.0k
csn/python	408.2k	22.8k	21.9k

Table 2. Transformations

Index	Name
T1	`RenameLocalVariables`
T2	`RenameParmeters`
T3	`RenameFields`
T4	`ReplaceTrueFalse`
T5	`InsertPrintStatements`
T6	`ShuffleLocalVariables`
T7	`ShuffleParameters`

Data normalization. These datasets come from their three original sources in three different formats. To resolve this discrepancy, our data-generation framework translates the original datasets into a *normalized representation*. As part of this process, Java and Python files are broken down into individual functions, and each individual function is stored, as a JSON object, with its original source code, tokenized body, sub-tokenized name, hash, and other features of interest. Table 1 provides the sizes of the four datasets.

Attack pre-generation. Using the normalized data as input, our data-generation framework pre-generates the transformed variants for each function. This process is expensive, and requires two separate code-transformation pipelines: one for Java programs, based on Spoon (Pawlak et al., 2015); another for Python programs, based on Astor (Berkerpeksag, 2020). The seven semantics-preserving transformations we use in our experiments are listed in Table 2. Our framework is capable of composing and applying arbitrary-length sequences of transforms.

We note that many of our transforms, such as the `Rename*` transforms, are parameterizable and non-deterministic. Because of the non-determinism involved in transforms such as random renaming, our data pre-generation framework may produce datasets of differing 'attack strength' from the same original data. We also note that the transforms T1-5 are commutative, as they modify different parts of the input program. Therefore, we consider order-independent combinations of these transforms in our experiments, instead of all possible permutations.

4.2. Experimental setup

Research questions. We designed our experiments to answer the following research questions:

Q1 Does adversarial training improve model robustness to semantics-preserving program transformations?

Q2 Is training adversarially with a k-adversary sufficient to achieve robustness against l-adversaries, for $l > k$?

Q3 Do adversarially trained models learn to focus on the semantically important features of the input program?

Code-summarization task. We considered the task of automatic code summarization (Allamanis et al., 2016), the prediction of a method's name, given its body. Method-name prediction requires an understanding of the high-level semantics of a program, and therefore presents a challenging task for deep-learning models.

Models. We experimented with two model architectures: (i) a sequence-to-sequence BiLSTM model (seq2seq) and (ii) Alon et al.'s (2018) code2seq model. The seq2seq model takes tokenized programs as inputs. We sub-tokenized the input program by splitting up camel-case and snake-case, and retain the original casing of tokens. We trained 2-layer BiLSTM models, with 512 units in the encoder and decoder, with embedding sizes of 512, and the input- and output-vocabulary sizes limited to 50k each. We built upon the implementation of IBM (2020) and trained our models for 10 epochs using the Adam optimizer.

code2seq (Alon et al., 2018) is a model for code summarization that samples paths from the program AST in its encoder, and uses a decoder with attention for predicting output tokens. We used the code2seq TensorFlow code,[2] default model parameters, their Java path extractor, along with a recently contributed python path extractor (stasbel, 2020). We train all code2seq models for 15 epochs.

All models were trained and evaluated using NVIDIA GPUs (GeForce RTX 2080 Ti and Tesla V100).

4.3. Robustness evaluation

We evaluated the robustness obtained by adversarial training of the two model architectures across the four datasets. We trained three models for each of the two architectures: normally, adversarially with Q^1 (Tr-Q^1), and adversarially with Q^{1+} (Tr-Q^{1+}), where Q^{1+} is Q^1 together with the sequence of transformations T1-5. Therefore, $|Q^1| = 7$ and $|Q^{1+}| = 8$. Henceforth, we refer to Q^1 and Q^{1+} as the 'weak' and 'strong' adversaries/attacks, respectively. Analogously, Tr-Q^1 and Tr-Q^{1+} are the weakly and strongly adversarially trained models, respectively.

We summarize the results in Table 3. Consider the seq2seq models trained on the c2s/java-small dataset. The normally trained model exhibits sharp drops in F1 under both weak (40.81 to 17.87) and strong (40.81 to 14.62) attacks. On

[2] `https://github.com/tech-srl/code2seq`

Table 3. **Evaluation of robustly trained models across four datasets on the task of code summarization**. The F1 scores obtained upon evaluation on the original (Ref), Q^1 (weak adversary) and Q^{1+} (strong adversary) test set are shown in the columns. The most robust models against the strong adversary are shown in bold, which was always Tr-Q^{1+}.

Model	Training	c2s/java-small			csn/java			sri/py150			csn/python		
		Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}
seq2seq	Normal	40.81	17.87	14.62	39.67	13.98	8.55	41.72	23.43	19.28	33.29	13.67	7.90
	Tr-Q^1	37.94	34.56	33.76	38.44	33.74	32.34	40.34	37.22	36.31	32.80	29.92	29.25
	Tr-Q^{1+}	37.63	35.15	**34.78**	36.34	33.15	**32.54**	39.08	37.06	**36.62**	31.55	29.86	**29.49**
code2seq	Normal	39.51	28.81	21.02	36.05	17.75	12.30	26.94	22.18	19.94	12.07	7.33	5.92
	Tr-Q^1	39.23	36.03	23.04	37.58	33.90	23.78	26.83	25.06	22.69	11.14	9.17	8.18
	Tr-Q^{1+}	39.08	35.58	**34.88**	37.28	33.51	**32.60**	26.76	24.90	**24.76**	11.43	9.27	**9.01**

the other hand, the Tr-Q^1 model suffers a drop of just a few F1 points under weak (37.94 to 34.56) and strong (37.94 to 33.76) attacks. The Tr-Q^{1+} model exhibits similar robustness to both weak (drop of 2.48) and strong (drop of 2.85) attacks. Similar trends may be observed across the other datasets and the code2seq model. *Both the weakly and strongly adversarially trained models show remarkable robustness to both attacks*. The small drop in their performance on the baseline test data (Ref) is compensated by their robustness to attacks. The performance-robustness trade-off has been previously observed for adversarial training in other machine-learning domains (Tsipras et al., 2019).

Though the Tr-Q^{1+} model is more robust to strong attacks than the Tr-Q^1 model in general, their F1 performance and robustness is comparable across datasets. In particular for the seq2seq model, the performance of the Tr-Q^1 model upon strong attack is within 3% of Tr-Q^{1+}, across datasets. This is a remarkable result: *adversarial training with a weak adversary (Tr-Q^1) is sufficient to achieve robustness against a much stronger attack (Q^{1+})*. This finding is analogous to results in computer vision, where training using a weak adversary (small number of PGD steps) builds robustness against stronger attacks (Zhang et al., 2019a; Madry et al., 2018; Wong et al., 2020).

$\mathbf{Q^{\leqslant 5}}$ vs $\mathbf{Q^{1+}}$. How does the 5-adversary attack ($Q^{\leqslant 5}$) compare to the above strong adversary (Q^{1+})? Figure 9b compares the relative strength of adversaries on the normal and Tr-Q^{1+} seq2seq models on csn/java. The strength of the attack increases with the length of transformation sequences. The slight non-monotonicity can be explained by the fact that the attack chosen for each test instance maximizes the loss, which may not necessarily correspond to the greatest drop in F1 score.

Observe that the attack efficacy of Q^{1+} is within a few F1 points (4.75) of $Q^{\leqslant 5}$ across datasets, justifying its tag of 'strong adversary'. See the appendix for more graphs.

*To summarize, our results demonstrate that (**Q1**) adversarial training increases robustness of the trained model, and additionally, (**Q2**) provides high robustness to stronger adversaries for the seq2seq model.*

seq2seq vs. code2seq. There are interesting contrasts between the seq2seq and code2seq models. The code2seq model struggles in two respects in comparison to seq2seq: (1) its performance on Python datasets is far inferior, and (2) the code2seq Tr-Q^1 models are not as robust to the strong attack on the two Java datasets. We conjecture that this difference arises because the code2seq model uses sampled program AST paths to make its prediction. In general, python is more abstract, less verbose, includes no type annotations, and its program ASTs are less descriptive than Java ASTs. As evidence of this, we find that the node-type histogram code2seq generates for Java ASTs has over 300 entries, whereas the node-type histogram generated for our python ASTs has only 151 entires. Moreover, a program transformation may affect several AST paths at once, making the code2seq model more vulnerable to attack.

4.4. Focus on semantic features

We also conducted an interpretability study on our models, which showed that adversarial training results in increased focus on semantic features of the input programs. We considered *feature attributions*, by which we analyze the contribution of each input feature towards the output of a model. To estimate attributions, we used *integrated gradients* (IG) (Sundararajan et al., 2017), a popular recent technique that satisfies several desirable properties.

The IG technique yields a vector of attributions, with each element corresponding to the contribution of an input feature towards the output. Formally, given a function $F : \mathbb{R}^n \to \mathbb{R}$, an input x, and a baseline input x', the i^{th} component of $\mathrm{IG}(x, x') \in \mathbb{R}^n$ is defined as the path integral of gradients along the straight line path from baseline x' to input x.

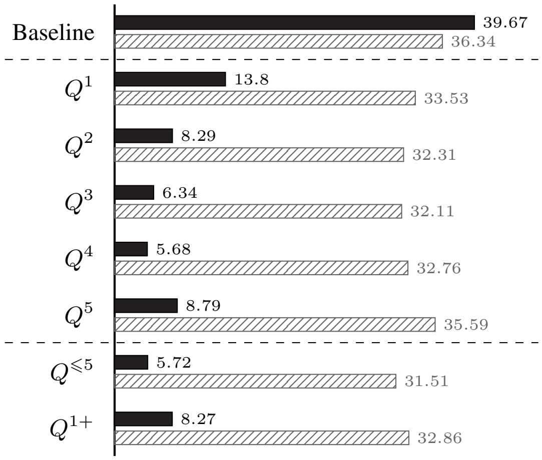


Figure 5. F1 scores of the normally trained (solid) and Tr-Q^{1+} (dashed) seq2seq models on csn/java dataset with respect to k-adversaries for $k = 1, \ldots, 5$

Table 4. Input token classes in descending order of average IG, for the normally and adversarially trained models on the c2s/java-small dataset (Avg. IG $\times$ 100 in brackets). Increased (▲) and decreased (▼) average IG shown in colors.

Normal	Tr-Q^{1+}
KEYWORD (1.49)	▲ KEYWORD (2.01)
OTHER (0.58)	▲ WORD (0.55)
BRACKET (0.51)	▼ BRACKET (0.31)
WORD (0.45)	▼ PUNCTUATION (0.29)
PUNCTUATION (0.32)	▲ OPERATOR (0.16)
NUMBER (0.09)	▼ NUMBER (0.06)
OPERATOR (0.04)	▼ OTHER (-0.06)

Mathematically:

$$\mathrm{IG}_i(x, x') := (x_i - x'_i) \int_0^1 \frac{\partial F(x' + \alpha(x - x'))}{\partial x_i} d\alpha$$

For our analysis, we focused on the seq2seq model and analyzed the IG attributions of each predicted output token with respect to each (sub)token of the input program.[3] At each decoder time step, the function F for the IG computation is the element in the final softmax layer corresponding to predicted token. Input tokens are fed into the seq2seq encoder through a non-differentiable embedding layer, which converts the sequence of input tokens into a matrix of embeddings. To get around this representation issue, we compute the IG of each output token with respect to each element of the input embedding matrix. Then, the IG of a given input token is taken to be the sum of the IG values of elements in its embedding vector. We used the matrix of zero embeddings as the input baseline for IG calculation.

[3]An analogous analysis on the code2seq model is difficult to interpret, because the model takes sampled AST paths as input.

`valid` $\xrightarrow{\Delta 0.051}$ `is`	`return` $\xrightarrow{\Delta 0.049}$ `get`
`cublas` $\xrightarrow{\Delta 0.036}$ `cuda`	`writer` $\xrightarrow{\Delta 0.028}$ `print`
`mercator` $\xrightarrow{\Delta 0.028}$ `bounding`	`mercator` $\xrightarrow{\Delta 0.056}$ `box`

Figure 6. 6 of the 25 (input token, predicted token) tuples with the highest absolute increase in mean IG scores (csn/java)

For our experiments, we considered the normally trained and Tr-Q^{1+} models. For brevity, we present selected results and discussion here, with full results and details of the IG computation in the appendix.

Increased IG for semantic features. We broadly classify the types of program (sub)tokens into 7 categories: keyword, word, number, operator, punctuation, bracket and other. Keywords, for instance, refer to a predefined list of Java and Python keywords, while words refer to variable names, types, etc. Any token not in the main categories is assigned to *other*. We examined the average IG attributions given to tokens in each category, across all input programs in the test set. The relative comparison between the normally and adversarially trained models is shown in Table 4, for the c2s/java-small dataset. We observe that the average IG attributions across the test set increases for semantically relevant token categories (keywords, words, operators) and decreases for the others (bracket, punctuation, other). Thus, adversarial training appears to help the model *focus* on semantic features.

Increased IG for specific tokens. To obtain a more fine-grained view of the changes in attribution that result from adversarial training, we considered the average IG attributions accorded to a given input token, by a given output token. We obtained the tuples of (input token, predicted token) that showed the greatest absolute increase in average IG after adversarial training. Figure 6 presents six intriguing examples for tuples of this form from the csn/java dataset. In each example, the input word gained a significant amount of IG attribution after adversarial training. For instance, on average the output token `get` had an IG of 0.028 more towards the input token `return`, after adversarial training. Another intuitive example is `writer` and `print` (The `mercator` examples refer to the OpenMap API in java, for which `bounding` and `box` are relevant). Thus, adversarial training increases focus on intuitive and logical input features.

*To summarize, our results provide (**Q3**) strong evidence that adversarial training shifts a model's attention towards semantic vs. syntactic features.*

5. Conclusion

To our knowledge, this work is the first to address adversarial training for source-code tasks. Our approach is general, in that it considers adversaries that perform arbitrary transformations. Our results demonstrate (i) the power of adversarial training at improving robustness, and (ii) the qualitative change in attributions induced by adversarial training—shifting toward semantic features of code.

For future work, it would be interesting to study the effects of adversarial training on other models, e.g., graph neural networks. It would also be interesting to look into efficiently incorporating gradient-directed transformations, e.g., (Yefet et al., 2019), into our set of transformations.

References

Allamanis, M., Peng, H., and Sutton, C. A. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016. URL http://arxiv.org/abs/1602.03001.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Athalye, A., Carlini, N., and Wagner, D. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

Ben-Tal, A., El Ghaoui, L., and Nemirovski, A. *Robust optimization*, volume 28. Princeton University Press, 2009.

Berkerpeksag. berkerpeksag/astor, Jan 2020. URL https://github.com/berkerpeksag/astor.

Bhagoji, A. N., He, W., Li, B., and Song, D. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *European Conference on Computer Vision*, pp. 158–174. Springer, 2018.

Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pp. 387–402. Springer, 2013.

Carlini, N. and Wagner, D. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pp. 39–57. IEEE, 2017.

Chen, P.-Y., Zhang, H., Sharma, Y., Yi, J., and Hsieh, C.-J. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp. 15–26. ACM, 2017.

Ebrahimi, J., Rao, A., Lowd, D., and Dou, D. Hotflip: White-box adversarial examples for text classification, 2017.

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T., and Song, D. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1625–1634, 2018.

FrameworkURL. Our data-generation framework for adversarial training, Feb 2020. URL https://anonymous.4open.science/r/4ac75222-3673-418e-8b86-9f36ba29fc59.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Graves, A. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL http://arxiv.org/abs/1308.0850.

Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 152–162, 2018.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search, 2019.

IBM. Ibm/pytorch-seq2seq, Jan 2020. URL https://github.com/IBM/pytorch-seq2seq.

Ilyas, A., Engstrom, L., Athalye, A., and Lin, J. Black-box adversarial attacks with limited queries and information. *arXiv preprint arXiv:1804.08598*, 2018.

Kolter, Z. and Madry, A. Ibm/pytorch-seq2seq, Feb 2020. URL https://adversarial-ml-tutorial.org.

Lei, Q., Wu, L., Chen, P.-Y., Dimakis, A. G., Dhillon, I. S., and Witbrock, M. Discrete adversarial attacks and submodular optimization with applications to text classification. In *SysML*, 2019.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJzIBfZAb.

Mudrakarta, P. K., Taly, A., Sundararajan, M., and Dhamdhere, K. Did the model understand the question? In *ACL*, 2018.

Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 506–519. ACM, 2017.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. doi: 10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428, 2014.

Raychev, V., Bielik, P., and Vechev, M. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pp. 731–747, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344449. doi: 10.1145/2983990.2984041. URL https://doi.org/10.1145/2983990.2984041.

Schuster, M. and Paliwal, K. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45: 2673 – 2681, 12 1997. doi: 10.1109/78.650093.

Sharif, M., Bhagavatula, S., Bauer, L., and Reiter, M. K. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1528–1540. ACM, 2016.

stasbel. tech-srl/code2seq, Jan 2020. URL https://github.com/tech-srl/code2seq/tree/master/Python150kExtractor.

Sundararajan, M., Taly, A., and Yan, Q. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 3319–3328. JMLR. org, 2017.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Tsipras, D., Santurkar, S., Engstrom, L., Turner, A., and Madry, A. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=SyxAb30cY7.

Wang, K. and Christodorescu, M. Coset: A benchmark for evaluating neural program embeddings, 2019.

Wong, E., Rice, L., and Kolter, J. Z. Fast is better than free: Revisiting adversarial training. *arXiv preprint arXiv:2001.03994*, 2020.

Yefet, N., Alon, U., and Yahav, E. Adversarial examples for models of code. *arXiv preprint arXiv:1910.07517*, 2019.

Zhang, D., Zhang, T., Lu, Y., Zhu, Z., and Dong, B. You only propagate once: Accelerating adversarial training via maximal principle. In *Advances in Neural Information Processing Systems*, pp. 227–238, 2019a.

Zhang, W. E., Sheng, Q. Z., Alhazmi, A., and LI, C. Adversarial attacks on deep learning models in natural language processing: A survey. *arXiv preprint arXiv:1901.06796*, 2019b.

Zhao, J., Albarghouthi, A., Rastogi, V., Jha, S., and Octeau, D. Neural-augmented static analysis of android communication. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 342–353, 2018.

Appendix: Semantic Robustness of Models of Source Code

6. Details of Program Transformations

We employ a total of nine semantics preserving transforms: seven single-step transforms; an `All` transform (which is the sequential composition of `RenameLocalVariables`, `RenameParameters`, `RenameFields`, `ReplaceTrueFalse`, and `InsertPrintStatements`); and an `Identity` transform which we use to validate our transformation pipeline. Each of these nine transforms are descried below:

1. **`Identity:`** the code passes through the transformation framework without modification.
2. **`RenameLocalVariables:`** local variables are identified and randomly renamed.
3. **`RenameParmeters:`** function parameters are identified and randomly renamed.
4. **`RenameFields:`** fields are identified and randomly renamed.
5. **`ReplaceTrueFalse:`** the literal values `true` and `false` are replaced by an equivalent expression.
6. **`ShuffleLocalVariables:`** all of the names of local variables are permuted.[4]
7. **`ShuffleParameters:`** all of the names of function parameters are permuted.[5]
8. **`InsertPrintStatements:`** `System.out.println()`, in Java, and `print()` in Python, are appended to the beginning and end of the function—the argument to print is randomly generated.
9. **`All:`** the five non-shuffle transformations are applied in sequence.

For each `Rename*` transformation there are several parameters: (i) a set of sub-tokens from which random names are composed, (ii) the minimum random name length, (iii) the maximum random name length, and (iv) the percentage of identified targets that are selected for renaming. In our framework we default these to using names of minimum length 2, maximum length 10, and we attempt to rename 100% of identified targets. We use a set of 314 sub-tokens that, when applied as a random renaming in a small set of hand built templates, often changed the prediction of a Code Summarization model. In this sense, our choice of sub-token set is slightly optimized. We also performed preliminary experiments using a sub-token set consisting of simply the top-N most frequently used sub-tokens. This works, but creates transforms that are slightly weaker than those generated by our alternate (semi-targeted) approach.

For the `ReplaceTrueFalse` transformation we replace each instance of `true` (`True` in Python) with either `1 == 1` or `0 == 0`. We choose each variant with equal likelihood. Similarly, we replace each instance of `false` (`False` in Python) with either `1 != 1` or `0 != 0`. Again, we choose each variant with equal likelihood.

Finally, for the `InsertPrintStatements` transform we insert a random number of print statements with a single literal value as an argument. This transform is parameterized by the minimum number of insertions, the maximum number of insertions, the minimum and maximum literal value length (in sub-tokens), the sub-token set used for literal generation, and the likelihood of inserting at the beginning of a procedure versus inserting at the end of a procedure. We default these to using random literals of minimum length 3 and maximum length 10; we insert a minimum of 3 print statements and a maximum of 10; we use the same sub-token set used for random renaming during literal generation; and, we insert at the beginning and end of procedures with an equal likelihood.

Three examples of code before and after transformations are presented in Figs. 7 to 9 on the following page.

[4] Ideally, a uniformly random derangement would be generated. For efficiency, we settle for a random permutation.

[5] See footnote 1.

```java
public void testCheckCommitAixCompatMode() throws IOException {
    DFSClient dfsClient = Mockito.mock(DFSClient.class);
    Nfs3FileAttributes attr = new Nfs3FileAttributes();
    HdfsDataOutputStream fos = Mockito.mock(HdfsDataOutputStream.class);
    // Last argument "true" here to enable AIX compatibility mode.
    OpenFileCtx ctx = new OpenFileCtx(fos, attr, "/dumpFilePath",
        dfsClient, new IdUserGroup(new NfsConfiguration()), true);
    // Test fall-through to pendingWrites check in the event that commitOffset
    // is greater than the number of bytes we've so far flushed.
    Mockito.when(fos.getPos()).thenReturn(((long) (2)));
    COMMIT_STATUS status = ctx.checkCommitInternal(5, null, 1, attr, false);
    Assert.assertTrue(status == COMMIT_STATUS.COMMIT_FINISHED);
    // Test the case when we actually have received more bytes than we're trying
    // to commit.
    Mockito.when(fos.getPos()).thenReturn(((long) (10)));
    status = ctx.checkCommitInternal(5, null, 1, attr, false);
    Assert.assertTrue(status == COMMIT_STATUS.COMMIT_DO_SYNC);
}
```

```java
public void testCheckCommitAixCompatMode() throws IOException {
    DFSClient dfsClient = Mockito.mock(DFSClient.class);
    Nfs3FileAttributes attr = new Nfs3FileAttributes();
    HdfsDataOutputStream fos = Mockito.mock(HdfsDataOutputStream.class);
    // Last argument "true" here to enable AIX compatibility mode.
    OpenFileCtx ctx = new OpenFileCtx(fos, attr, "/dumpFilePath",
        dfsClient, new IdUserGroup(new NfsConfiguration()), 1 == 1);
    // Test fall-through to pendingWrites check in the event that commitOffset
    // is greater than the number of bytes we've so far flushed.
    Mockito.when(fos.getPos()).thenReturn(((long) (2)));
    COMMIT_STATUS status = ctx.checkCommitInternal(5, null, 1, attr, 0 != 0);
    Assert.assertTrue(status == COMMIT_STATUS.COMMIT_FINISHED);
    // Test the case when we actually have received more bytes than we're trying
    // to commit.
    Mockito.when(fos.getPos()).thenReturn(((long) (10)));
    status = ctx.checkCommitInternal(5, null, 1, attr, 1 != 1);
    Assert.assertTrue(status == COMMIT_STATUS.COMMIT_DO_SYNC);
}
```

Figure 7. A procedure from the c2s/java-small dataset transformed by the `ReplaceTrueFalse` transform.

```python
def delete_record(cls, record):
    """Delete a record and it's persistent identifiers."""
    record.delete()
    PersistentIdentifier.query.filter_by(object_type='rec', object_uuid=
        record.id).update({PersistentIdentifier.status: PIDStatus.DELETED})
    cls.delete_buckets(record)
    db.session.commit()
```

```python
def delete_record(scheme_job_scheduler, response_master_running_report_args):
    """Delete a record and it's persistent identifiers."""
    response_master_running_report_args.delete()
    PersistentIdentifier.query.filter_by(object_type='rec', object_uuid=
        response_master_running_report_args.id).update({
        PersistentIdentifier.status: PIDStatus.DELETED})
    scheme_job_scheduler.delete_buckets(response_master_running_report_args)
    db.session.commit()
```

Figure 8. A procedure from the csn/python dataset transform by the `RenameParameters` transform.

```python
def replace(name):
    """Replaces a snapshot"""
    app = get_app()
    snapshot = app.get_snapshot(name)
    if not snapshot:
        click.echo("Couldn't find snapshot %s" % name)
        sys.exit(1)
    app.remove_snapshot(snapshot)
    app.create_snapshot(name)
    click.echo('Replaced snapshot %s' % name)
```

```python
def replace(identifier_commit_invalid):
    print('line_maps')
    print('mode_clean_invalid_mask_valid_am_fetch')
    """Replaces a snapshot"""
    app = get_app()
    tracker_or_and_reader_failover = app.get_snapshot(identifier_commit_invalid
        )
    if not tracker_or_and_reader_failover:
        click.echo("Couldn't find snapshot %s" % identifier_commit_invalid)
        sys.exit(1)
    app.remove_snapshot(tracker_or_and_reader_failover)
    app.create_snapshot(identifier_commit_invalid)
    click.echo('Replaced snapshot %s' % identifier_commit_invalid)
    print('usage_incr_segment_stat_feature_started_apps')
    print('metadata_read_locations_summary_one')
```

Figure 9. A procedure from the csn/python dataset transformed by the `All` transform.

7. Additional Results

7.1. Complete Robustness Evaluation

We provide the precision, recall, and exact match scores for all of our trained models across all of our four datasets in Tables 5 to 7. We highlight, in boldface, the most robust seq2seq and code2seq models (with respect to the Q^{1+} attack).

Table 5. The **precision/recall** scores obtained upon evaluation on the original (Ref), Q^1 (weak adversary) and Q^{1+} (strong adversary) test set are shown in the columns for the c2s/java-small and csn/java datasets.

Model	Training	c2s/java-small			csn/java		
		Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}
seq2seq	Normal	29.32 / 32.72	11.87 / 12.98	9.56 / 10.43	29.14 / 33.65	9.92 / 11.14	5.91 / 6.65
	Tr-Q^1	27.10 / 30.15	24.22 / 27.56	23.67 / 26.94	28.07 / 32.28	23.91 / 28.26	22.86 / **26.99**
	Tr-$Q^{1,5}$	26.74 / 29.81	24.77 / 27.91	**24.51 / 27.67**	26.35 / 30.14	23.53 / 27.47	**23.02** / 26.97
code2seq	Normal	49.77 / 32.75	34.28 / 24.84	25.09 / 18.08	43.96 / 30.55	18.76 / 16.84	13.09 / 11.60
	Tr-Q^1	50.12 / 32.22	46.22 / 29.53	30.39 / 18.56	45.09 / 32.21	40.73 / 29.03	29.09 / 20.11
	Tr-$Q^{1,5}$	48.68 / 32.65	44.47 / 29.65	**43.68 / 29.03**	44.96 / 31.83	40.61 / 28.52	**39.58 / 27.72**

Table 6. The **precision/recall** scores obtained upon evaluation on the original (Ref), Q^1 (weak adversary) and Q^{1+} (strong adversary) test set are shown in the columns for the sri/py150 and csn/python datasets.

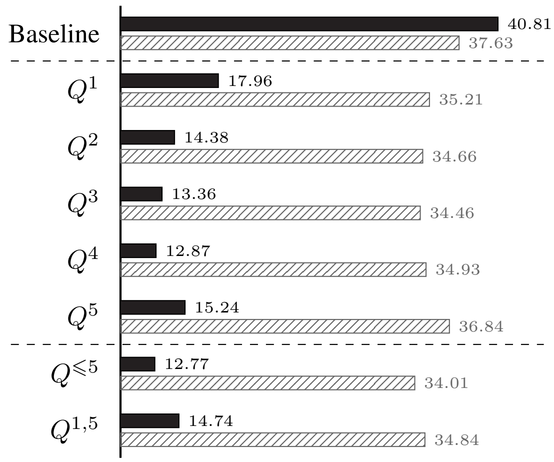
Model	Training	sri/py150			csn/python		
		Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}
seq2seq	Normal	31.17 / 35.19	16.95 / 18.57	13.91 / 14.93	23.05 / 27.39	8.54 / 10.74	4.80 / 6.11
	Tr-Q^1	29.99 / 33.70	27.11 / 31.01	26.34 / 30.25	22.62 / 26.82	19.91 / 24.44	19.30 / **23.88**
	Tr-$Q^{1,5}$	28.78 / 32.39	26.98 / 30.71	**26.61 / 30.37**	21.64 / 25.43	19.98 / 24.05	**19.64** / 23.75
code2seq	Normal	33.99 / 22.32	27.70 / 18.50	25.25 / 16.47	15.69 / 9.80	9.29 / 6.05	7.54 / 4.88
	Tr-Q^1	33.50 / 22.37	31.45 / 20.82	28.76 / 18.74	14.61 / 9.00	12.16 / 7.36	10.89 / 6.55
	Tr-$Q^{1,5}$	33.52 / 22.26	31.36 / 20.64	**31.22 / 20.51**	14.99 / 9.23	12.31 / 7.43	**11.95 / 7.23**

Table 7. The **exact match** scores obtained upon evaluation on the original (Ref), Q^1 (weak adversary) and Q^{1+} (strong adversary) test set are shown in the columns.

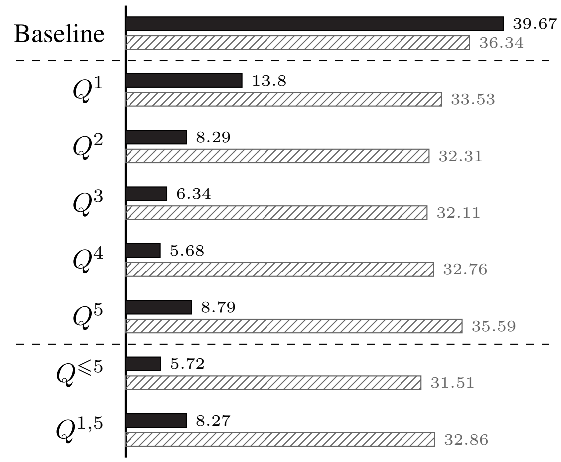
Model	Training	c2s/java-small			csn/java			sri/py150			csn/python		
		Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}	Ref	Q^1	Q^{1+}
seq2seq	Normal	13.91	1.70	1.12	15.90	2.85	1.63	20.26	8.88	6.66	11.05	3.58	2.26
	Tr-Q^1	10.97	8.38	7.96	14.72	11.25	10.59	19.15	16.16	15.36	10.88	8.82	8.37
	Tr-$Q^{1,5}$	10.85	8.85	**8.62**	13.77	11.17	**10.87**	18.00	16.22	**15.88**	10.17	8.83	**8.56**
code2seq	Normal	9.68	4.32	1.42	10.80	2.92	1.21	12.05	9.32	8.05	1.95	0.85	0.69
	Tr-Q^1	10.07	7.93	2.20	11.88	9.77	4.43	11.74	10.76	9.23	1.68	1.08	1.08
	Tr-$Q^{1,5}$	8.31	6.42	**5.87**	11.62	9.61	**8.72**	11.87	10.80	**10.51**	1.80	1.19	**1.14**

7.2. $\mathbf{Q^{1,5}}$ vs $\mathbf{Q^{\leqslant 5}}$: Full Results

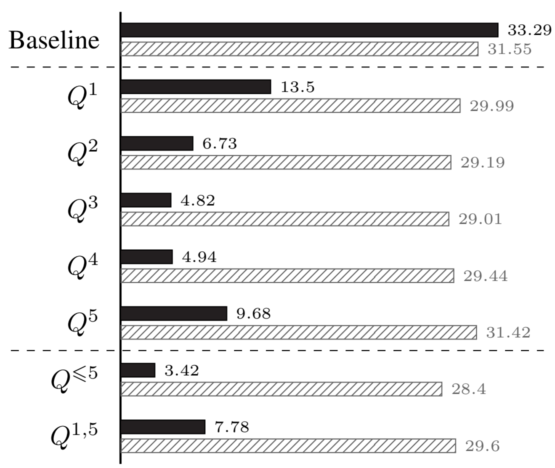
Figure 10 on the following page provides full results for our depth-k study. We note that the same general trends hold across all four of our datasets. Although mentioned briefly in the main paper, we would like to again note that the transformations used in this study were not all seven transforms introduced earlier in the paper, but instead just the five non-shuffle transforms. We limit the set of single-step transforms for this study to both reduce the amount of dataset expansion (from x128 for a set of 7 commutative transforms at a maximum depth of 7 to x32 for a set of 5 commutative transforms at a maximum depth of 5). This, and the non-determinism of our data pre-generation, is what causes slight differences in measurements such as those for Q^1 and $Q^{1,5}$ when comparing to the main table used in our robustness study.



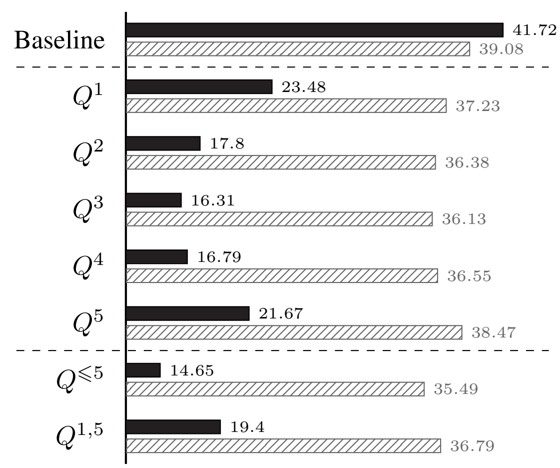
a. Performance of both the normally trained (solid) and Adv-$Q^{1,5}$ (dashed) seq2seq models on the c2s/java-small dataset



b. Performance of both the normally trained (solid) and Adv-$Q^{1,5}$ (dashed) seq2seq models on the csn/java dataset



c. Performance of both the normally trained (solid) and Adv-$Q^{1,5}$ (dashed) seq2seq models on the csn/python dataset



d. Performance of both the normally trained (solid) and Adv-$Q^{1,5}$ (dashed) seq2seq models on the sri/py150 dataset

Figure 10. The above plots provide an examination of depth-k attackers for each of our four datasets. We included the csn/java plot (b) in the main paper but, as expected, the general trend remains the same across all four datasets. Furthermore, the strength of an ideal attacker $Q^{\leqslant 5}$ is no more than 5 F1 points away from the strength of the $Q^{1,5}$ attacker we use for robust training in all four studies.

7.3. Details of IG Calculation

As described in the paper, the IG calculation yields a vector of attributions, with each element corresponding to the contribution of an input feature towards the output. Formally, given a function $F : \mathbb{R}^n \rightarrow \mathbb{R}$, an input x and a baseline input x', the $\text{IG}(x, x')$ is a vector $\in \mathbb{R}^n$, with its ith component defined as the path integral of gradients along the straight line path from the baseline x' to the input x. Mathematically:

$$\text{IG}_i(x, x') := (x_i - x'_i) \int_0^1 \frac{\partial F(x' + \alpha(x - x'))}{\partial x_i} d\alpha$$

In practice, this integral is estimated via the following Riemann summation:

$$\text{IG}_i(x, x') \approx \frac{(x_i - x'_i)}{m} \sum_{k=0}^{m} \frac{\partial F(x' + \frac{k}{m}(x - x'))}{\partial x_i}$$

We choose the number of steps in the Riemann sum (m) to be equal to 250 (the authors of IG recommend m between 20 and 300 to obtain a good approximation).

As described in the paper, we obtain attributions of each output token with respect to the input matrix of embeddings, using the zero embedding matrix as the baseline. As the baseline is all zeros, the above summation evaluates the derivative at scaled versions of the original embedding matrix. However, there arises a subtlety when calculating the IG at each time step of the decoder. Since the decoder is auto-regressive (future predictions depend upon the output of previous time steps), we need to provide the 'correct' tokens to the decoder at each time step (akin to teacher forcing during the training of the seq2seq model). This is necessary in order to make the prediction at each step differentiable with respect to the input.

The 'correct' tokens refer to the actual output predicted by the decoder when the program is the input. Thus in order to compute IG, we first pass the input tokens through the model to obtain the output. Then we pass scaled versions of the input embedding matrix to the model (this requires some additional hacking in the model encoder, as it ordinarily takes as input a sequence of token indices), and evaluate the derivative of the softmax outputs corresponding to the 'correct' tokens. The 'correct' tokens are also passed through the model simultaneously (for teacher forcing) with the necessary offset. The derivatives obtained are summed as per the IG formula to yield the attributions corresponding to elements in the input embedding matrix. These values are then summed column-wise to obtain token-level IG attributions. Our implementation of the IG calculation is made available anonymously in our repository at: https://anonymous.4open.science/r/4ac75222-3673-418e-8b86-9f36ba29fc59.

set

Source Token	IG (N)	IG (A)
(	0.014	-0.414
Job	-0.396	0.038
Priority	-0.475	0.280
priority	-0.149	-0.126
)	1.363	0.759
{	1.697	0.733
this	1.360	1.293
.	0.564	0.872
priority	-0.350	-0.307
=	2.713	1.974
priority	0.266	-0.173
;	0.087	-0.138
}	-0.185	-0.225
	-0.657	-0.285

priority

Source Token	IG (N)	IG (A)
(	-0.506	-1.646
Job	0.186	0.205
Priority	0.296	0.051
priority	1.639	1.178
)	-0.031	-0.317
{	-0.477	-0.177
this	-0.061	1.369
.	0.993	2.690
priority	2.685	4.433
=	0.933	-0.033
priority	2.316	1.726
;	0.239	0.276
}	-0.099	-0.065
	-0.132	-0.148

get

Source Token	IG (N)	IG (A)
(	1.592	-0.539
)	0.615	0.371
{	-0.798	0.475
return	3.762	4.152
class	0.055	-0.122
Name	0.293	0.106
;	-0.005	-0.001
}	0.299	0.047
	0.191	-0.061

class

Source Token	IG (N)	IG (A)
(	-0.205	-0.105
)	0.135	-0.025
{	0.146	0.979
return	2.129	0.028
class	2.534	6.235
Name	-0.913	-0.007
;	1.266	0.372
}	0.314	-0.182
	-0.124	0.180

name

Source Token	IG (N)	IG (A)
(	-0.468	-0.570
)	0.518	0.967
{	0.444	0.868
return	1.003	1.753
class	-0.655	-0.172
Name	0.582	-0.120
;	0.615	0.783
}	-0.264	0.103
	0.003	-0.092

Figure 11. An expanded visualization of IG scores under a normal (IG (N)) model and an adversarially trained (IG (A)) model. Here, and in Figure 12, negative IG scores are highlighted in purple. On the left, we show scores under both models for the task of predicting the method name `setStatus`. On the right, we show scores under both models for the task of predicting the method name `getClassName`. (We show negative scores in these examples to provide a more complete picture of the differences between models but it remains unclear how to qualitatively interpret these scores; therefore, for clarity, we often show only positive scores.)

```
( security expression handler < filter invocation > expression handler ) {
  assert . not null ( expression handler expression handler cannot be null ) ;
  this . expression handler = expression handler ;
  return this ;
}
```

a. Attribution scores under a normally trained model

```
( security expression handler < filter invocation > expression handler ) {
  assert . not null ( expression handler expression handler cannot be null ) ;
  this . expression handler = expression handler ;
  return this ;
}
```

b. Attribution scores under an adversarially trained model

Figure 12. Attribution scores (generated via Integrated Gradients) for a tokenized program under (left) a normally trained model and (right) an adversarially trained model on the task of code summarization. The coloration represents attribution values for each input word during the prediction of the output word `expression`.

```
( string msg ) {
base . set status ( msg ) ;
}
```

```
( string msg ) {
base . set status ( msg ) ;
}
```

a. Attribution scores under a normally trained model

```
( string msg ) {
base . set status ( msg ) ;
}
```

```
( string msg ) {
base . set status ( msg ) ;
}
```

b. Attribution scores under an adversarially trained model

Figure 13. Attribution scores (generated via Integrated Gradients) for a tokenized program under (left) a normally trained model and (right) an adversarially trained model on the task of code summarization. The coloration represents attribution values for each input word during the prediction of the output words: `set` and `status`. Here (and in Fig 2 in the main paper), we ignore negative attributions for greater clarity.

7.4. Focus on Semantic Features

Table 8 shows the additional results for ‘Increased IG for semantic features’. Across datasets, adversarially trained models have the highest average IG attributions for semantics features such as keywords and words.

Figures 8-11 show the full results for ‘Increased IG for specific tokens’. The top 25 pairs of (input token, output token) tuples which displayed the greatest absolute incerease in average IG after adversarial training are shown for all datasets. As it can be seen, adversarial training often results in increases which are logical and intuitive.

Table 8. Each table shows the input token classes in descending order of average IG, for the normally and adversarially trained models on the corresponding dataset (Avg. IG $\times$ 100 in brackets). Increased (▲) and decreased (▼) average IG shown in colors.

csn/java		csn/python		sri/py150	
Normal	Adv-$Q^{1,5}$	Normal	Adv-$Q^{1,5}$	Normal	Adv-$Q^{1,5}$
Keyword (0.73)	▲ Keyword (0.91)	Word (0.42)	▲ Word (0.43)	Word (0.90)	▲ Keyword (1.12)
Word (0.617)	▼ Word (0.616)	Keyword (0.23)	▲ Punctuation (0.32)	Operator (0.87)	▼ Word (0.85)
Other (0.33)	▼ Other (0.28)	Bracket (0.22)	▲ Keyword (0.31)	Keyword (0.79)	▼ Operator (0.73)
Number (0.20)	▲ Operator (0.11)	Punctuation (0.21)	▼ Other (0.03)	Other (0.40)	▲ Punctuation (0.64)
Bracket (0.14)	▲ Punctuation (0.10)	Other (0.10)	▲ Number (0.10)	Punctuation (0.32)	▲ Other (0.54)
Operator (0.10)	▼ Bracket (0.06)	Operator (0.07)	▲ Operator (0.08)	Bracket (0.31)	▼ Number (0.11)
Punctuation (0.08)	▼ Number (-0.05)	Number (0.06)	▼ Bracket (-0.09)	Number (0.20)	▼ Bracket (-0.09)

`super`	$\xrightarrow{\Delta 0.231}$	`service`	`return`	$\xrightarrow{\Delta 0.132}$	`get`	`super`	$\xrightarrow{\Delta 0.114}$	`get`
`checkpoint`	$\xrightarrow{\Delta 0.110}$	`task`	`super`	$\xrightarrow{\Delta 0.109}$	`up`	`sleep`	$\xrightarrow{\Delta 0.109}$	`wait`
`fail`	$\xrightarrow{\Delta 0.105}$	`assert`	`delete`	$\xrightarrow{\Delta 0.101}$	`tear`	`string`	$\xrightarrow{\Delta 0.095}$	`main`
`rand`	$\xrightarrow{\Delta 0.093}$	`random`	`run`	$\xrightarrow{\Delta 0.092}$	`main`	`unsupported`	$\xrightarrow{\Delta 0.088}$	`get`
`format`	$\xrightarrow{\Delta 0.081}$	`convert`	`reduce`	$\xrightarrow{\Delta 0.080}$	`set`	`int`	$\xrightarrow{\Delta 0.076}$	`num`
`init`	$\xrightarrow{\Delta 0.075}$	`service`	`io`	$\xrightarrow{\Delta 0.075}$	`test`	`instance`	$\xrightarrow{\Delta 0.073}$	`new`
`return`	$\xrightarrow{\Delta 0.073}$	`help`	`user`	$\xrightarrow{\Delta 0.070}$	`name`	`this`	$\xrightarrow{\Delta 0.068}$	`job`
`return`	$\xrightarrow{\Delta 0.068}$	`create`	`next`	$\xrightarrow{\Delta 0.062}$	`random`	`fully`	$\xrightarrow{\Delta 0.062}$	`delete`
`return`	$\xrightarrow{\Delta 0.061}$	`is`						

Figure 14. The top 25 (input token, predicted token) tuples with the highest absolute increase in mean IG scores for c2s/java-small

super $\xrightarrow{\Delta 0.231}$ service	return $\xrightarrow{\Delta 0.132}$ get	super $\xrightarrow{\Delta 0.114}$ get
checkpoint $\xrightarrow{\Delta 0.110}$ task	super $\xrightarrow{\Delta 0.109}$ up	sleep $\xrightarrow{\Delta 0.109}$ wait
fail $\xrightarrow{\Delta 0.105}$ assert	delete $\xrightarrow{\Delta 0.101}$ tear	string $\xrightarrow{\Delta 0.095}$ main
rand $\xrightarrow{\Delta 0.093}$ random	run $\xrightarrow{\Delta 0.092}$ main	unsupported $\xrightarrow{\Delta 0.088}$ get
format $\xrightarrow{\Delta 0.081}$ convert	reduce $\xrightarrow{\Delta 0.080}$ set	int $\xrightarrow{\Delta 0.076}$ num
init $\xrightarrow{\Delta 0.075}$ service	io $\xrightarrow{\Delta 0.075}$ test	instance $\xrightarrow{\Delta 0.073}$ new
return $\xrightarrow{\Delta 0.073}$ help	user $\xrightarrow{\Delta 0.070}$ name	this $\xrightarrow{\Delta 0.068}$ job
return $\xrightarrow{\Delta 0.068}$ create	next $\xrightarrow{\Delta 0.062}$ random	fully $\xrightarrow{\Delta 0.062}$ delete
return $\xrightarrow{\Delta 0.061}$ is		

Figure 15. The top 25 (input token, predicted token) tuples with the highest absolute increase in mean IG scores for csn/java

writes $\xrightarrow{\Delta 0.075}$ write	lexpos $\xrightarrow{\Delta 0.073}$ t	deletes $\xrightarrow{\Delta 0.069}$ delete
initialize $\xrightarrow{\Delta 0.065}$ init	gets $\xrightarrow{\Delta 0.054}$ get	reads $\xrightarrow{\Delta 0.049}$ read
program $\xrightarrow{\Delta 0.048}$ p	del $\xrightarrow{\Delta 0.047}$ remove	creates $\xrightarrow{\Delta 0.046}$ create
the $\xrightarrow{\Delta 0.046}$ set	list $\xrightarrow{\Delta 0.045}$ p	check $\xrightarrow{\Delta 0.043}$ visit
arguments $\xrightarrow{\Delta 0.043}$ args	removes $\xrightarrow{\Delta 0.042}$ remove	retrieves $\xrightarrow{\Delta 0.041}$ get
from $\xrightarrow{\Delta 0.040}$ init	self $\xrightarrow{\Delta 0.039}$ t	super $\xrightarrow{\Delta 0.039}$ get
cli $\xrightarrow{\Delta 0.037}$ get	validation $\xrightarrow{\Delta 0.037}$ validate	an $\xrightarrow{\Delta 0.035}$ set
creating $\xrightarrow{\Delta 0.035}$ create	vcard $\xrightarrow{\Delta 0.034}$ get	a $\xrightarrow{\Delta 0.034}$ set
search $\xrightarrow{\Delta 0.033}$ find		

Figure 16. The top 25 (input token, predicted token) tuples with the highest absolute increase in mean IG scores for csn/python

super $\xrightarrow{\Delta 0.465}$ parser	super $\xrightarrow{\Delta 0.408}$ context	get $\xrightarrow{\Delta 0.286}$ calculate
gz $\xrightarrow{\Delta 0.264}$ grad	model $\xrightarrow{\Delta 0.256}$ predictions	model $\xrightarrow{\Delta 0.226}$ calculate
starttag $\xrightarrow{\Delta 0.216}$ visit	stf $\xrightarrow{\Delta 0.207}$ create	delete $\xrightarrow{\Delta 0.206}$ backwards
down $\xrightarrow{\Delta 0.189}$ tear	fastbinary $\xrightarrow{\Delta 0.174}$ write	super $\xrightarrow{\Delta 0.173}$ data
return $\xrightarrow{\Delta 0.167}$ specific	return $\xrightarrow{\Delta 0.162}$ byte	sublime $\xrightarrow{\Delta 0.156}$ run
super $\xrightarrow{\Delta 0.155}$ init	locals $\xrightarrow{\Delta 0.146}$ call	return $\xrightarrow{\Delta 0.145}$ repr
return $\xrightarrow{\Delta 0.140}$ action	removes $\xrightarrow{\Delta 0.139}$ remove	apply $\xrightarrow{\Delta 0.138}$ make
on $\xrightarrow{\Delta 0.138}$ branch	replay $\xrightarrow{\Delta 0.138}$ test	super $\xrightarrow{\Delta 0.127}$ down
return $\xrightarrow{\Delta 0.127}$ iter		

Figure 17. The top 25 (input token, predicted token) tuples with the highest absolute increase in mean IG scores for sri/py150